**Санкт-Петербургский государственный университет информационных технологий, механики и оптики**

# Учебно-методическое пособие по дисциплине «Анализ и проектирование на UML»

Новиков Ф.А.,
канд. физ.-мат. наук, доцент кафедры «Технологии программирования»

**Санкт-Петербург**
**2007**

## Оглавление

# Введение

Учебное пособие «Анализ и проектирование на UML» содержит полное описание унифицированного языка моделирования UML и набор рекомендаций по применению языка для анали и проектирования программных систем.

Как опытному преподавателю, автору твердо известно, что существуют три уровня понимания обучающимся нового предмета, которые характеризуются следующими признаками:

1. возникает приятное чувство понимания;
2. может повторить своими словами;
3. видит ошибки.

Данная книга написана на третьем уровне и адресована обучающимся, ориентированным на третий уровень. Многие неловкости, неудобства и даже ошибки UML обсуждаются в этой книге и предлагаются авторский подход к их исправлению.[1]

[1] Не исключено, что авторские предложения также содержат новые, более тяжелые ошибки.

# Тема 1. Введение в UML

- Что обозначает аббревиатура UML?
- Кому и зачем нужен UML?
- Что послужило причиной возникновения UML?
- Для чего используют UML на практике?
- Как определен UML?
- Из чего состоит UML?
- Что такое модель UML?
- Из каких элементов состоит модель?
- Как комбинируются элементы модели?
- Что делать, если готовых элементов не хватает?
- Какова общая структура модели?

## 1.1. Что такое UML?

Предметом этой книги является UML в целом. Прежде чем обсуждать что-либо «в целом», резонно сначала точно определить, о чем идет речь в частности. Обсуждаемый предмет обозначается идентификатором UML, который является аббревиатурой полного названия Unified Modeling Language. Правильный перевод этого названия на русский язык — унифицированный язык моделирования. Таким образом, обсуждаемый предмет характеризуется тремя словами, каждое из которых является точным термином.

### *1.1.1. UML — это язык*

Главным словом в этом сочетании является слово "язык".

| Язык — это знаковая система для хранения и передачи информации. |
|---|

Различаются языки *формальные*, правила употребления которых строго и явно определены и *неформальные*, употребление которых основано на сложившейся практике. Различаются также языки *естественные*, появляющиеся как бы сами собой[2] в результате неперсонифицированных усилий массы людей и языки *искусственные*, являющиеся плодом видимых усилий определенных лиц. Филологи, наверное, смогут назвать еще дюжину различных характеристик: нормативный и ненормативный язык, живой и мертвый, синтетический и аналитический и т. д.

**Что определяют определения?**

На самом деле, разумеется, определения ничего не определяют. Но броско и ёмко составленная формулировка может породить у читателя правильные ассоциации и стимулировать понимание. В частности, в предшествующем абзаце словосочетания "знаковая система", "строгое определение" и другие сами нуждаются в определениях, однако, любому здравомыслящему человеку ясно, что такая иерархия определений никуда не приведет — где-то придется остановиться, не дойдя до конца. Видимо, чем раньше, тем лучше. Одного-двух

[2] Точнее говоря, природа этого явления до конца не изучена.

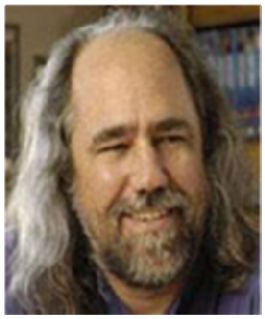
определений достаточно, чтобы перейти к примерам — самому надежному и доходчивому способу аргументации.

Например, язык, на котором написана эта книга (мы полагаем, что это русский язык) является неформальным и естественным. С другой стороны, подавляющее большинство языков программирования являются формальными и искусственными. Встречаются и другие комбинации: например, язык алгебраических формул мы считаем формальным и естественным, а эсперанто — неформальным искусственным.

**ЗАМЕЧАНИЕ**

Мы предполагаем, что читатель по меньшей мере наслышан о языках программирования.

Так вот, UML можно охарактеризовать как формальный искусственный язык, хотя и не в такой степени, как многие распространенные языки программирования. Признаком искусственности служит наличие трех общепризнанных авторов (рис. 1.1):



Grady Booch
Грэди Буч

James Rumbaugh
Джеймс Рамбо

Ivar Jacobson
Айвар Якобсон

**Рис. 1.1.Авторы UML.**

В то же время в формирование языка внесли вклад многие теоретики и разработчики, имя которым легион. Языкотворческая практика применительно к UML непрерывно продолжается (мы это обсудим позднее), что дает основание считать UML до некоторой степени естественным языком. Описание UML по большей части формальное, но содержит и явно неформальные составляющие. Такие особенности UML как *точки вариации семантики* и стандартные *механизмы расширения* (см. раздел 1.7.4), заметно отличают UML от языков, которые, по общему мнению, являются образцами формализма.

**ЗАМЕЧАНИЕ**

В этой книге обсуждаются конкретные версии UML, для которых имеются утвержденные международные стандарты. Однако наличие стандарта — это еще не основание считать язык формальным и искусственным.

Для описания формальных искусственных языков (в частности, для описания языков программирования) придумано и используется множество различных способов. Однако на практике сложилась общепринятая структура таких описаний. Считается, что формальный искусственный язык описан должным образом, если это описание содержит по меньшей мере следующие части.

- Синтаксис то есть определение правил конструирования выражений языка
- Семантика, то есть определение правил приписывания смысла выражениям языка.
- Прагматика, то есть определение правил использования выражений языка для достижения определенных целей.

Как формальный искусственный язык UML имеет синтаксис, семантику и прагматику, хотя эти части названы в некоторых случаях иначе и описаны по другому, нежели это принято в языках программирования. Тому есть очевидная причина, указанная в следующем разделе.

### *1.1.2. UML — это язык моделирования*

Слово "моделирование", входящее в название UML, имеет множество смысловых оттенков и сложившихся способов употребления. Понятно, что когда мы моделируем, мы что-то такое делаем с моделями, но не совсем понятно, с какими моделями и что именно делаем. Нам представляется необходимым остановиться чуть подробнее на смысле слов "модель" и "моделирование" в контексте UML, в противном случае есть риск упустить главное, погрузившись в технические детали. Начать придется издалека и, на первый взгляд, несколько сбоку: с обсуждения понятий *жизненный цикл* и *процесс разработки* программной системы.

Во Введении мы кратко упомянули, и здесь еще раз повторяем, что UML имеет отношение прежде всего и главным образом к созданию и применению компьютерных программ.

**ЗАМЕЧАНИЕ**

> Словосочетания "компьютерная программа", "программный продукт", "программная система" мы считаем, во-первых, известными и понятными читателю, и, во-вторых, синонимами, вместо которых всюду в дальнейшем для краткости будет использоваться термин "приложение".

Давно замечено, что приложение за время жизни претерпевает многочисленные изменения своей формы, зависящие от состояния процесса разработки и эксплуатации приложения. Обычно совокупность и последовательность этих изменений называется *жизненный цикл*. В разных парадигмах и технологиях программирования понятие жизненного цикла определяется и трактуется немного по разному, но в общем близко к схеме, представленной на рис. 1. Важно подчеркнуть, что за время своей жизни программа проходит метаморфозы, как правило, несколько раз,[3] т.е. это именно цикл, причем не один, а несколько.

[3] Чего, к сожалению, не случается с программистами.

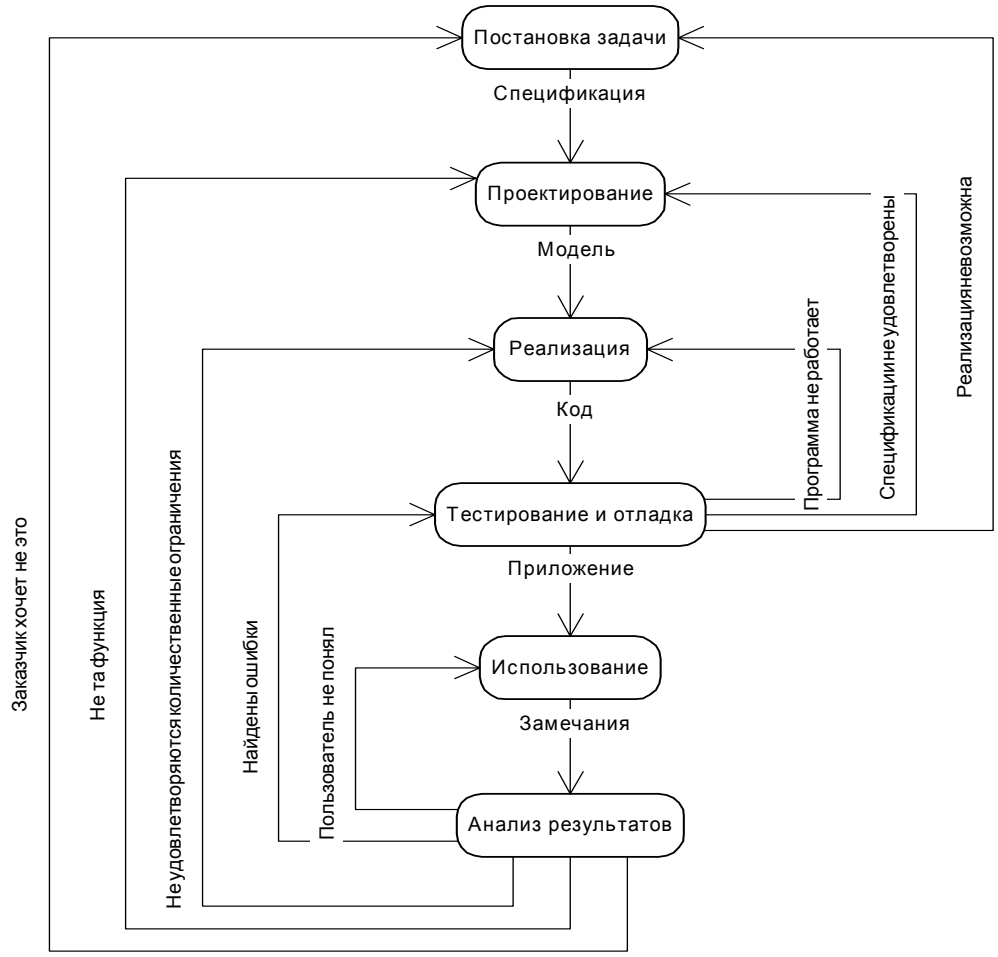



**Рис. 1.2. Жизненный цикл приложения**

## ЗАМЕЧАНИЕ

Приложение является преимущественно идеальным (а не материальным) объектом, поэтому в жизненный цикл включаются и те "эмбриональные" формы, в которых приложение существует в начале разработки. С другой стороны, материально существующие на магнитных носителях старые программы для больших машин не используются и не сопровождаются, а потому их жизненный цикл можно считать завершенным. Таким образом, приложение живо, покуда оно живо в голове у программиста и/или пользователя.

С понятием жизненного цикла приложения тесно связано понятие *процесса разработки* приложения, т.е. определенной последовательности сменяющих друг друга видов деятельности. В обыденном языке слово "разработка" подразумевает создание чего-то, а когда это что-то (в данном случае, приложение) создано, то разработка закончена. Вы прекрасно понимаете, что в случае приложений, после того, как разработка закончена, начинается самое интересное — эксплуатация, которая оказывается теснейшим образом связанной с разработкой. Для приложений отделять друг от друга разработку и эксплуатацию, а тем более противопоставлять эти понятия, было бы в корне неверно.

**ЗАМЕЧАНИЕ**

> Был даже придуман специальный термин — *продолжающаяся разработка*, который включает в себя как разработку в обычном смысле, так и модификацию программы (и другие действия) в процессе эксплуатации. Для краткости мы предлагаем вам понимать термин "процесс разработки" в широком смысле, включая в него все действия, которые производятся с приложением – проектирование, кодирование, отладка, внедрение и т. д.

В результате понятия жизненного цикла и процесса разработки можно считать равнообъемными, даже можно сказать, что это две стороны одного понятия. Когда мы говорим "жизненный цикл", мы смотрим на предмет как бы с точки зрения программы, которая пассивно переживает одно состояние за другим, а когда мы говорим "процесс разработки", мы смотрим на тот же самый предмет с точки зрения программиста, который активно выполняет одно действие за другим.

Жизненный цикл и процесс разработки каждого конкретного приложения индивидуальны, и потому они не представляют большого интереса. Для широкого круга потребителей более интересно рассмотреть *модели* жизненного цикла и процесса разработки.

**ЗАМЕЧАНИЕ**

> Здесь слово "модель" используется в обычном смысле – как абстрактное описание некоторого явления, в котором существенные закономерности учтены, а несущественные детали опущены. Не следует путать это с моделями UML, о которых идет речь в других местах книги.

Модель жизненного цикла и модель процесса разработки взаимно определяют друг друга и являются согласованными. В каждой организации, которая специализируется на разработке программного обеспечения, и даже у особо продвинутых программистов-одиночек (далее такая организация или человек называются обобщенно — *разработчик*), существуют свои собственные модели жизненного цикла и процесса разработки. Конечно, в большинстве случаев они оказываются очень близки, но все-таки имеют некоторые индивидуальные особенности. Мы рассматриваем те модели жизненного цикла и процесса разработки, которые используем сами, стараясь, по возможности, опускать влияние на них наших собственных индивидуальных особенностей. От этого наше рассмотрение становится более абстрактным, но, одновременно, более широко применимым. Вы можете приспосабливать наши модели к своим нуждам, меняя их в соответствии с конкретными обстоятельствами.

Мы используем модель процесса разработки приложений, основанную на понятиях *фаза* и *веха*. Большинство современных моделей процесса разработки также основаны на этих понятиях.

*Веха* — это одномоментное идентифицируемое событие в процессе разработки, сопровождающееся появлением и фиксацией некоторого отчуждаемого артефакта (документа, программы, и др.). *Фаза*[4] — это часть процесса, во время которой выполняются определенные функции с целью достижения определенной вехи. Таким образом, веха наступает в конце фазы и является определяющим признаком фазы. Очень важным является то обстоятельство, что фазы и вехи не обязательно

[4] Иногда с той же целью используют слова "стадия" или "этап". В этой книге используется слово "фаза", потому что оно кажется нам более выразительным.

вытянуты в линейную последовательность. Т.е. некоторые фазы могут полностью или частично перекрываться во времени.

### Водопадная модель процесса разработки

Одной из самых первых моделей процесса разработки была так называемая модель "водопада", или модель "конвейера". В этой модели процесс разработки носит линейный характер. Он также делится на фазы, но фазы сменяют друг друга строго последовательно (так, как выполняются производственные операции над изделием на конвейере). Таким образом, в процессе разработки программа как бы опускается с более высоких, абстрактных уровней (ТЗ, эскизный проект и т. п.), на все более низкие детальные уровни (код, данные в базе и т. п.). Образно говоря, поток разработки направлен в одну сторону, вниз. Отсюда и происходит название этой модели.

Большинство используемых в настоящее время моделей разработки носят циклический характер (так называемая *итеративная* или инкрементальная разработка). Например, в этой книге мы используем простую модель, представленную на рис. 1.2.

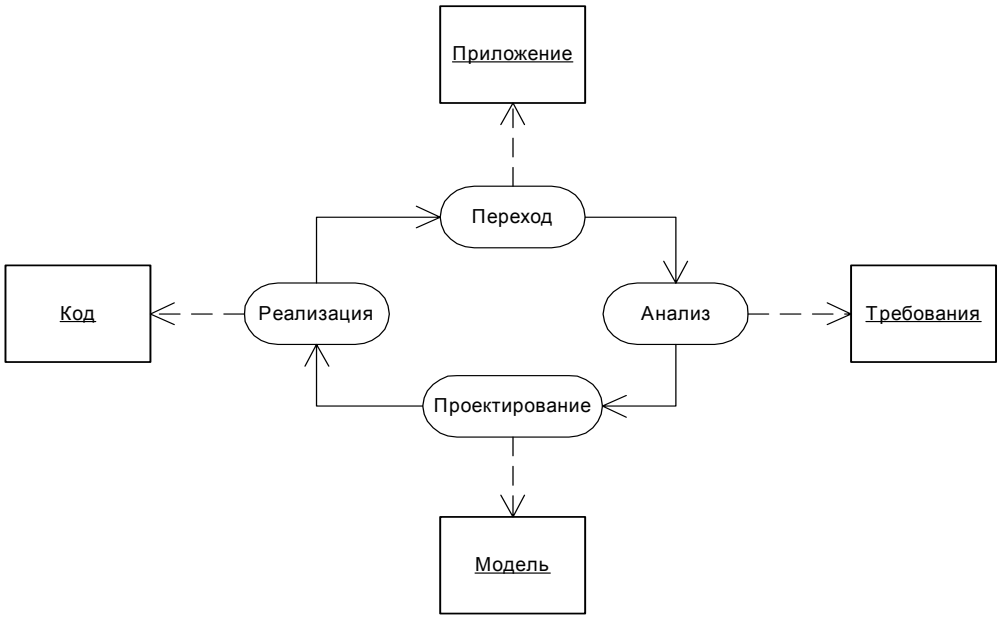



**Рис. 1.3. Итеративный процесс разработки**

Оставим пока в стороне детальное обсуждение рис. 1.2 и 1.3 и остановимся только на одном моменте: на наличии состояния в жизненном цикле и фазы в процессе разработки, которые у нас названы одним словом "проектирование".

### О пользе чертежей

В литературе по технологии программирования часто используется одна аналогия, которую мы считаем довольно убедительной и хотим повторить.

Рассмотрим такую область человеческой деятельности, как архитектура и строительство. В проектировании и строительстве домов (по индивидуальному проекту) есть много общего с разработкой

приложений. В обоих случаях требуется как творческий подход, так и большой объем рутинной работы; применяются как формальные, так и неформальные методы; очень многое зависит от опыта и квалификации разработчика; конечный результат оценивается степенью удовлетворенности заказчика. Таким образом, аналогия имеет место.

Но есть одно различие, которое сразу бросается в глаза. В архитектуре и строительстве очень широко используются *чертежи*. Чертежи разные — рисунки архитектора с общим видом будущего здания, детальные строительные чертежи, по которым ведется строительство, различные вспомогательные схемы инженерных коммуникаций и др. Конечно, садовый домик[5] можно построить без всяких чертежей, "на глазок", но с чертежами обычно получается все-таки лучше. Наверное, можно попробовать построить и трехэтажный дом, не пользуясь чертежами, хотя результат вряд ли будет надежным и красивым. Но высотный дом нельзя построить без тщательного предварительного проектирования, учета строительных норм и правил (СНИП) и составления огромного количества разнообразных чертежей.

Между тем, при разработке приложений слишком часто приходиться наблюдать, как неопытные разработчики проскакивают фазу проектирования и получив техническое задание, сразу приступают к реализации, т.е. начинают "класть кирпичи". Если при этом спросить их: "А где же чертеж будущего приложения?", то они даже не понимают вопроса. Если речь идет о приложении типа "садовый домик", то такой подход может сработать — помогут опыт и чутье. Но если нужно построить высотный дом? Без чертежей не обойтись! В солидном приложении деталей (функций, процедур, модулей, форм, элементов управления, операторов) не меньше, а больше, чем в высотном доме отдельных строительных деталей. Отсюда и наблюдаемое соотношение результативности: случай обрушения дома из-за ошибок проектирования — это ЧП, которое случается очень редко. Что же касается разработки приложений, то по данным некоторых авторов, свыше половины проектов по разработке оканчиваются неудачей: не доводятся до конца, прекращаются из-за перерасхода времени и средств, имеют неудовлетворительный результат и т.д. Анализ показывает, что в подавляющем большинстве случаев причина неудач кроется в плохом проектировании.

Одним из объективных факторов, объясняющих такое положение дел, является сравнительная молодость программирования, как инженерной дисциплины. Архитекторы и строители накапливали опыт тысячелетиями, а чертежами пользуются уже столетия. У них было время придумать понятную, удобную и надежную систему обозначений. История разработки приложений насчитывает всего полвека. Сейчас только-только появляются системы обозначений, сравнимые по выразительности и удобству со строительными чертежами (и наиболее перспективным нам представляется UML).

---

[5] Авторы UML повторяют эту аналогию из книги в книгу, обсасывая её на разные лады. Правда, в качестве альтернативы небоскребу они рассматривают собачью конуру, но у нас пусть будет садовый домик — мы хотим пощадить чувства тех, кто еще не использует UML.

Другими словами, ощущается дефицит инженерно проработанных средств проектирования приложений. Это обстоятельство создает дополнительные трудности для разработчиков, но не может служить оправданием для безответственного "проскакивания" фазы проектирования.

В процессе проектирования что-то делается и в результате нечто получается. Если эта деятельность и форма результата регламентированы определенным образом, то им уместно дать название. Так сложилось, что результат проектирования (и анализа), оформленный средствами определенного языка принято называть моделью.[6] Деятельность по составлению моделей естественно назвать моделированием. Именно в этом смысле UML является языком моделирования.

**Modeling vs. simulation**

Как уже отмечалось, слово "моделирование" многозначно. В частности, английские слова modeling и simulation оба переводятся как моделирование, хотя означают разные вещи. В первом случае речь идет о составлении модели, которая используется только для описания моделируемого объекта или явления. Во втором случае подразумевается составление модели, которая может быть использована для получения существенной информации о моделируемом объекте или явлении. Во втором случае обычно добавляется уточняющее прилагательное: численное, математическое и др. UML является языком моделирования в первом смысле, хотя автору известны некоторые успешные попытки использования UML и во втором смысле. Прочие вариации смысла слова моделирование оставлены за рамками книги. В частности, названия профессий: модельер, модельщик и моделист не имеют отношения к предмету. Тот, кто составляет модели UML по-русски никак не называется (пока).

Таким образом, модель UML — это, прежде всего, основной артефакт фазы проектирования, а также и кое-что другое, а именно все, что авторам UML удалось включить в язык, не нарушая принципа, к изложению которого мы переходим в следующем разделе.

### *1.1.3. UML — это унифицированный язык моделирования*

Описывая историю создания UML, его авторы характеризуют эпоху до UML как период "войны методов". Пожалуй, "война" — это слишком сильно сказано, но, действительно, UML является отнюдь не первым языком моделирования. К моменту его появления насчитывались десятки других, различающихся системой обозначений, степенью универсальности, способами применения и т. д. Авторы языков и теоретики программирования препирались между собой, выясняя чей подход лучше, а практические программисты всю эту "войну методов" равнодушно игнорировали, поскольку ни один из методов не дотягивал до уровня индустриального стандарта.

---

[6] Что вполне согласуется с наиболее общим смыслом слова модель — абстрактное описание чего-либо.

Толчком к изменению ситуации послужили обстоятельства. Во-первых, массовое распространение получил *объектно-ориентированный подход* к программированию (ООП), в результате чего возникла потребность в соответствующих средствах. Другими словами, появления чего-то, подобного UML с нетерпением ждали практики. Во-вторых, три крупнейших специалиста в этой области, авторы наиболее популярных методов, решились объединить усилия именно с целью унификации своих (и не только своих) разработок в соответствии с социальным заказом.

### Объектно-ориентированное программирование

Наиболее популярная в настоящее время методология программирования, являющаяся развитием структурного программирования. Центральной идеей ООП является *инкапсуляция*, т. е. структурирование программы на модули особого вида, объединяющего данные и процедуры их обработки, причем внутренние данные модуля могут быть обработаны только предусмотренными для этого процедурами. В разных вариациях ООП этот модуль называют по-разному: класс, абстрактный тип данных, кластер и др. Каждый такой *класс* имеет внутреннюю часть, называемую реализацией (или представлением), и внешнюю часть, называемую интерфейсом. Доступ к реализации возможен только через интерфейс. Обычно в интерфейсе различают свойства (которые синтаксически выглядят как переменные) и методы (которые синтаксически выглядят как процедуры или функции). Класс может иметь методы, называемые конструкторами и деструкторами, позволяющие во время выполнения программы динамически порождать и уничтожать *экземпляры класса*. Экземпляры одного класса сходны между собой (например, наследуют методы класса), но имеют различия (например, имеют разные значения свойств). Классы и экземпляры классов называют *объектами*, откуда и происходит название объектно-ориентированное программирование.

Приложив заслуживающие уважения усилия авторам UML (рис. 1.1) при поддержке и содействии всей международной программистской общественности удалось свести воедино (унифицировать) большую часть того, что было известно им и до них. В результате унификации получилась теоретически изящная и практически полезная вещь — UML.

Если попытаться проследить историю возникновения и развития элементов UML, как на уровне основополагающих идей, так и на уровне технических деталей, то пришлось бы назвать сотни имен и десятки организаций. Мы не будем этого делать, и не только из экономии места, но и потому, что история развития UML отнюдь не завершена — язык постоянно совершенствуется, обогащается и расширяется.[7] Мы полагаем достаточным воспроизвести на рис. 1.4. картинку, которой авторы UML обычно иллюстрируют историю своего детища.

[7] Вы тоже можете принять в этом участие.

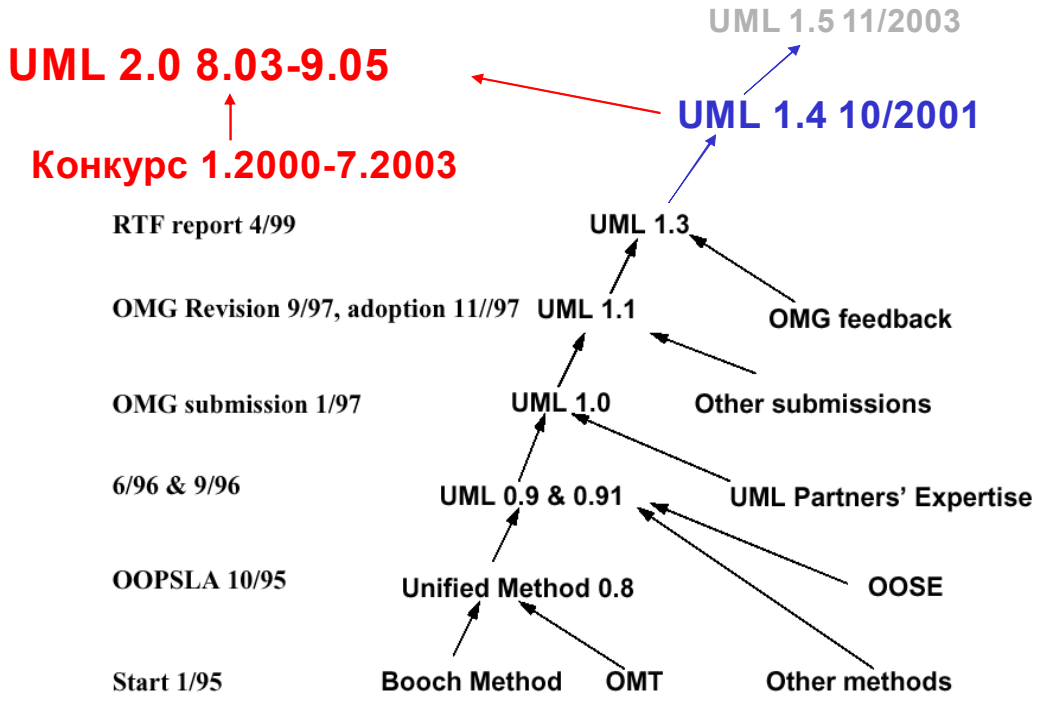



**Рис. 1.4. История развития UML**

**ЗАМЕЧАНИЕ**

> UML — это *унифицированный* язык моделирования, но никак не единый и не универсальный (мы видели такие ошибочные толкования первой буквы U в некоторых источниках). Далее мы еще не раз вернемся к этому вопросу.

## 1.2. 1.2. Назначение UML

UML предназначен для моделирования. Сами авторы UML определяют его как графический язык моделирования общего назначения, предназначенный для спецификации, визуализации, проектирования и документирования всех артефактов, создаваемых при разработке приложений. Мы полностью согласны с этим определением, и не только одобряем выбор ключевых слов (см. врезку "Что определяют определения?"), но придаем большое значение порядку, в котором они перечислены.

### *1.2.1. Спецификация*

В типичных случаях разработки приложений участвуют по меньше мере два действующих лица: *заказчик* (конкретный человек, или группа лиц, или организация) и *разработчик* (это может быть программист-одиночка, временная команда проекта или целая организация, специализирующаяся на разработке приложений). Из-за того, что действующих лиц двое, очень многое зависит от степени их взаимопонимания.

Одним из ключевых этапов разработки приложения является определение того, что, собственно, должно делать разрабатываемое приложение. В результате этого этапа появляется формальный или неформальный документ (артефакт), который

называют по-разному, имея в виду примерно одно и то же: постановка задачи, пользовательские требования, техническое задание, внешние спецификации и др. Аналогичные по назначению, но, может быть, отличные по форме и содержанию артефакты появляются и на других этапах разработки, особенно если в разработку включено много действующих лиц. Для них также используются различные названия: функциональные спецификации, архитектура приложения и др. Мы будем все такие артефакты называть спецификациями.

| |
|---|
| *Спецификация* — это декларативное описание того, как нечто устроено или работает. |

Необходимо принимать во внимание три толкования спецификаций.

- То, которое имеет в виду действующее лицо, являющееся источником спецификации (например, заказчик).
- То, которое имеет в виду действующее лицо, являющееся потребителем спецификации (например, разработчик).
- То, которое объективно обусловлено природой специфицируемого объекта.

Эти три трактовки спецификаций могут не совпадать, и, к сожалению, как показывает практика, сплошь и рядом не совпадают, причем значительно. Заказчик может не осознавать своих объективных потребностей, или неверно их интерпретировать, или заблуждаться относительно природы своих затруднений, пытаясь с помощью заказного офисного приложения лечить симптомы, а не причину болезни своего бизнеса. Разработчик может не разбираться в предметной области заказчика и интерпретировать формулировки спецификаций совершенно превратным образом. Если же в формулировке спецификаций участвует разработчик, то злоупотребление технической терминологией может совершенно дезориентировать заказчика.

### О формальных спецификациях

Идеальным было бы формулировать спецификации с математической строгостью, так чтобы сам язык спецификаций исключал неоднозначности в описании функций разрабатываемого приложения.

Математикам известны средства такого рода, например, язык исчисления предикатов. Действительно, если задать с помощью логического выражения на языке исчисления предикатов условие, которому должны удовлетворять входные данные приложения (так называемое *предусловие*) и условие, которому должны удовлетворять выходные данные приложения (*постусловие*), то не остается места для неоднозначности в определении того, делает приложение то, что нужно, или нет. Как говорят математики "не нужно спорить, давайте вычислять".

Например, предусловие $a \neq 0 \ \& \ b^2-4ac>0$ вместе с постусловием $ax^2+bx+c=0$ является исчерпывающим ТЗ на разработку приложения

"Решение квадратного уравнения" (входные данные `a`, `b`, `c`, выходное данное `x`).[8]

Если для рассматриваемой предметной области действительно существует строгая математическая модель, то формальная спецификация является наиболее предпочтительной. Именно такие спецификации используют при разработке приложений в инженерных областях.

К сожалению, в большинстве предметных областей сколько-нибудь проработанных математических моделей нет, степень формализации очень низка, и навыки использования формальных математических методов не очень распространены. При попытке применить формальные методы объем спецификации оказывается намного больше объема самого специфицируемого приложения, а разработка спецификаций оказывается гораздо более трудоемкой (и требующей более высокой квалификации!), чем разработка приложения.

По этому поводу в программистском фольклоре бытует следующее выражение: наиболее полной, точной и краткой спецификацией программы является текст программы. Для математических программ с этим можно поспорить, но для офисных приложений это безусловно так.

Основное назначение UML — предоставить, с одной стороны, достаточно формальное, с другой стороны, достаточно удобное, и, с третьей стороны, достаточно универсальное средство, позволяющее до некоторой степени снизить риск расхождений в толковании спецификаций.

### *1.2.2. Визуализация*

Известная поговорка гласит, что лучше один раз увидеть, чем сто раз услышать. Мы добавим: тем паче тысячу раз прочитать. Особенности человеческого восприятия таковы, что текст с картинками воспринимается легче, чем голый текст. А картинки с текстом (это называется "комиксы") — еще легче. Модели UML допускают представление в форме картинок, причем эти картинки наглядны, интуитивно понятны, практически однозначно интерпретируются и легко составляются. Фактически, развитие и детализация этого тезиса составляет большую часть содержания остальной части книги. Мы не будем забегать вперед, и просто приведем пример без всяких объяснений (рис. 1.4). Разве что-нибудь непонятно?

[8] Нужно еще указать, что все числа вещественные и задать точность, с которой должен быть определены корни, но это уже детали.

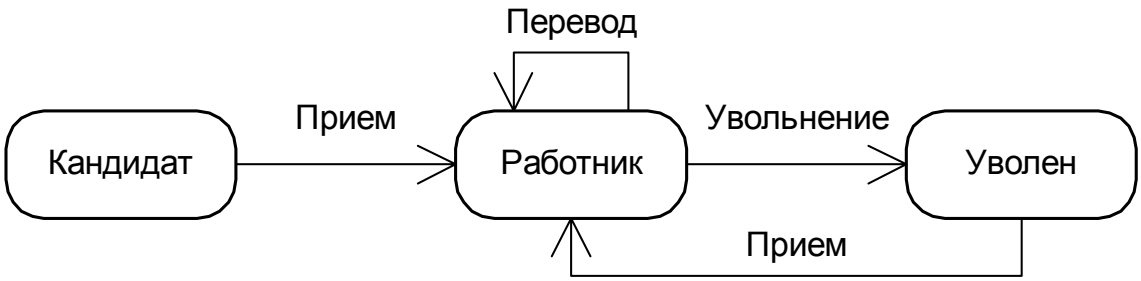



**Рис. 1.4. Жизненный цикл работника на предприятии**

Таким образом, второе по важности назначение UML состоит в том, чтобы служить адекватным средством коммуникации между людьми. Разумеется, наглядность визуализации моделей UML имеет значение, только если они должны составляться или восприниматься человеком — это назначение UML не имеет отношения к компьютерам.

**ЗАМЕЧАНИЕ**

> Графическое представление модели UML не тождественно самой модели. Это важное обстоятельство часто упускается из виду при первом знакомстве с UML.

### *1.2.3. Проектирование*

В оригинале данное назначение UML определено с помощью слова construct, которое мы передаем осторожным термином "проектирование". Речь идет о том, что UML предназначен не только для описания абстрактных моделей приложений, но и для непосредственного манипулирования артефактами, входящими в состав этих приложений, в том числе такими, как программный код. Другими словами, одним из назначений UML является, например, создание таких моделей, для которых возможна автоматическая генерация программного кода (точнее, фрагментов кода) соответствующих приложений. Более того, природа моделей UML такова, что возможен и обратный процесс: автоматическое построение модели по коду готового приложения.[9]

Сказанное в предыдущем абзаце требует оговорок "до некоторой степени", "в известной мере" буквально после каждого утверждения. Самое досадное, что в данный момент точно указать "степень" и "меру" не представляется возможным. Причина не в том, что никто не удосужился этим заняться, а в том, что это очень трудная задача.

#### Автоматический синтез программ

> Синтезом программ называется автоматическая генерация программы по некоторой спецификации. В зависимости от метода спецификации автоматический синтез программ подразделяют на несколько категорий.

[9] Это называется по-английски reverse engineering и обычно переводится на русский как "обратное проектирование". Нам этот перевод категорически не нравится: какое же это проектирование и почему оно обратное? Но другого термина мы не знаем, а потому вовсе не будем его употреблять.

Если спецификация задана в виде формальных логических условий, связывающих входные и выходные данные, то говорят о дедуктивном синтезе программ. Например, если спецификация задана с помощью предусловия (см. врезку "о формальных спецификациях") `P(x)` и постусловия `Q(x,y)`, связывающего входные данные `x` с выходными данными `y`, то из конструктивного доказательства теоремы существования $\forall x(P(x) \rightarrow \exists y Q(x,y))$ может быть автоматически извлечена программа вычисления `y` по `x`. К сожалению, задача автоматического доказательства теорем во всех смыслах трудная, так что тот факт, автоматический синтез программ сводится к автоматическому доказательству теорем не слишком обнадеживает.

Если спецификация задана в виде набора примеров (то есть набора пар входных и соответствующих им выходных данных), то говорят об индуктивном синтезе программ. Известны классы программ, для которых можно указать, сколько и каких примеров достаточно для того, чтобы однозначно синтезировать программу. Поясним это аналогией из геометрии: если известны координаты двух точек на прямой, то не трудно получить уравнение этой прямой. Но если про интересующую нас фигуру (линию) ничего не известно, то никаким конечным множеством точек (то есть примеров) задать ее нельзя.

Все остальные виды формального, но неалгоритмического описания задачи, например, в виде моделей UML, относят к трансформационному синтезу.

В общем случае задача автоматического синтеза программ алгоритмически неразрешима, то есть не существует алгоритма, который бы по произвольной спецификации строил соответствующую программу, однако известно множество частных, но практически важных случаев, в которых синтез программ возможен. Результаты, получаемые в ходе теоретических исследований по автоматическому синтезу программ находят практическое применение в оптимизирующих компиляторах, электронных таблицах и др. областях.

Автоматическое (или автоматизированное) проектирование и конструирование приложений по спецификациям дело трудное, но не безнадежное. Инструменты, поддерживающие UML, все время совершенствуются, так что в перспективе третье предназначение UML может выйти и на первое место.

**ЗАМЕЧАНИЕ**

> Некоторым уставшим от бесконечной отладки программистам может показаться, что стоит изучить UML и все проблемы программирования будут решены. К сожалению, пока это не так.

### *1.2.4. Документирование*

Модели UML являются артефактами, которые можно хранить и использовать как в форме электронных документов, и в виде твердой копии. В последних версиях UML с целью достижения более полного соответствия этому назначению сделано довольно много. В частности, специфицировано представление моделей UML в

форме документов в формате XMI[10], что обеспечивает практическую интероперабельность при работе с моделями. Другими словами, модели UML не являются вещью в себе, которой можно только любоваться — это документы, которые можно использовать самыми разными способами, начиная с печати картинок и заканчивая автоматической генерацией человекочитаемых текстовых описаний.

### *1.2.5. Чем НЕ является UML*

Не следует думать, что UML — это панацея от всех детских[11] болезней программирования. Для ясного понимания назначения и области применения UML полезно сопоставить UML с другими родственными явлениями.

Во-первых, UML не является языком программирования (хотя генерация кода не возбраняется, см. предыдущий раздел). Дело не в том, что UML язык графический, а подавляющее большинство практических языков программирования являются линейными текстовыми языками. Гораздо важнее то, что для моделей UML не определена *операционная семантика*, то есть не определен способ выполнения моделей на компьютере. Это сделано вполне сознательно, в противном случае UML оказался бы зависимым от некоторой модели вычислимости, уровень абстрактности его концепций пришлось бы существенно снизить и он не отвечал бы своему основному назначению: служить средством спецификации приложений и других систем на любом уровне абстракции и в различных предметных областях.

Во-вторых, UML не является спецификацией инструмента (хотя инструменты подразумеваются и имеются, например, Together, Rational Rose, Visual Paradigm, Microsoft Visio и др.). В последние годы в компьютерной индустрии широкое распространение получили комплексные системы разработки приложений, которые обычно называют CASE[12] средствами. В таких системах разработки предпринимаются попытки согласованным образом поддержать и обеспечить все фазы процесса разработки приложений, а не только фазы кодирования и отладки, традиционно поддерживаемые обычными системами программирования. Поскольку управление спецификациями и проектирование, по общему мнению, являются важнейшими составляющими процесса разработки приложений, понятно, что в CASE средствах должно быть предусмотрено нечто, эквивалентное основному назначению UML. Однако сам язык никоим образом не навязывает то, как его нужно поддерживать инструментальными средствами. Решение всех вопросов, связанных с реализацией UML на компьютере полностью отдано на откуп разработчикам инструментов.

**ЗАМЕЧАНИЕ**

> Все без исключения CASE средства, появившиеся на рынке в последние год-два так или иначе поддерживают UML.

---

[10] Специальное приложение XML.

[11] Информационным технологиям от силы полвека — это еще младенческий возраст для новой области человеческой деятельности.

[12] CASE — аббревиатура Computer Aided Software Engineering.

В третьих, UML не является моделью процесса разработки приложений (хотя модель процесса необходима и имеется множество различных. Конечно, у авторов UML есть собственная модель процесса — Rational Unified Process (RUP), которую они не могли не иметь в голове, разрабатывая язык, но, тем не менее, ими сделано все для того, чтобы устранить прямое влияние RUP на UML и сделать UML пригодным для использования в *любой* модели процесса или даже без оной.

**ЗАМЕЧАНИЕ**

> У автора этой книги тоже есть своя модель процесса, которая не может не сказываться на описании прагматики языка, но везде, где такое влияние замечено, сделаны соответствующие оговорки.

### *1.2.6. Способы использования UML*

Из сказанного выше видно, что UML предназначен для решения различных задач, соответственно он может быть использован и практически используется по-разному. Далее мы перечисляем различные способы использования UML в порядке убывания важности. Числа в скобках — это наша, может быть весьма субъективная оценка того, насколько практически важен и востребован данный способ в текущей ситуации.

- **Рисование картинок (+3)**. Графические средства UML можно и нужно использовать безотносительно ко всему остальному. Даже рисование диаграмм карандашом на бумаге позволяет упорядочить мысли и зафиксировать для себя существенную информацию о моделируемом приложении или иной системе. Мы ставим такое использование UML на первое место по важности.
- **Обмен информацией (+2)**. Сообщество людей, применяющих и понимающих UML стремительно растет. Если вы будете использовать UML, то вас будут понимать другие и вы будете понимать других с полувзгляда.
- **Спецификация систем (+1)**. Это важнейший способ использования UML. В нашей оценочной шкале использование UML для спецификации не стоит на первом месте только потому, что, к сожалению, еще не во всех случаях UML оказывается абсолютно адекватным средством спецификации (примеры приведены в других местах книги). Но по мере развития языка все меньше будет оставаться случаев, в которых UML оставляет желать лучшего.
- **Повторное использование архитектурных решений (0)**. Повторное использование ранее разработанных решений — ключ к повышению эффективности. Наше мнение по этому поводу изложено в главе 5. Однако, пока что модели UML повторно используются еще в ограниченных масштабах. Оценка 0 (золотая середина) означает: хорошо, но мало.
- **Генерация кода (–1)**. В разделе 1.2.3 приведена достаточно подробная аргументация, объясняющая оценку –1. Генерировать код нужно и можно, но возможности имеющихся инструментов не стоит переоценивать.
- **Имитационное моделирование (–2)**. Возможности построения моделей UML, из которых путем вычислительных экспериментов можно было бы извлекать информацию о моделируемом объекте, пока что уступают

возможностям специализированных систем, сконструированных для этой цели.

- **Верификация моделей (–3)**. Было бы замечательно, если бы по модели можно было бы делать формальные заключения об ее свойствах: модель непротиворечива, согласована, эффективна и т. п. Кое-что UML позволяет проверить, но, конечно, очень мало. Здесь уместно привести аналогию с традиционными системами программирования: они позволяют быстро и надежно избавиться от синтаксических ошибок, но с логическими ошибками дело обстоит гораздо хуже. Может быть, в будущем...

## *1.2.7. Инструментальная поддержка*

Рассмотрим, как соотносится сегодняшняя практика использования UML с декларированным выше назначением языка. По мнению автора, можно выделить три основных варианта использования UML (рис. 1.5).


**Рис. 1.5. Инструментальная поддержка.**

Вариант использования "Рисование диаграмм" подразумевает изображение диаграмм UML с целью обдумывания, обмена идеями между людьми, документирования и тому подобного. Значимым для пользователя результатом в этом случае является само изображение диаграмм. Вообще говоря, в этом варианте использования языка поддерживающий инструмент не очень нужен. Иногда рисование диаграмм от руки фломастером с последующим фотографированием цифровым аппаратом может оказаться практичнее.

Вариант использования "Составление моделей" подразумевает создание и изменение модели системы в терминах тех элементов моделирования, которые предусматриваются метамоделью UML. Значимым результатом в этом случае является машинно-читаемый артефакт с описанием модели. Мы будем для краткости называть такой артефакт просто моделью, деятельность по составлению модели называть моделированием, а субъекта моделирования называть архитектором (потому что соответствующие существительные — модельщик, модельер — в русском языке уже заняты).

Вариант использования "Разработка приложений" подразумевает детальное моделирование, проектирование, реализацию и тестирование приложения в терминах UML. Значимым для пользователя результатом в этом случае является работающее приложение, которое может быть скомпилировано во входной язык конкретной системой программирования или сразу интерпретировано средой выполнения инструмента. Этот вариант использования наиболее сложен в реализации.

Однако современные инструменты поддерживают указанные варианты использования далеко не в равной степени. Все инструменты умеют (плохо или хорошо) визуализировать все типы диаграмм UML, некоторые инструменты позволяют построить модель, допускающую какое-то дальнейшее использование, но только немногие инструменты могут генерировать исполнимый код и отнюдь не для всех диаграмм. Имеется множество практических и организационных причин, по которым указанные выше варианты использования неравноправны и в разной степени поддержаны в современных инструментах.

## 1.3. Определение UML

Искусственный язык должен быть как-то описан. Искусственный язык, претендующий на массовое использование, должен быть описан так, чтобы притягивать, а не отпугивать потенциальных пользователей. А это сделать не так просто.

### *1.3.1. Определения искусственных языков*

История языков программирования насчитывает полвека и столько же продолжается развитие методов их определения. На этом пути были прорывы (определения Алгола 60, Алгола 68, Паскаля), были и неудачи (PL/1). От определения искусственного языка требуется, чтобы оно было:

- точным,
- понятным,
- кратким,
- полным.

Между тем, эти требования зачастую исключают друг другу.[13] В некоторых случаях во главу угла ставится что-либо одно, но результат редко оканчивается удачей. Например, определение Алгола 68, где все было нацелено на формальную точность, явилось новым словом в методах определения языков и оказало большое влияние на развитие теории. В тоже время, это описание оказалось настолько непонятным рядовым пользователям, что отпугнуло их от потенциально многообещающей разработки. Другой пример: с целью "понятности" первые диалекты Кобола описывались совершенно неформально, в результате получалось плохо, потому что для достижения минимальной необходимой точности и полноты неформальное описание приходилось делать очень длинным, а потому непонятным.

Наиболее удачными оказались компромиссные решения, и авторы UML пошли именно по этому пути.

---

[13] Автору однажды встретилось замечательное рекламное объявление: "мы обучаем иностранному языку быстро, качественно и недорого — любые два условия из трех по выбору заказчика".

### *1.3.2. Метод определения UML*

В основу описания UML положен метод раскрутки, то есть использование определяемого языка для определения этого языка. А именно, основные конструкции UML формально определены с помощью UML. Конечно, с чего-то раскрутку нужно начать, и это описано в UML неформально, с помощью текстов на естественном (английском) языке.

**ЗАМЕЧАНИЕ**

> Метод раскрутки часто применяется при определении формальных языков. Например, можно очень изящно определить операционную семантику языка программирования, если написать транслятор или интерпретатор данного языка на этом же языке. Одним из первых этот прием использовал Н. Вирт, создавая язык Паскаль.

В описании UML используются три языковых уровня.

- Мета-метамодель, то есть описание языка, на котором описана метамодель.
- Метамодель, то есть описание языка, на котором описываются модели.
- Модель, то есть описание самой моделируемой предметной области.

Это кажется нагромождением сущностей без нужды, но на самом деле только использование ученой приставки мета (от греческого μετα, что означает "между", "после", "через") может быть несколько непривычным. Действительно, рассмотрим описание какого-либо из обычных языков программирования. Чтобы никого не обидеть, пусть этот язык называется X. Если описание хорошее, то в самом начале указывается язык (иногда его так и называют — метаязык), который используется для описания языка X. Например, приводится фраза такого типа: "синтаксис языка X описан с помощью контекстно-свободной грамматики, правила которой записаны в форме Бэкуса-Наура (БНФ), контекстные условия и семантика описаны на естественном языке". Если описание не очень хорошее, то такая фраза может и отсутствовать, но она все равно неявно подразумевается и от читателя требуется понять используемый метаязык по контексту. Далее с помощью метаязыка более или менее формально описываются конструкции языка X; все, что не удается описать формально, описывается на естественном языке (при этом зачастую вводится большое количество "новых" терминов и используются трудные для чтения канцеляризмы). Все это, как правило, сопровождается многочисленными конкретными примерами фрагментов текстов на языке X. Чтобы читатель не путался, где текст на языке X, а где текст на метаязыке, или где общее описание, а где конкретный пример, применяются различные полиграфические приемы: изменение гарнитуры и начертания шрифта, отступы, подчеркивание и т. д. В результате получается неплохо: многоязыковая смесь становится вполне удобочитаемой (при минимальном навыке).

#### Формальные грамматики

Для описания своих языков программисты с успехом используют теорию формальных грамматик, разработанную Н. Хомским. Вкратце суть заключается в следующем. Рассматривается конечный алфавит *A* (множество символов) и язык над этим алфавитом. Здесь язык — это просто множество последовательностей символов алфавита (цепочек). Другими словами, среди всех возможных цепочек (которых заведомо

бесконечно много) нужно иметь возможность выделить те и только те, которые являются цепочками языка (еще говорят "входят в язык", "принадлежат языку", "являются выражениями языка").

Для описания (бесконечного) множества цепочек Хомский предложил использовать *формальную грамматику* — конечное множество *правил* вида $\alpha \rightarrow \beta$, где $\alpha$ и $\beta$ конечные цепочки, составленные из символов алфавита *A* (которые называются *терминальными* символами) и из символов некоторого дополнительного алфавита *N* (эти символы принято называть *нетерминальными*). В левую часть правила (т. е. в цепочку $\alpha$) обязательно должен входить хотя бы один нетерминальный символ. Среди всех нетерминальных символов выделяется один, который называется начальным символом (или *аксиомой*) грамматики.

Далее все очень просто. Взяв за основу аксиому грамматики, разрешается строить новые цепочки, заменяя в уже построенной цепочке любое вхождение левой части любого правила на правую часть этого правила. И так до тех пор, пока в цепочке не останутся одни терминальные символы. Всякая цепочка терминальных символов, которую можно построить таким процессом, по определению входит в язык, а если цепочку нельзя так построить, то она не входит в язык. Например, грамматика, которая имеет один нетерминальный символ S, два терминальных символа 0 и 1 и два правила $S \rightarrow 01$ и $S \rightarrow 0S1$ определяет (как говорят *порождает*) язык, состоящий из цепочек вида 01, 0011, 000111 и так далее.

Разумеется, все это так просто, пока не поставлены разные каверзные вопросы. Всякий ли язык может быть определен таким способом? Если дана грамматика и конкретная терминальная цепочка, то как узнать, принадлежит она языку или нет? Однозначен ли процесс порождения цепочки? Зависят ли ответы на предыдущие вопросы от вида (формы) правил?

Но если основная идея ясна, то можно оставить эти (трудные) вопросы математикам и принять тот факт, что синтаксис подавляющего большинства языков программирования легко описывается[14] подклассом формальных грамматик, которые называются *контекстно-свободными* (левая часть всех правил состоит из одного нетерминального символа). Для этого подкласса грамматик все вопросы связанные с распознаванием принадлежности цепочки языку (это называется *синтаксическим анализом*, или *разбором*) надежно решены.

Таким образом, основная идея описания UML вполне традиционна и согласуется с общепринятой практикой: мета-метамодель — это описание используемого формализма; метамодель — это и есть собственно описание языка (элементов моделирования); там, где формализм не срабатывает, на помощь приходит естественный язык и все это сопровождается примерами фрагментов моделей.

[14] А если не описывается, или описывается нелегко, то всегда можно привлечь естественный язык и все, что нужно, досказать словами. В качестве примера возьмите любое описание популярного языка программирования: C++, Visual Basic, Java.

### *1.3.3. Структура стандарта UML*

Весь текст описания UML каждой версии находится в свободно распространяемых документах, доступных по адресу http://www.omg.org. В таблице 1.2. перечислены основные документы входящие в комплект документации для текущей версии UML 2.1.1 (октябрь 2007).

**Таблица 1.2.** Структура описания UML

| Часть описания | Основное содержание | Число страниц |
|---|---|---|
| UML Infrastructure | Описание базовых механизмов | 218 |
| UML Superstructure | Описание самого языка | 732 |
| UML Diagram interchange | Отображение средств обмена диаграммами | 83 |
| OCL Specifications | Описание объектного языка ограничений OCL | 232 |
| Всего страниц | | 1265 |

Без специальной предварительной подготовки читать в этих документах имеет смысл только вводные разделы, примерно соответствующие по задачам этой главе, и последние разделы, в которых собраны толкования основных терминов. Остальное предназначено не для ознакомительного чтения пользователями, а для скрупулезного изучения разработчиками инструментов моделирования. По замыслу авторов языка для пользователей должны быть написаны другие книги, и они написаны, как самими авторами, так и любителями UML. Один из примеров у вас в руках.

Самым важным является раздел описания семантики. Семантика описана следующим образом. Для определяемого элемента моделирования задается абстрактный синтаксис (в форме диаграммы классов UML) и указываются ограничения, которым должен удовлетворять описываемый элемент в форме выражений языка OCL. Все остальное, включая прагматику, описывается на естественном языке, который авторы называют "plain English".

Далее для кажого элемента моделирования, указывается как уже определенные элементы моделирования рекомендуется изображать в графическом виде. Фактически, этот раздел соответствует разделу описания синтаксиса для обычного языка программирования. Но UML не обычный язык, а графический! Пока что для описания синтаксиса графических языков не придумано столь же удобного механизма, как формальные грамматики, поэтому в этом разделе стандарта приведены довольно многословные описания в следующем стиле: "Состояние автомата изображается прямоугольником со скругленными углами, внутри которого указывается имя состояния в виде строки текста, а также, возможно…" и так далее. К счастью, графический синтаксис UML довольно прост и тщательно продуман, так что, если вы отличаете окружность от квадрата, то разобраться не трудно.

Большая часть содержания этой книги посвящена неформальному пересказу и обсуждению того, что формально описано в основном документе UML Superstructure. Прочие же разделы стандарта здесь фактически не рассматриваются, и не потому, что они длинные, а потому, что знание их содержания практически не влияет на достижение главной цели книги: объяснить, как, когда и почему можно и нужно использовать UML при разработке программного обеспечения.

**ЗАМЕЧАНИЕ**

> Как уже было сказано, читать стандарт для первого знакомства бесполезно. Но если вы уже знаете UML, то изучение того, как именно авторы используют UML для описания UML является чрезвычайно полезным упражнением.

### *1.3.4. Терминология и нотация*

Проблема терминологии является одной из самых болезненных в этой книге.

Во-первых, авторы UML старались сделать язык независимым от конкретных языков программирования, моделей вычислимости и тому подобного. С этой целью они зачастую вводили новые термины для определяемых понятий, чтобы случайное совпадение со старым термином, уже занятым в какой-либо смежной области программирования, не вводило в заблуждение пользователя. Кроме того, в UML унифицированы многие различные подходы и методы, каждый со своей терминологической традицией и их все необходимо было учесть и иногда пойти навстречу. Как всякий компромиссный вариант, терминология UML получалась довольно замысловатой и не всегда последовательной.

Во-вторых, эта книга написана по-русски и для русскоязычного читателя. Стало быть, термины UML должны так или иначе быть переданы кириллическими буквосочетаниями. UML сравнительно молод (но уже моден), бум публикаций в отечественной литературе начался только сейчас (с опозданием на десять лет, как обычно), поэтому устоявшейся терминологической традиции пока нет. Хотелось бы использовать хорошие русские слова — "эктор" и "персистентный" решительно отметаются. Словарь в таких случаях бесполезен (хотя им многие пытаются пользоваться). Остается фантазировать, опираясь на вышедшие из печати труды коллег, отечественные программистские традиции и собственный опыт. Основной критерий, который использован нами при выборе переводов терминов: как можно точнее передать *смысл* исходного термина (но не звучание, не морфологию и не буквальное значение).

Чтобы подчеркнуть, что UML язык графический, авторы называют правила записи (рисования) моделей не синтаксисом, а нотацией. Видимо, это вполне оправдано.

При разработке UML были предложены и приняты разумные рекомендации по выбору нотации. Авторы исходили из того, что UML будет использоваться по разному: начиная от не очень аккуратного рисования от руки на листке бумаги, печати черно-белых изображений в книгах и заканчивая созданием сложных диаграмм с помощью компьютера. Поэтому в качестве основных графических элементов были выбраны такие, которые было бы легко использовать во всех случаях. Типов элементов нотации четыре:

- фигуры;
- линии;
- значки;
- тексты.

*Фигуры* в UML используются двумерные (т. е. их можно нарисовать на плоскости) и замкнутые (т. е. есть внутренняя и внешняя части). Фигуры могут менять свои размеры и форму, сохраняя при этом свои интуитивные отличительные признаки. Например, среди фигур UML есть прямоугольники и эллипсы. Они могут быть изображены многими способами: разного размера, с разным соотношением длин сторон (или, соответственно, полуосей), по разному ориентированы относительно границ страницы и т. д., но всех случаях прямоугольник отличен от эллипса и не может быть с ним спутан. Внутри фигур могут помещаться другие элементы нотации: тексты, линии, значки и даже другие фигуры. Единственное требование: должно быть однозначно понятно, что элемент нотации

находится внутри фигуры, в частности, его изображение не должно пересекать границу фигуры.

*Линии* в UML, естественно, одномерные. Линии всегда присоединяются своими концами к фигурам или значкам, они не могут быть нарисованы сами по себе. Что значит "присоединяются" — формально определить довольно трудно, но неформально (а нотация UML *не* формальна) все совершенно ясно: линия должна быть нарисована так, чтобы любому нормальному человеку было ясно, присоединяется данная линия к данной фигуре, или нет. Форма линий произвольна: это могут быть прямые, ломаные, плавные кривые — значения это не имеет. Толщина линий также произвольна. А вот *стиль линии* (т. е. то, как линия нарисована) имеет значение. К счастью, в UML используется только два стиля линий, которые трудно спутать: сплошные и пунктирные линии. К линиям могут быть пририсованы различные дополнительные элементы: стрелки на концах тексты и т. д. Единственное требование: должно быть ясно, что дополнительный элемент относится именно к данной линии. Линии могут пересекаться, и это ничего не значит, но рекомендуется избегать таких случаев, поскольку это затрудняет восприятие.

*Значки* в UML похожи на фигуры тем, что они двумерные, а отличаются тем, что не имеют внутренности, в которую можно что-то поместить, и, как правило, не меняют свою форму и размеры. Впрочем, значки в UML используются очень умеренно (не так, как в графическом интерфейсе Windows), а потому сохраняют свою основную функцию однозначно воспринимаемого иероглифа.

*Тексты* в UML — это, как обычно, последовательности различимых символов некоторого алфавита. Алфавит не фиксирован — он только должен быть понятен читателю модели. Гарнитура, размер и цвет шрифта не имеют значения, а вот начертание шрифта имеет: в UML различаются прямые, *курсивные* и подчеркнутые тексты. Предполагается, что читатель сумеет их различить.

В общем, нотация UML довольно свободная: рисовать можно как угодно, лишь бы не возникало недоразумений. Поставщики инструментов, поддерживающих UML пользуются этой свободой кто во что горазд. Использование цветов для заливки фигур и раскрашивания линий, тени у значков и фигур, разные шрифты в текстах, наконец, анимация изображений — все это, конечно, полезные вещи, постольку, поскольку повышают наглядность картинок. Важно при этом знать меру, а мера очень проста: картинка должна оставаться вразумительной после печати на черно-белом принтере.

В книге имеется довольно много иллюстраций, выполненных в нотации UML. В качестве инструмента рисования использованы приложения Sun Java Studio Enterprise 8, Together, Visio 2000 Professional. Каждый из них имеет свои особенности и может получать разнообразные оценки пользователями, однако, из сказанного ранее следует, что семантика первична, а картинки вторичны, поэтому разработчики инструментов вольны в своем творчестве…

## 1.4. Модель и ее элементы

Модель UML — это конечное множество сущностей и отношений между ними. Сущности и отношения модели — это экземпляры метаклассов метамодели.

Обсуждение этого определения требует использования общеизвестных математических понятий, которые мы предполоагаем известными.
Таким образом, рассматривая модель UML с наиболее общих позиций, можно сказать, что это граф (точнее, нагруженный мульти-псевдо-гипер-орграф), в котором вершины и ребра нагружены дополнительной информацией и могут иметь сложную внутреннюю структуру. Вершины этого графа называются сущностями, а ребра — отношениями. Остальная часть параграфа содержит беглый (предварительный), но полный обзор имеющихся типов сущностей и отношений. К счастью, их не слишком много. В последующих главах книги все сущности и отношения рассматриваются еще раз, более детально и с примерами.

### *1.4.1. Сущности*

Для удобства обзора сущности в UML можно подразделить на четыре группы:

- структурные;
- поведенческие;
- группирующие;
- аннотационные.

**О классификациях и группировках**

Хорошо известна всемирная константа 7±2 (по другим источникам 5±2) — усредненная верхняя оценка количества различных объектов, понятий, шагов рассуждения, которые средний нормальный человек может полноценно воспринимать (держать в голове) одновременно. Нехорошо, если у начальника больше 7 подчиненных, которыми он должен одновременно командовать. Трудно работать больше чем с пятью приложениями одновременно. Не следует помещать на диаграмму больше девяти сущностей.

А что делать, если нужно? Ответ стар как мир: разделяй и властвуй. Другими словами, нужно ввести классификацию или группировку так, чтобы групп оказалось немного и работать на уровне групп. Если одного уровня группировки мало, следует ввести два, три — столько, сколько нужно чтобы уложиться в ограничение 7±2.

Следует подчеркнуть, что классификации и группировки сущностей, отношений, диаграмм и представлений, рассматриваемые в этом параграфе, не являются в строгом смысле частями спецификации UML, хотя и присутствуют в стандарте. Это скорее методический прием, призванный облегчить восприятие и изучение языка в соответствии с принципом разделяй и властвуй.

Структурные сущности, как нетрудно догадаться, предназначены для описания структуры. Обычно к структурным сущностям относят следующие.

- *Класс* — описание множества объектов с общими атрибутами и операциями.
- *Интерфейс* — множество операций, которое определяет набор услуг (службу), предоставляемых классом или компонентом.
- *Действующее лицо* — сущность, находящаяся вне моделируемой системы и непосредственно взаимодействующая с ней.

- *Вариант использования* — описание последовательности производимых системой действий, доставляющей значимый для некоторого действующего лица результат.
- *Компонент* — физически заменяемый артефакт, реализующий некоторый набор интерфейсов.
- *Узел* — физический вычислительный ресурс.

Приведенная классификация не является исчерпывающей. У каждой из этих сущностей есть различные частные случаи и вариации, рассматриваемые в последующих главах.

Поведенческие сущности предназначены для описания поведения. Основных поведенческих сущностей всего две: состояние и действие (точнее, две с половиной, потому что иногда употребляется еще и деятельность, которая является частным случаем состояния).

*Состояние* — период в жизненном цикле объекта, в котором объект удовлетворяет некоторому условию, выполняет деятельность или ожидает события.

*Деятельность* — состояние, в котором выполняется работа, а не просто пассивно ожидается наступление события.

*Действие* — атомарный неделимый элемент работы.

Это только надводная часть айсберга поведенческих сущностей: состояния бывают самые разные (см.). Кроме того, при моделировании поведения используется еще ряд вспомогательных сущностей, которые здесь не перечислены, потому что сосуществуют только вместе с указанными основными.

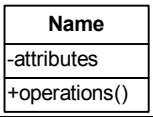
Группирующая сущность в UML одна — пакет — зато универсальная.

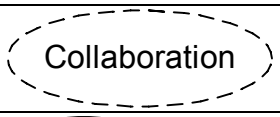
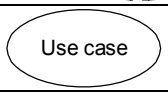
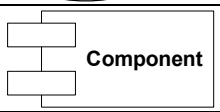
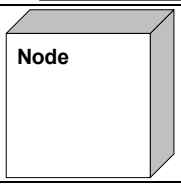
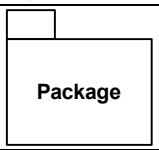
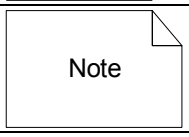
*Пакет* — группа элементов модели (в том числе пакетов).

Аннотационная сущность тоже одна — *примечание* — зато в нее можно поместить все что угодно, так как содержание примечания UML не ограничивает.

В таблице 1.3 приведена стандартная нотация в минимальном варианте для упомянутых сущностей.

**Таблица 1.3** Нотация основных сущностей

| | |
|---|---|
| Класс |  |
| Действующее лицо |  |
| Интерфейс |  |

| | |
|---|---|
| Кооперация |  |
| Вариант использования |  |
| Компонент |  |
| Узел |  |
| Состояние |  |
| Деятельность |  |
| Развилка / слияние |  |
| Ветвление / соединение |  |
| Пакет |  |
| Примечание |  |

**Ортогональные системы обозначений**

Очень хорошо, если система обозначений такова, что все ее элементы абсолютно необходимы, используются в равной мере и допускают единственный способ выражения данной мысли. Такие системы обозначений принято называть ортогональными, по аналогии с ортогональной системой векторов в векторном пространстве. Действительно, ортогональная система образует базис, т.е. любой вектор можно выразить в виде линейной комбинации векторов базиса, причем это выражение единственно. Это удобно.

К сожалению, распространенные языки программирования не являются ортогональными системами обозначений: как вы, несомненно, знаете, один и тот же алгоритм можно запрограммировать очень и очень по разному, некоторые элементы языка используются в каждой программе,

а про наличие других программист может и не подозревать. Прогресс в направлении ортогонализации языков программирования за последнее десятилетие очевиден, но идеал пока не достигнут.

UML также далек от идеала. Некоторые элементы абсолютно необходимы в любой модели, и их мы перечисляем в этом параграфе, но есть и такие, которые автору, вслед за Э. Дейкстрой, хочется отнести к "свистулькам и погремушкам".

### *1.4.2. Отношения*

В UML используются четыре основных типа отношений:

- зависимость;
- ассоциация;
- обобщение;
- реализация.

*Зависимость* — это наиболее общий тип отношения между двумя сущностями, Отношение зависимости указывает на то, что изменение независимой сущности каким-то образом влияет на зависимую сущность. Графически отношение зависимости изображается в виде пунктирной стрелки, направленной от независимой сущности к зависимой. Как правило, семантика конкретной зависимости уточняется в модели с помощью дополнительной информации. Например, зависимость со стереотипом «`use`» означает, что зависимая сущность использует (скажем, вызывает операцию) независимую сущность.

*Ассоциация* — это наиболее часто используемый тип отношения между сущностями. Отношение ассоциации имеет место, если одна сущность непосредственно связана с другой (или с другими — ассоциация может быть не только бинарной). Графически ассоциация изображается в виде сплошной линии с различными дополнениями, соединяющей связанные сущности. На программном уровне непосредственная связь может быть реализована различным образом, главное, что ассоциированные сущности знают друг о друге. Например, отношение часть–целое является частным случаем ассоциации и называется отношением агрегации.

*Обобщение* — это отношение между двумя сущностями, одна их которых является частным (специализированным) случаем другой. В UML отношение обобщения подразумевает выполнение принципа подстановочности: если сущность А (общее) суть обобщение сущности Б (частное), то Б может быть подставлено вместо А в любом контексте. Принцип подстановочности правомерен, поскольку обобщение подразумевает, что сущность Б обладает всеми свойствами и поведением сущности А и, возможно, еще чем-то. Графически обобщение изображается в виде сплошной стрелки с треугольником на конце, направленной от частного к общему. Отношение наследования между классами в объектно-ориентированных языках программирования является типичным примером обобщения.

Отношение *реализации* используется несколько реже, чем предыдущие три типа отношений, поскольку часто подразумеваются по умолчанию. Отношение реализации указывает, что одна сущность является реализацией другой. Например, класс является реализацией интерфейса. Графически реализация изображается в

виде пунктирной стрелки с треугольником на конце, направленной от реализующей сущности к реализуемой.
Перечисленные типы отношений являются основными, различные их вариации и дополнительные отношения детально рассматриваются в последующих главах.

### О важности правильных обозначений

Нельзя не отметить тщательность выбора графической нотации в UML. Если основные принципы поняты, то обозначения оказываются вполне естественными. Рассмотрим, например, нотацию отношений. Два типа линий подчеркивают степень определенности семантики, привносимой отношением в модель. Ассоциация и обобщение имеют четко определенную семантику, по которой можно прямо генерировать код, поэтому и линии сплошные. Зависимость и реализация более расплывчаты — линии используются пунктирные. Далее, направления стрелок тоже выбраны не случайным, а единственно правильным образом. Независимая сущность может и не знать, что ее использует другая сущность, а вот зависимая сущность не может не знать, от чего она зависит. Эта аналогия уместна и для обобщения и реализации. Например, описывая наследование (которое часто реализуется указателем в наследующем классе на наследуемый класс) в большинстве языков программирования имя наследуемого класса появляется в заголовке наследующего, и направление указателя становится явно видным. То же относится и к дополнительным элементам стрелок: обобщение и реализация имеют много общего, поэтому и наконечники стрелок у них одинаковые.

При рассмотрении последующих (более детальных и содержательных) примеров нотации читателю было бы полезно задавать себе вопрос: "почему выбрано такое обозначение?" до тех пор, пока сам собой не появится ответ: "а как же иначе!".

## 1.5. Диаграммы

Предыдущий параграф преследует примерно те же цели, какие имеет описание лексики в обычном языке программирования. А именно, мы обрисовали (еще не полностью, но уже достаточно) множество слов UML (лексем, графических примитивов — называйте как хотите — фиксированного термина нет). Пора переходить к синтаксису, а именно к описанию того, как из слов конструируются предложения.
На первый взгляд, все очень просто: берутся сущности и, если нужно, указываются отношения между ними. В результате получается модель, то есть граф (с разнородными вершинами и ребрами), нагруженный дополнительной информацией. Но при более внимательном рассмотрении обнаруживаются проблемы. Мы хотим затратить некоторые усилия на обсуждение этих проблем, иначе целый ряд особенностей UML может показаться висящим в воздухе, хотя на самом деле, эти особенности и есть продуманное решение замалчиваемых обычно проблем.
Рассмотрим следующую аналогию с естественным языком. Каждая тройка сущность — отношение — сущность в модели вполне может рассматриваться как простое утверждение: 2<5, ртуть тяжелее железа, Ф.А. Новиков является сотрудником Политехнического университета (все это примеры отношений). Пока

все хорошо, но вспомним, что в графе (в модели) никакой упорядочивающей структуры нет: нельзя сказать, что это вершина первая, а это — вторая. Продолжая нашу аналогию, получается, что модель — это множество несвязанных между собой предложений, никак не упорядоченное, некий "поток сознания", не в обиду современной литературе будь сказано. Если взять какой-нибудь "нормальный" текст,[15] претендующий на внятное описание чего-либо, то можно заметить, что помимо структуры предложений, имеется масса дополнительных структур: предложения объединены в абзацы, абзацы собраны в параграфы и главы, у которых есть заголовки, помимо обычных абзацев и заголовков есть врезки и сноски. И все эти дополнительные структуры по сути ничего не добавляют к содержанию книги, но серьезно влияют на ее читабельность. Текст, в котором нет этих структур, понять очень трудно.[16] Отсюда вывод: помимо сущностей и отношений, в модели должна быть какая-то структура, которая бы помогала ее составлению и пониманию.

Диаграммы UML и есть та основная накладываемая на модель структура, которая облегчает создание и использование модели.

| Диаграмма — это графическое представление некоторой части графа модели. |
|---|

Вообще говоря, в диаграмму можно было бы включить любые (допустимые) комбинации сущностей и отношений, но произвол в этом вопросе затруднил бы понимание моделей. Поэтому авторы UML определили набор рекомендуемых к использованию типов диаграмм, которые получили название *канонических* типов диаграмм.

**ЗАМЕЧАНИЕ**

> Инструменты моделирования, как правило, обеспечивают работу со всеми каноническими диаграммами, но делают это довольно догматически, не позволяя отойти от канона ни на шаг, даже если эту нужно по сути задачи. С другой стороны, некоторые инструменты (например, Together) наряду с каноническими поддерживают и апокрифические типы диаграмм. Было бы удобно, если бы набор канонических диаграмм предлагался по умолчанию, но пользователь мог бы настроить, изменить и переопределить этот набор в случае необходимости, примерно так, как это делается с шаблонами Word.

### *1.5.1. Классификация диаграмм*

В UML 1.x всего определено 9 канонических типов диаграмм. Ниже перечислены их названия, принятые в этой книги (в других источниках есть отличия).

- Диаграмма использования
- Диаграмма классов
- Диаграмма объектов
- Диаграмма состояний
- Диаграмма деятельности
- Диаграмма последовательности

---

[15] Скажем эту книгу — не в качестве образца, а в качестве примера.

[16] Смеем вас уверить, что писать структурированный текст также намного легче.

- Диаграмма кооперации
- Диаграмма компонентов
- Диаграмма размещения

Этот список является итогом многочисленных дискуссий и компромиссов, поэтому не следует воспринимать его как догму. В частности, расхожее утверждение "в UML определены 9 типов диаграмм" является не совсем верным: в метамодели UML определены элементы модели (сущности и отношения) и способы их комбинирования, а 9 типов диаграмм — это уже надстройка над языком, отражающая сложившуюся практику его использования.

Канонические диаграммы отнюдь не образуют полного ортогонального набора: они пересекаются как по включенным в них средствам, так и по области применения. Более того, некоторые из них являются частными случаями других, есть просто семантические эквивалентные пары, можно привести примеры допустимых диаграмм, для которых затруднительно указать однозначно, к какому именно из канонических типов диаграмма относится.

Сказанное можно проиллюстрировать условной классификацией диаграмм, приведенной на рис. 1.6.


**Рис. 1.6. Иерархия типов диаграмм.**

## ЗАМЕЧАНИЕ

> Мы поместили диаграмму использования отдельно, не относя ее ни к диаграммам описания структуры, ни к диаграммам описания поведения. В большинстве источников диаграммы использования относят к описанию поведения, что нам представляется некоторой натяжкой.

Далее мы по порядку, но очень бегло опишем все канонические типы диаграмм, с тем чтобы иметь определенный контекст и словарный запас для последующего изложения. Детали изложены в остальных главах книги.

## 1.5.2. Диаграмма использования

*Диаграмма использования* — это наиболее общее представление функционального назначения системы. Диаграмма использования призвана ответить на главный вопрос моделирования: что делает система во внешнем мире?

На диаграмме использования применяются два типа основных сущностей: варианты использования и действующие лица, между которыми устанавливаются следующие основные типы отношений:

- ассоциация между действующим лицом и вариантом использования;
- обобщение между действующими лицами;
- обобщение между вариантами использования;
- зависимости (различных типов) между вариантами использования.

Кроме того, на диаграмме использования можно применить специальный графический комментарий — обозначить границу системы (действующие лица находятся снаружи, а варианты использования — внутри), если это почему-либо не очевидно.

Основные элементы нотации, применяемые на диаграмме использования, показаны на рис. 1.7.

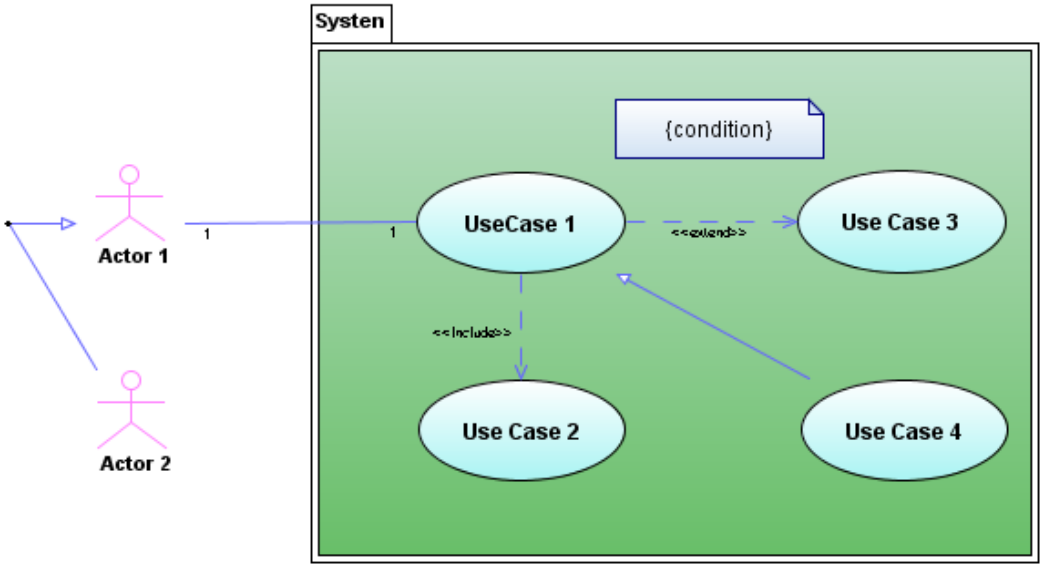



**Рис.1.7. Нотация диаграммы использования (UML 2.0)**

## 1.5.3. Диаграмма классов

*Диаграмма классов* — основной способ описания структуры системы. Это не удивительно, поскольку UML сильно объектно-ориентированный язык, и классы являются основным (если не единственным) "строительным материалом" системы.

На диаграмме классов применяются один основной тип сущностей: классы (включая многочисленные частные случаи классов: интерфейсы, типы, классы-ассоциации и многие другие), между которыми устанавливаются следующие основные типы отношений:

- ассоциация между классами (с множеством дополнительных подробностей);

- обобщение между классами;
- зависимости (различных типов) между классами и интерфейсами.

Основные элементы нотации, применяемые на диаграмме классов, показаны на рис. 1.8.

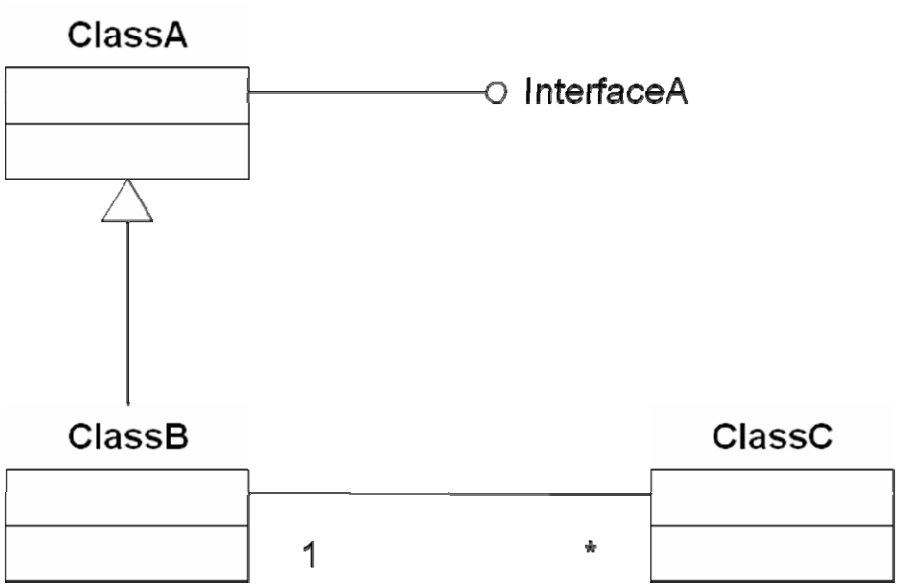



**Рис. 1.8. Нотация диаграммы классов**

### *1.5.4. Диаграмма объектов*

*Диаграмма объектов* — это частный случай диаграммы классов. Диаграммы объектов имеют вспомогательный характер — по сути это примеры (можно сказать дампы памяти), показывающие, какие имеются объекты и связи между ними в некоторый конкретный момент функционирования системы.

На диаграмме объектов применяют один основной тип сущностей: объекты (экземпляры классов), между которыми указываются конкретные связи (экземпляры ассоциаций).

Основные элементы нотации, применяемые на диаграмме объектов, показаны на рис. 1.9.

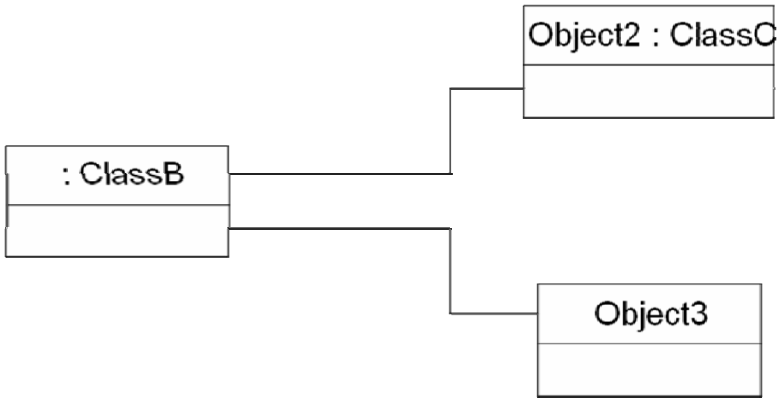



**Рис. 1.9. Нотация диаграммы объектов**

### *1.5.5. Диаграмма состояний*

*Диаграмма состояний* — это основной способ детального описания поведения в UML. В сущности, диаграммы состояний представляют собой граф состояний и переходов конечного автомата (см. главу 4), нагруженный множеством дополнительных деталей и подробностей.

На диаграмме состояний применяют один основной тип сущностей — состояния, и один тип отношений — переходы, но и для тех и для других определено множество разновидностей, специальных случаев и дополнительных обозначений. Перечислять их все во вступительном обзоре не имеет смысла.

Детальное описание всех вариаций диаграмм состояний приведено в главе 4, а на рис. 1.10 показаны только основные элементы нотации, применяемые на диаграмме состояний.

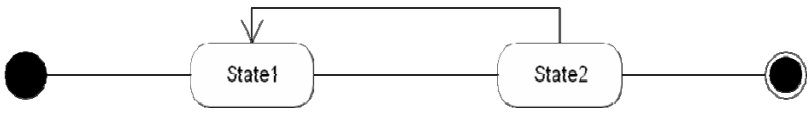



**Рис. 1.10. Нотация диаграммы состояний (UML 1.x)**

### *1.5.6. Диаграмма деятельности*

*Диаграмма деятельности* — это, фактически, старая добрая блок-схема алгоритма, в которой модернизированы обозначения, а семантика согласована с современным объектно-ориентированным подходом, что позволило органично включить диаграммы деятельности в UML.

На диаграмме деятельности применяют один основной тип сущностей — деятельность, и один тип отношений — переходы (передачи управления), а также графические обозначения (развилки, слияния и ветвления), которые похожи на сущности, но таковыми на самом деле не являются, а представляют собой графический способ изображения некоторых частных случаев гипердуг в гиперграфе (см. врезку "Множества, отношения и графы"). Семантика элементов диаграмм деятельности подробно разобрана в главе 4.

Помимо потока управления на диаграмме деятельности можно показать и поток данных, используя такую сущность, как объект (в определенном состоянии) и соответствующую зависимость. Кроме того, на диаграмме деятельности можно применить специальный графический комментарий — так называемые дорожки — подчеркивающие, что некоторые деятельности отличаются друг от друга, например, выполняются в разных местах. Основные элементы нотации, применяемые на диаграмме деятельности, показаны на рис. 1.11.

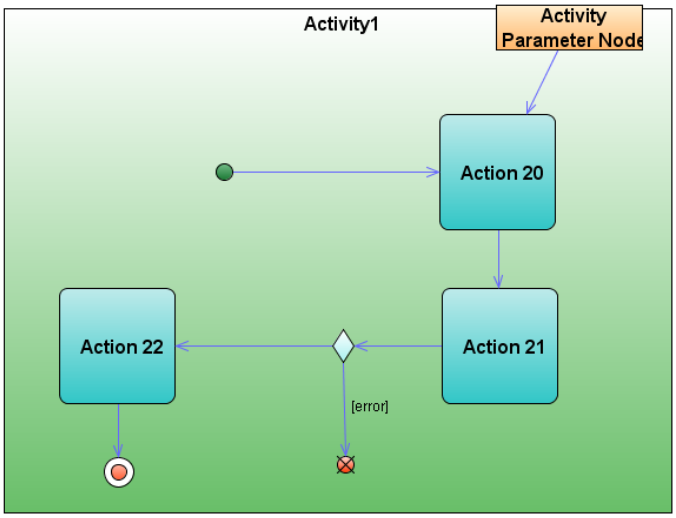



**Рис. 1.11. Нотация диаграммы деятельности (UML 2.0)**

## *1.5.7. Диаграмма последовательности*

*Диаграмма последовательности* — это способ описать поведение системы "на примерах". Фактически, диаграмма последовательности — это запись протокола конкретного сеанса работы системы (или фрагмента такого протокола). В объектно-ориентированном программировании самым существенным во время выполнения является посылка сообщений взаимодействующими объектами. Именно последовательность посылки сообщений отображается на данной диаграмме, отсюда и название.

На диаграмме последовательности применяют один основной тип сущностей — объекты (экземпляры взаимодействующих классов и действующих лиц), и один тип отношений — сообщения, которыми обмениваются взаимодействующие объекты. Предусмотрено несколько типов сообщений, которые в графической нотации различаются видом стрелки, соответствующей отношению.

Важным аспектом диаграммы последовательности является явное отображение течения времени. В отличии от всех других типов диаграмм, на диаграмме последовательности имеет значение не только наличие графических связей между элементами, но и взаимное положение элементов на диаграмме. А именно, считается, что имеется (невидимая) ось времени, по умолчанию направленная сверху вниз, и то сообщение, которое отправлено позже, нарисовано ниже.

**ЗАМЕЧАНИЕ**

> Ось времени может быть направлена горизонтально, в этом случае считается, что время течет слева направо.

На рис. 1.12 показаны основные элементы нотации, применяемые на диаграмме последовательности. Для обозначения самих взаимодействующих объектов применяется стандартная нотация — прямоугольник с подчеркнутым именем объекта. Пунктирная линия, выходящая из объекта, называется *линией жизни*. Это не обозначение отношение в модели, а графический комментарий, призванный направить взгляд читателя диаграммы в правильном направлении. Фигуры в виде узких полосок, наложенных на линию жизни, также не являются изображениями

моделируемых сущностей. Это графический комментарий, показывающий отрезки времени, в течении которых объект имеет управление (как говорят, имеет место активация объекта). Создание объекта в процессе взаимодействия отмечается тем, что значок объекта расположен ниже (т. е. объект появляется позже). Уничтожение объекта отмечает большим косым крестом и прекращением линии жизни. Прочие детали нотации диаграммы последовательностей см. в главе 4.

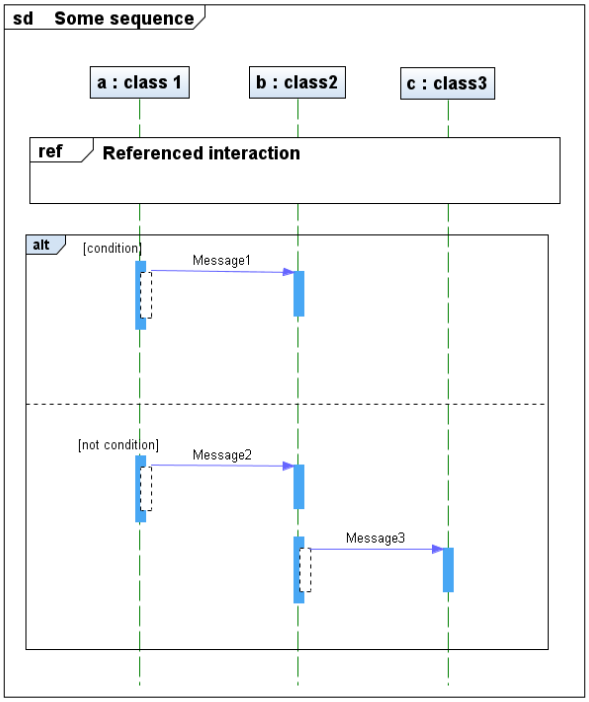



**Рис. 1.12. Нотация диаграммы последовательности (UML 2.0)**

## *1.5.8. Диаграмма кооперации*

*Диаграмма кооперации*, которая в UML 2 переименована в диаграмму коммуникации, семантически эквивалентна диаграмме последовательности. Фактически, это такое же описание последовательности обмена сообщениями взаимодействующих объектов, только выраженное другими графическими средствами. Более того, большинство инструментов умеет автоматически преобразовывать диаграммы последовательности в диаграммы кооперации и обратно. Такое преобразование является взаимно-однозначным.

Таким образом, на диаграмме кооперации также применяют один основной тип сущностей — объекты (экземпляры взаимодействующих классов и действующих лиц), и один тип отношений — сообщения, которыми обмениваются взаимодействующие объекты. Однако здесь акцент делается не на времени, а на связях между конкретными объектами.

На рис. 1.13 показаны основные элементы нотации, применяемые на диаграмме кооперации. Для обозначения самих взаимодействующих объектов применяется стандартная нотация — прямоугольник с подчеркнутым именем объекта. Взаимное положение объектов на диаграмме кооперации не имеет значения — важны только

связи (экземпляры ассоциаций), вдоль которых передаются сообщения. Для отображения упорядоченности сообщений во времени применяется иерархическая десятичная нумерация. Сравните рис. 1.12 и рис. 1.13 (на них изображена одна и та же диаграмма), и вам все станет понятно. Прочие детали нотации диаграммы кооперации см. в главе 4.

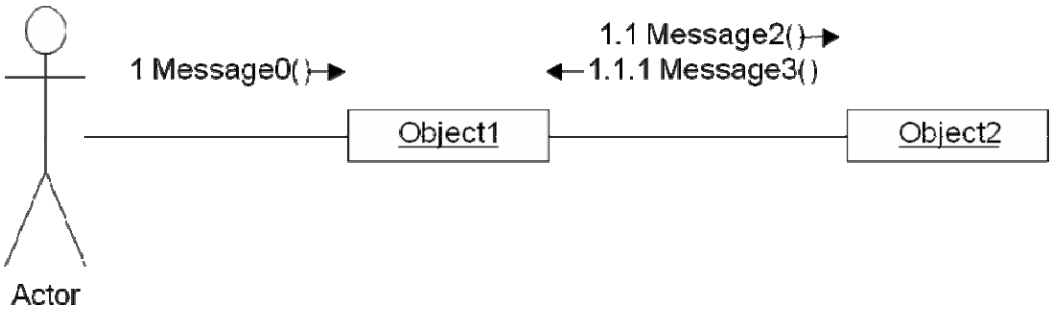



**Рис. 1.13. Нотация диаграммы кооперации**

### *1.5.9. Диаграмма компонентов*

*Диаграмма компонентов* — это, фактически, список артефактов, из которых состоит моделируемая система, с указанием некоторых отношений между артефактами. Наиболее существенным типом артефактов программных систем являются программы. Таким образом, на диаграмме компонентов основной тип сущностей — это компоненты (как исполнимые модули, так и другие артефакты), а также интерфейсы (чтобы указывать взаимосвязь между компонентами) и объекты (входящие в состав компонентов). На диаграмме компонентов применяются следующие отношения:

- реализации между компонентами и интерфейсами (компонент реализует интерфейс);
- зависимости между компонентами и интерфейсами (компонент использует интерфейс);
- зависимости между объектами и компонентами (объект входит в компонент).

На рис. 1.14 показаны основные элементы нотации, применяемые на диаграмме компонентов. Отношение зависимости, соответствующее включению (например, объекта в компонент), часто[17] изображают, помещая фигуру одной сущности внутрь фигуры другой сущности.

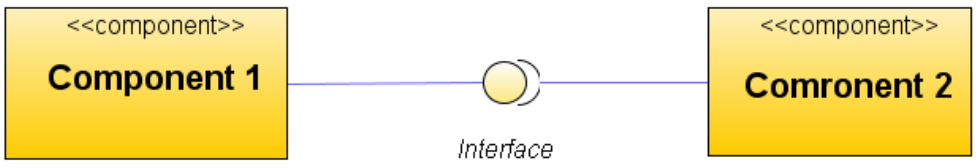



**Рис. 1.14. Нотация диаграммы компонентов (UML 2.0)**

[17] Если позволяют возможности используемого инструмента.

### *1.5.10. Диаграмма размещения*

*Диаграмма размещения* немногим отличается от диаграммы компонентов. Фактически, наряду с отображением состава и связей компонентов здесь показывается, как физически размещены компоненты на вычислительных ресурсах во время выполнения. Таким образом, на диаграмме размещения, по сравнению с диаграммой компонентов, добавляется один тип сущностей — узел (может быть как классификатор, описывающий тип узла, так и конкретный экземпляр), а также отношение ассоциации между узлами, показывающее, что узлы физически связаны во время выполнения.

На рис. 1.15 показаны основные элементы нотации, применяемые на диаграмме размещения. Прием с включением фигуры одной сущности внутрь фигуры другой сущности здесь применяется особенно часто.

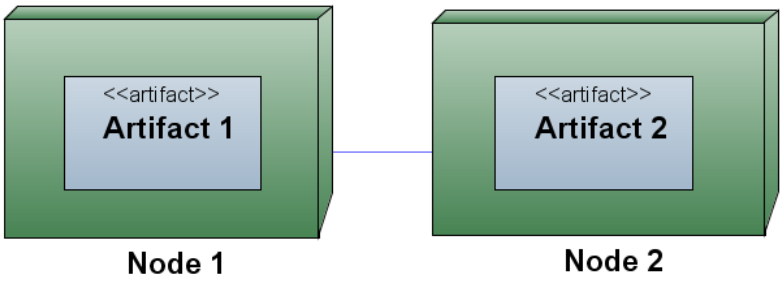



**Рис. 1.15. Нотация диаграммы размещения**

## 1.6. Представления

Было бы очень соблазнительно иметь возможность строить модели любых систем единообразно, придерживаясь, так сказать, одной универсальной точки зрения. Во многих ранних методологиях моделирования и проектирования программных систем такие попытки (более или менее удачные) предпринимались.

Как показывает практический опыт, не удается описать с единой точки зрения все без исключения аспекты моделируемой системы. Действительно, в модели нужно отразить множество вещей: интерфейсы для взаимодействия с внешним миром, внутреннюю логическую структуру программы, структуру хранимых данных, алгоритмы функционирования, состав артефактов, включаемых в поставку и многое другое. Было бы самонадеянно утверждать, что единое средство описания всех аспектов сразу в принципе невозможно — просто пока мы не знаем такого средства. Отсюда следует вывод: моделировать сложную систему следует с нескольких различных точек зрения, каждый раз принимая во внимание один аспект моделируемой системы и абстрагируясь от остальных. Этот тезис является одним из основополагающих принципов UML, по нашему мнению, может быть самым важным принципом, предопределившим практический успех UML.

Идея состоит в том, что абстрактный граф модели, состоящий из множества разнотипных сущностей и отношений, не подлежит конструированию или изучению в целом. Каждый раз для визуализации, изменения или иных манипуляций из этого общего графа вычленяются только сущности и отношения, релевантные для определенного аспекта моделируемой системы, а все остальные

игнорируются. Такой вид с определенной точки зрения, можно сказать проекцию модели, мы называем *представлением*.

### *1.6.1. Пять представлений*

Набор используемых представлений модели является еще менее формальным и догматическим, чем набор канонических типов диаграмм. Например, одним из самых популярных является набор представлений, описанных авторами UML в [1] и показанных на рис. 1.16.[18]

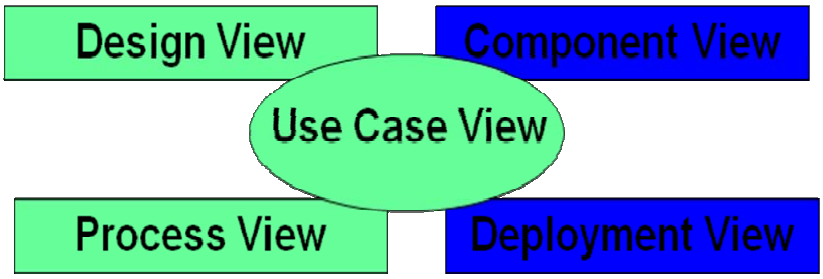



**Рис. 1.16. Пять представлений модели**

*Представление использования* — это описание поведения системы с точки зрения внешних по отношению к ней агентов. Структурные аспекты передаются диаграммами использования, а поведенческие аспекты — диаграммами взаимодействия, состояний и деятельности.

*Логическое представление* предназначено для описания словаря предметной области, то есть, в парадигме объектно-ориентированного программирования, классов. Структурные аспекты передаются диаграммами классов и объектов, а поведенческие аспекты — диаграммами взаимодействия, состояний и деятельности.

*Представление процессов* — это описание взаимодействия процессов в во время работы системы. Оно отражает такие аспекты, как параллелизм, синхронизация, производительность, масштабируемость, пропускная способность. Структурные аспекты передаются с помощью концепции активных классов: процессы и потоки, а поведенческие аспекты — диаграммами взаимодействия, состояний и деятельности.

*Представление компонентов* — это описание конфигурации системы на уровне артефактов. Структурные аспекты передаются диаграммами компонентов, а поведенческие аспекты — диаграммами взаимодействия, состояний и деятельности.

*Представление размещения* отражает топологию связей аппаратных средств и размещения на них компонентов. Структурные аспекты передаются диаграммами размещения, а поведенческие аспекты — диаграммами взаимодействия, состояний и деятельности.

---

[18] Этот рисунок заимствован из [1] с соответствующим изменением терминологии.

### 1.6.2. Восемь представлений

С другой стороны, те же авторы в книге [2],[3] рассматривают уже восемь представлений, как показано в табл. 1.4.[19]

**Таблица 1.4.** Представления модели и диаграммы в языке UML

| Представления | Диаграммы | Основные концепции |
|---|---|---|
| Статическое представление | Диаграмма классов | Класс, ассоциация, обобщение, зависимость, реализация, интерфейс |
| Представление использования | Диаграмма использования | Вариант использования, действующее лицо, ассоциация, расширение, включение, обобщение вариантов использования |
| Представление реализации | Диаграмма компонентов | Компонент, интерфейс, зависимость, реализация |
| Представление размещения | Диаграмма размещения | Узел, компонент, зависимость, расположение |
| Представление конечных автоматов | Диаграмма состояний | Состояние, событие, переход, действие |
| Представление деятельности | Диаграмма деятельности | Состояние, деятельность, переход по завершении, развилка, слияние |
| Представление взаимодействия | Диаграмма последовательности | Взаимодействие, объект, сообщение, активация |
| | Диаграмма кооперации | Кооперация, взаимодействие, роль в кооперации, сообщение |
| Представление управления моделью | Диаграмма классов | Пакет, подсистема, модель |

Нельзя не заметить, что здесь набор представлений не многим отличается от набора канонических диаграмм, если не считать управления моделями. Включение этого аспекта в число представлений нам представляется спорным.

### 1.6.3. Три представления

Учитывая неформальный характер понятия представления и опираясь на собственный опыт использования UML, мы рискнем предложить свой вариант набора представлений. Их всего три.

*Представление использования*. По сути это то же самое представление, что было указано выше. Представление использования призвано отвечать на вопрос, *что* делает система полезного. Определяющим признаком для отнесения элементов модели к представлению использования является, по нашему мнению, явное сосредоточение внимание на факте наличия у системы внешних границ, то есть выделение внешних действующих лиц, взаимодействующих с системой, и

[19] Эта таблица заимствована из [2],[3] с соответствующим изменением терминологии.

внутренних вариантов использования, описывающих различные сценарии такого взаимодействия. Таким образом, единственным выразительным средством представления использования оказываются диаграммы использования.

*Представление структуры*. Представление структуры призвано отвечать (с разной степенью подробности) на вопрос: *из чего* состоит система. Определяющим признаком для отнесения элементов модели к представлению структуры является явное выделение структурных элементов — составных частей системы — и описания взаимосвязей между ними. Принципиальным является чисто статический характер описания, то есть отсутствие понятия времени в любой форме, в частности, в форме последовательности событий и/или действий. Представление структуры описывается прежде всего и главным образом диаграммами классов, а также, если нужно, диаграммами компонентов и размещения и, в редких случаях, диаграммами объектов.

*Представление поведения*. Представление поведения призвано отвечать на вопрос: *как* работает система. Определяющим признаком для отнесения элементов модели к представлению поведения является явное использования понятия времени, в частности, в форме описания последовательности событий/действий, то есть в форме алгоритма. Представление поведения описывается диаграммами состояний и деятельности, а также диаграммами взаимодействия в форме диаграмм кооперации и/или последовательности.

Такой набор представлений является (почти[20]) ортогональным и согласованным с классификацией диаграмм (см. рис. 2.3). Более того, он во многом инспирирован личным опытом моделирования. Автору никогда не удавалось построить разумную модель для мало-мальски сложной системы так, как это рекомендуется в некоторых учебниках: сначала построить представление использования, затем последовательно логическое представление, представления процессов и компонентов и, наконец, представление развертывания.

Если процесс моделирования заканчивался удовлетворительным результатом, то он (процесс) оказывается итеративным и параллельным, примерно таким как показано на рис. 2.17.

[20] Немного выпадают из общей схемы активные классы, которые хочется отнести к представлению поведения.

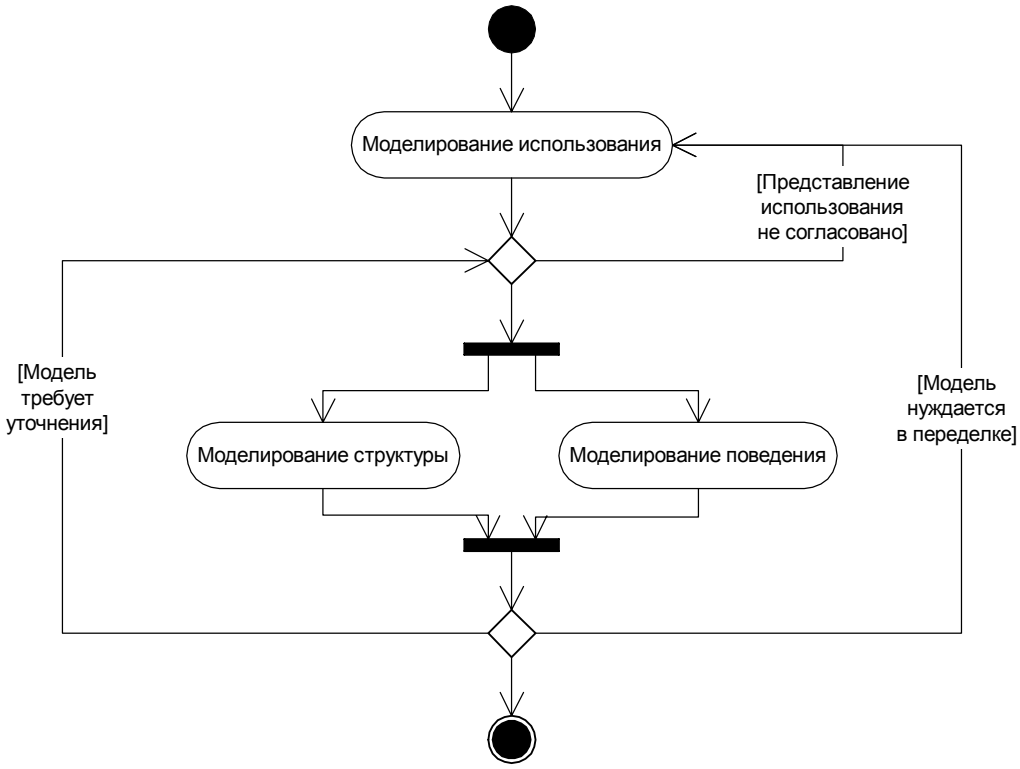



**Рис. 1.17. Процесс моделирования**

Другими словами, процесс итеративный, на каждом шаге может присутствовать уточнение представления использования, за которым следует параллельное моделирование структуры и поведения.

Исходя из сказанного, мы положили свой набор представлений в основу структуры книги.

## 1.7. Общие механизмы

В UML имеются общие правила и механизмы, которые относятся не к конкретным элементам модели, а ко всему языку в целом. Обычно выделяют следующие общие механизмы:

- внутреннее представление модели;
- дополнения;
- подразделения;
- механизмы расширения.

Степень значимости и сложности этих механизмов, равно как и их влияние на применение языка отнюдь неодинакова. Мы начнем с элементарных и закончим более сложными.

### *1.7.1. Внутреннее представление модели*

Модель имеет *внутреннее представление* — как бы иначе работали инструменты? Для графов (а модель — это нагруженный граф) известно множество разнообразных способов их представления в компьютере.

#### Представление графов в компьютере

Известно много способов представления графов в памяти компьютера, различающихся объемом занимаемой памяти и скоростью выполнения операций над графами. Представление выбирается исходя из потребностей конкретной задачи. Перечислим три наиболее характерных способа представления графа с $p$ вершинами и $q$ ребрами.

*Матрица смежности* — булевская квадратная матрица $M$ размера $p \times p$, в которой элемент $M[i,j]=1$, если вершины $i$ и $j$ смежны. Объем занимаемой памяти — $O(p^2)$. *Списки смежности* — структура данных, построенная на указателях, где для каждой вершины хранится связный список смежных вершин. Объем занимаемой памяти — $O(p+4q)$. *Список ребер* — массив длины $q$, хранящий пары смежных вершин. Объем занимаемой памяти — $O(2q)$.

На практике используется, как правило, некоторая комбинация указанных представлений с добавлением структур для хранения информации, нагруженной на вершины и ребра. Очевидные модификации позволяют использовать эти представление для мульти, псевдо и орграфов. Следует подчеркнуть, что нельзя указать представление графа, которое было бы наилучшим во всех возможных случаях. В разных ситуациях оказывается выгодным использовать различные представления.

Разработчики инструментов для моделирования на UML вправе использовать любое представление или придумать свое (что обычно и делается). Важным общим правилом UML является спецификация того, какую именно семантическую информацию, связанную с тем или иным графическим элементом нотации, инструмент обязан хранить. Другими словами, у каждой картинки есть оборотная сторона, где все записано, даже то, что в данном контексте ненужно или нельзя показывать на картинке. Например, инструмент может поддерживать режим, в котором часть информации о классе (скажем, список операций) не отображается на картинке или отображается не полностью. Но при этом полный список со всеми деталями во внутреннем представлении сохраняется. Более того, внутреннее представление может быть переведено в текст в формате XMI (конкретное приложение XML) без потери информации. Таким образом, в UML определен результат сериализации модели. Это важное правило, на котором базируется интероперабельность инструментов.

В первом приближении можно считать, что внутреннее представление содержит список стандартных свойств (то есть пар имя–значение), определенных для каждого элемента модели. Понятно, что такое внутреннее представление может быть однозначно (без потери информации) переведено во внешнее представление, в том числе в линейное текстовое представление. Это сделать можно и инструменты, соответствующие стандарту UML, умеют это делать. Однако такое текстовое представление не предназначено для чтения и понимания человеком. Оно неизбежно оказывается необозримо длинным, ненаглядным и плохо структурированным. Текстовое представление моделей UML не отвечает основному назначению языка (см. раздел 1.2.6), а потому не используется людьми, но используется инструментами, например, для обмена моделями.

### 1.7.2. Дополнения и украшения

У каждого элемента модели есть базовая графическая нотация. Эта нотация может быть расширена путем использования дополнительных текстовых и/или графических объектов, присоединяемых к базовой нотации. Такие дополнительные объекты так и называются — *дополнения*.[21] Дополнения позволяют показать на диаграмме те части внутреннего представления модели, которые не отображаются с помощью базовой графической нотации.

Например, базовой нотацией отношения ассоциации является сплошная линия. Эта базовая нотация может быть расширена целым рядом дополнений: именем ассоциации, указанием кратности полюсов ассоциации, ролей, направления навигации, агрегации и т. д. Еще пример: базовой нотацией класса является прямоугольник с именем класса. Эта нотация может быть расширена разделами со списками атрибутов и операций, дополнительными разделами, указанием кратности, стереотипа и т. д. (рис. 1.18).

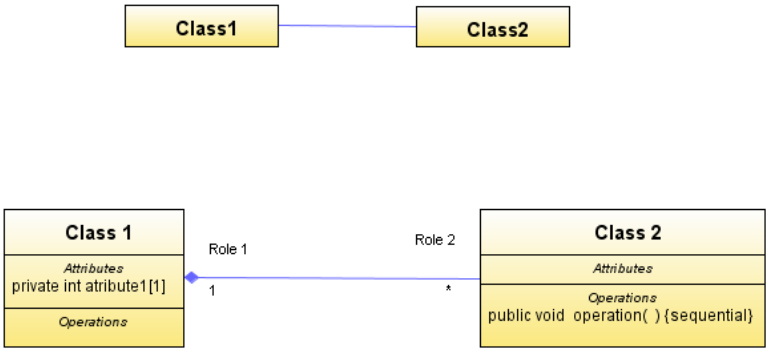



**Рис. 1.18. Пример использования дополнений (украшений)**

**ЗАМЕЧАНИЕ**

> Не следует злоупотреблять дополнениями: максимально подробное изображение не всегда самое понятное. Старайтесь не перегружать диаграммы, применяя только те дополнения, которые необходимы в данном контексте.

Как правило, инструменты позволяют детально управлять отображением дополнений, скрывая ненужные подробности или, наоборот, выявляя все детали.

### 1.7.3. Подразделения

UML является объектно-ориентированным языком, поэтому базовые понятия объектно-ориентированного подхода имеют в языке, так сказать, сквозное действие. В частности, всюду, где возможно, единообразно применяются и изображаются две дихотомии: класс–объект и интерфейс–реализация.

Дихотомия класс–объект означает, что всегда четко различается о чем идет речь: об общем описании некоторого множества однотипных объектов (т. е. о классе) или о конкретном объекте из некоторого множества однотипных объектов (т. е. об экземпляре класса). Это важное различие передается единообразно: если это

[21] В UML 2 они переименованы в украшения, что, может быть, правильно, посокольку такие элементы действительно украшают диаграмму.

конкретный объект (экземпляр класса), то его имя подчеркивается; если это класс (описание множества), то оно не подчеркивается. Одна и та же сущность в разных контекстах может рассматриваться и как класс (множество) и как экземпляр класса (элемент). Это совершенно естественно и неизбежно, хотя бы потому, что элементами множеств могут быть множества. Важно четко указать, что именно подразумевается в данном контексте. Например, класс в обычной модели является описанием множества объектов и его имя не подчеркивается. Но в то же время каждый класс является объектом — экземпляром метакласса `classifier`, определенного в метамодели UML.

Дихотомия интерфейс–реализация позволяет указать в модели, чем именно является та или иная сущность: абстрактным описанием того, чем она должна быть по отношению к другим сущностям или конкретным описанием того, чем сущность физически является. Наиболее привычный пример из программирования: заголовок функции является ее интерфейсом, а тело функции является ее реализацией. В некоторых языках программирования, например, в C, даже синтаксически различаются объявление функции (интерфейс) и описание функции (интерфейс вместе с реализацией). В UML для этой цели используется следующий синтаксический прием: если это абстрактный интерфейс, то при записи имени используется курсивное начертание шрифта, если конкретная реализация — используется прямое начертание.

### *1.7.4. Механизмы расширения*

*Механизмы расширения* — это встроенный в язык способ изменить язык. Авторы UML при унификации различных методов постарались включить в язык как можно больше различных средств (удерживая объем языка в рамках разумного), так чтобы язык оказался применимым в разных контекстах и предметных областях. Но нельзя объять необъятного![22] Вполне могут возникать и возникают случаи, когда стандартных элементов моделирования не хватает или они не вполне адекватны.

#### Язык ядро и язык оболочка

> В период бурного языкотворчества в 60-70 годах прошлого века при сравнительном обсуждении языков программирования было принято противопоставлять языки по включенному в них изначально набору готовых средств и имеющимся в языке механизме определения новых средств. Язык, который обходился минимумом базовых средств, а все необходимое предлагалось доопределять средствами самого языка, называли язык ядро. Язык, в который изначально включалось множество готовых средств, может быть даже частично дублирующих друг друга, называли языком оболочкой. Хрестоматийный пример: язык ядро — Algol 68, язык оболочка — PL/1. Оба типа языков имеют свои достоинства и свои недостатки. Язык ядро подкупает своей лаконичностью и изяществом — программист может всегда расширить язык нужными для данной задачи средствами. Это так, в принципе может, но всегда ли это получится наилучшим образом? Язык оболочка предлагает готовые средства — бери и пользуйся. Отлично, но чтобы воспользоваться, нужно знать *все* средства, а их утомительно много…

[22] Точнее говоря, можно, но издержки будут велики.

> С этой точки зрения UML, несомненно — язык оболочка: авторы включили в него все что можно. Но в то же время он имеет значительный "ядерный потенциал" — стандартные механизмы расширения.

Механизмы расширения позволяют определять новые элементы модели на основе существующих управляемым и унифицированным способом. Таких механизмов три:

- помеченные значения;
- ограничения;
- стереотипы.

Эти механизмы не являются независимыми — они тесно связаны между собой.

*Помеченное значение* — это пара: имя свойства и значение свойства, которую можно добавить к любому стандартному элементу модели.

Другими словами, помеченное значение — это свойство, добавляемое к внутреннему представлению модели (см. раздел 2.4.1). К любому элементу модели можно добавить любое помеченное значение, которое будет хранится также, как и все стандартные свойства данного элемента. Способ обработки помеченного значения, определенного пользователем, стандартом не определяется и отдается на откуп инструменту.

Помеченные значения записываются в модели в виде строки текста, имеющей следующий синтаксис: в фигурных скобках указывается пара: имя и значение, разделенные знаком равенства. Можно указывать сразу список пар, разделяя элементы списка запятыми. Если из контекста ясно, какое значение является значением по умолчанию, то знак равенства вместе со значением можно опустить.[23] На рис. 2.19 приведен пример использования помеченных значений.

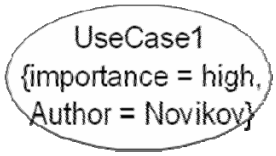



**Рис. 1.19. Пример использования помеченных значений**

*Ограничение* — это логическое утверждение относительно значений свойств элементов модели.

Логическое утверждение может иметь два значения: истина и ложь, то есть задаваемое им условие либо выполняется, либо не выполняется. Указывая ограничение для элемента модели, мы расширяем его семантику, требуя, чтобы ограничение выполнялось. Ограничение может относится к отдельному элементу или к совокупности элементов модели.

Ограничения записываются в виде строки текста, заключенной в фигурные скобки. Это может быть неформальный текст на естественном языке, логическое выражение языка программирования, поддерживаемого инструментом или

[23] Нам очень нравится этот стиль UML — если что-то и так ясно, то это можно не писать. Насколько такой стиль приятнее занудного синтаксиса традиционных языков программирования!

выражение на некотором другом формальном языке, специально включенного для этой цели в UML. Аналогично помеченным значениям, ограничение можно написать после имени элемента или в отдельном примечании, присоединенном к элементу. На рис. 1.20 приведен пример использования ограничения.

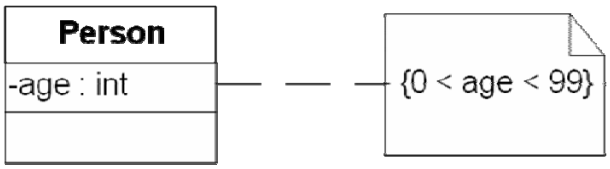



**Рис. 1.20. Пример использования ограничения**

### ЗАМЕЧАНИЕ

Для помеченных значений и ограничений одна и та же синтаксическая конструкция — текст в фигурных скобках — используется не случайно. На самом деле помеченное значение можно считать ограничением. А именно, если в модели указано помеченное значение {A=B}, то это можно рассматривать как запись условия: "свойство A обязательно имеет значение B".

*Стереотип* — это определение нового элмента моделирования в UML на основе существующего элемента моделирования.

Определение стереотипа производится следующим образом. Взяв за основу некоторый существующий элемент модели, к нему добавляют новые помеченные значения (расширяя тем самым внутреннее представление), новые ограничения (расширяя семантику) и дополнения, то есть новые графические элементы (расширяя нотацию).

После того, как стереотип определен, его можно использовать как элемент модели нового типа. Если при создании стереотипа не использовались дополнения и графическая нотация взята от базового элемента модели, на основе которого определен стереотип, то стереотип элемента обозначается при помощи имени стереотипа, заключенного в двойные угловые скобки (типографские кавычки), которое помещается перед именем элемента модели. Если же для стереотипа определена своя нотация, например, новый графический символ, то указывается этот символ. Впрочем, для пущей ясности, можно указать как имя стереотипа, так и графический символ (значок). На рис. 2.21 приведен пример использования стереотипа «`PowerUser`», который мы определили на базе стандартного стереотипа действующего лица.

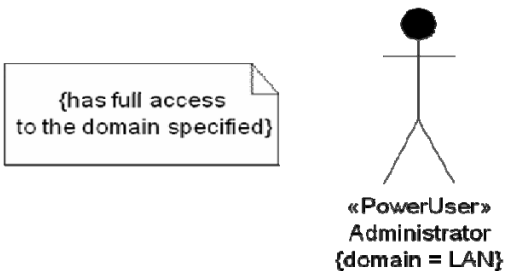



**Рис. 1.21. Пример определения и использования стереотипа**

В UML имеется большое количество предопределенных стереотипов. Стереотипы используются очень часто, поэтому примеры их применения рассредоточены по всей книге.
Стереотипы являются мощным механизмом расширения языка, однако им следует пользоваться умеренно. Определяя большое число нестандартных стереотипов, легко сделать модель непонятной никому, кроме автора.

**ЗАМЕЧАНИЕ**

> Не следует путать стереотип и отношение обобщения. Пусть суперкласс A является обобщением подкласса B. Тогда как A, так и B являются экземплярами *одного и того же* метакласса, определенного в метамодели. Если же стереотип C определен на основе типа D, то в метамодели им соответствуют *разные* метакалассы. Точнее говоря, в метамодели D является суперметаклассом для C. Другими словами, можно считать, что стереотип — это обобщение на мета уровне.

## 1.8. Общие свойства модели

Модель в целом может обладать (или не обладать) важными свойствами, которые оказывают значительное влияние на ее практическую применимость. Исчерпывающим образом описать эти свойства во вступительном обзоре невозможно — их детализация рассредоточена по все книге, но назвать и обозначить необходимо.

### *1.8.1. Правильность*

Прежде всего, модель должна удовлетворять формальным требованиям к описанию сущностей, отношений и их комбинаций. Другими словами, модель должна быть *синтаксически правильной*. Например, отношение (ребро в графе модели) всегда определяется между сущностями, оно не может просто так "висеть в воздухе". На диаграмме линия должна начинаться и заканчиваться в фигуре, иначе это синтаксическая ошибка. Некоторые тексты в модели (например, описания атрибутов и операций классов) должны иметь определенный синтаксис.
В UML первична семантика, а синтаксис вторичен. Конечно, стандарт рекомендует вполне определенный синтаксис, но это не более чем рекомендация: разработчики инструментов вправе использовать другой синтаксис, и этим правом они пользуются. Таким образом, понятие синтаксической правильности, столь ясное для традиционных языков программирования, для UML становится несколько расплывчатым.
Большинство инструментов, особенно новых, просто не позволяют ввести в модель синтаксически неправильную конструкцию. Мы будем полагаться на эту возможность инструментов и считать все модели синтаксически правильными.[24]

### *1.8.2. Непротиворечивость*

В некоторых случаях даже синтаксически правильная модель может содержать такие конструкции, семантика которых не определена или неоднозначна. Такая

[24] Мы уже упоминали, что некоторые диаграммы в книге подготовлены с помощью Visio 2000, а это приложение *не* проверяет синтаксическую правильность. Так что модели в книге синтаксически правильны настолько, насколько внимателен был автор.

модель называется противоречивой (ill formed), а модель, в которой все в порядке и семантика всех конструкций определяется однозначно, называется непротиворечивой (well formed). Например, пусть мы определим в модели, что класс A является подклассом класса B, класс B — подкласс C, а класс C — подкласс A. Каждое из этих отношений обобщения в отдельности допустимо и синтаксически правильно, а все вместе они образуют противоречие.
Далее в тексте мы всегда указываем правила непротиворечивости при обсуждении соответствующих конструкций языка. Впрочем, их не так много и по большей части они совершенно очевидны на уровне здравого смысла: синтаксические конструкции нужно использовать так, чтобы не возникало двусмысленностей при семантической интерпретации.

### Автоматическая верификация моделей

> Вопрос о том, можно ли полностью автоматически проверить любую модель UML на непротиворечивость является открытым. Скорее всего, ответ отрицательный и верификация моделей является алгоритмически неразрешимой проблемой.

Инструменты стараются проверять выполнение правил непротиворечивости, как могут, но 100% уверенности не обеспечивают. Другими словами: пользуясь любым инструментом, можно составить синтаксически правильную и семантически бессмысленную модель. Ответственность за непротиворечивость модели лежит на ее авторе.

## *1.8.3. Полнота*

Модель не создается мгновенно — она появляется в результате многочисленных итераций (рис. 2.17) и на каждой из них "по определению" не полна. Инструмент обязан дать возможность сохранить модель в любом (синтаксически правильном) состоянии с тем, чтобы в дальнейшем ее можно было дополнять и изменять.
Что такое полная модель? На этот вопрос нельзя ответить однозначно. В некоторых случаях оказывается достаточно одной диаграммы использования, а в других необходимы диаграммы всех типов, прорисованные до мельчайших деталей. Все зависит от прагматики, т. е. от того, для чего составляется модель. Если модель составляется с расчетом на автоматическую генерацию кода, то полной естественно считать модель, по которой инструмент может синтезировать работающую программу. Пока это практически невозможно — только для очень простых и ограниченных случаев. Если модель составляется с целью спецификации требований к разрабатываемому приложению, то модель можно считать полной, если заказчик и разработчик согласны считать ее таковой. Критерий, как видите, субъективный.
Мы склонны считать, что понятие полноты модели следует определять применительно к конкретному процессу разработки (см. главу 1). Грубо говоря, для каждой фазы процесса разработки есть свое понятие полноты: полная модель фазы анализа, полная модель фазы проектирования и т. д. Примеры приведены в соответствующих главах.

### *1.8.4. Вариации семантики*

Последняя тема, рассматриваемая в этой главе, является в некотором смысле уникальной особенностью UML. В описании метамодели (семантики UML) определено некоторое количество *точек вариации семантики* (semantic variation point). По сути авторы стандарта, описывая семантику какого-то понятия, говорят: "мы понимаем это так-то и так-то, но допускаем, что другие могут это понимать иначе". При реализации языка в конкретном инструменте разработчики в точке вариации семантики вправе выбрать альтернативный вариант, если он не противоречит семантике остальной части языка.

Точки вариации семантики расставлены в глубинных слоях языка, там где приходится принимать во внимание особенности реализации инструмента и конкретную систему программирования. Рядовому пользователю точки вариации семантики не заметны и он может о них не думать. Например, каждый конкретный объект является экземпляром одного конкретного класса. Но это точка вариации семантики: можно допустить динамическую типизацию (классификацию), при которой объект во время своей жизни может менять класс, которому он принадлежит.[25]

Эта замечательная особенность придает языку необходимую гибкость и универсальность: ведь UML предназначен для использования на разных платформах и в разных средах с разной операционной семантикой. Жесткая фиксация одного единственного решения связала бы разработчиков инструментов по рукам и ногам, стандарт вступил бы в конфликт с требованиями реальной жизни (такие конфликты всегда заканчиваются гибелью стандарта). Авторы языка приняли нетрадиционное и мудрое компромиссное решение: они допустили свободу, но под контролем — все точки вариации семантики явно отмечены в метамодели, в других местах никаких вольностей не допускается.

## 1.9. Выводы

- UML — это еще один формальный язык, который необходимо быстро освоить.
- Знание UML является необходимым, но не является достаточным условием построения разумных моделей программных систем.
- UML имеет синтаксис, семантику и прагматику, которые нужно знать и использовать с учетом особенностей реальной задачи и инструмента.
- Модель UML состоит из описания сущностей и отношений между ними.
- Элементы модели группируются в диаграммы и представления для наилучшего описания моделируемой системы с различных точек зрения.
- В случае необходимости элементы UML могут быть расширены и переопределены средствами самого языка.

[25] Иногда это бывает удобно и существуют языки и системы программирования, в которых это реализовано.

# Тема 2. Моделирование использования

- Что такое моделирование использования и зачем оно нужно?
- Какие средства применяются при моделировании использования?
- Как идентифицировать варианты использования и действующих лиц?
- Как реализуются варианты использования?

## 2.1. Значение моделирования использования

При первом знакомстве с диаграммами использования в UML у разработчиков программного обеспечения, особенно у опытных разработчиков, часто возникает вопрос: зачем это нужно? При этом такого вопроса относительно других средств UML не возникает, поскольку ответ на него в большинстве случаев очевиден по аналогии. Действительно, рассмотрим несколько примеров типичных ассоциаций, возникающих у программистов при первом знакомстве с UML.

Диаграммы деятельности — это не что иное, как блок-схемы алгоритмов. Разработчик программ, особенно со стажем, прекрасно понимает назначение и границы применимости блок-схем. Вопроса "зачем?" просто не возникает, возникают другие вопросы: стоит ли использовать в данном проекте, какую из альтернативных систем обозначений применять и тому подобное.

Диаграммы состояний — это конечные автоматы (см. главу 4), которые являются основным аппаратом в целом ряде областей программирования: трансляторы, логическое управление и другие.

Диаграммы классов, в которых показаны атрибуты классов и ассоциации между ними, очень похожи на диаграммы "сущность–связь", хорошо известные разработчикам приложений баз данных.

**Диаграммы сущность–связь**

> *Диаграмма сущность-связь* была предложена П. Ченом в 1976 году для графического представления моделей данных. Диаграмма сущность–связь представляет собой двудольный граф, в котором вершины первого рода соответствуют сущностям, то есть множествам однотипных записей данных (обозначаются прямоугольниками), а вершины второго рода соответствуют связям между экземплярами сущностей (обозначаются ромбами). На ребрах указывается, по сколько экземпляров каждой сущности соединяет данная связь.

Перечень возникающих у программистов ассоциаций можно продолжать и продолжать, для большинства средств UML нетрудно отыскать аналоги среди широко используемых практических методов конструирования программных систем. А вот для диаграмм использования известный аналог указать труднее. Мы попытаемся объяснить прагматику моделирования использования на конкретном примере.

### *2.1.1. Сквозной пример*

В остальных частях книги рассматривается один сквозной пример моделирования несложного приложения — информационной системы отдела кадров. Выбор примера обусловлен следующими соображениями. Во-первых, предметная область до некоторой степени знакома всем (мы полагаем, что читатель где-то работает, а значит по крайней мере один раз испытывал на себе работу отдела кадров). Таким

образом, суть задачи заранее понятна и можно сосредоточить внимание на тонкостях применения UML, а не на объяснении особенностей предметной области. Во-вторых, система отдела кадров — это типичное офисное приложение из самого распространенного класса систем автоматизации делопроизводства. UML как нельзя лучше подходит для моделирования именно таких систем и все средства языка можно проиллюстрировать естественным образом. В-третьих, автору случалось проектировать такие системы на самом деле, а не только в книге.
Итак, поставим себя на место разработчика и предположим, что в нашем распоряжении имеется следующий текст, поступивший от заказчика.

**ТЕХНИЧЕСКОЕ ЗАДАНИЕ**

> Информационная система «Отдел кадров» (сокращенно ОК) предназначена для ввода, хранения и обработки информации о сотрудниках и движении кадров. Система должна обеспечивать выполнение следующих основных функций.
>
> 1. Прием, перевод и увольнение сотрудников.
>
> 2. Создание и ликвидация подразделений.
>
> 2. Создание вакансий и сокращение должностей.

Конечно, техническое задание из одного абзаца текста и трех нумерованных пунктов — это не более чем учебный пример. Однако даже на этом примере видны многие характерные "особенности" подобных документов, которые, увы, слишком часто встречаются в реальной жизни. С одной стороны, что-то написано, а, с другой стороны не очень понятно, что делать дальше. Безо всяких объяснений заказчик использует термины свой предметной области — разработчик должен их знать и понимать. Требований к реализации нет вовсе. Функции не упорядочены по приоритетам: не ясно, что является критически важным, а чем можно поступиться в случае необходимости. В общем, раскритиковать это техническое задание в пух и прах не составляет труда. Однако, каким бы недостаточным ни было это техническое задание, после получения ответа на вопрос "кто виноват", встает вопрос "что делать?".
Можно заявить, что это техническое задание никуда не годится и отказаться работать с заказчиком, который его представил. Можно потребовать уточнить техническое задание, включив в него ответы на все вопросы разработчика. Это во всех отношениях идеальный вариант, только, к сожалению, так не бывает. Можно, наконец, попытаться что-то сделать самим с имеющимся материалом. Мы выберем третий путь, имея в виду показать, как, применяя UML, можно постепенно превратить расплывчатое описание приложения во вполне четкую модель, пригодную для реализации.

### *2.1.2. Подходы к проектированию*

Преимущества моделирования использования UML мы попробуем выявить, сравнивая его с другими подходами. Известно множество различных подходов к проектированию, то есть рекомендаций, что делать после получения технического задания. Мы рассмотрим три (наиболее нам близких) из них и попытаемся указать те подводные камни, которые позволяет обойти использование UML.

Наиболее заслуженным является, видимо, метод структурного проектирования "сверху вниз".

### Программирование сверху вниз, снизу вверх и вширь

*Программирование сверху вниз* — это обобщающее название для модульного программирования без оператора Go To методом пошагового уточнения. Английское название — Top-down approach — более точно акцентирует важнейшее достижение технологической программистской мысли, аккумулированное в этом подходе: проектирование и реализация (кодирование) программы являются двумя неразрывно связанными фазами одного процесса, причем проектирование первично, а реализация вторична. При программировании сверху вниз процесс программирования заключается в следующем. Исходная задача разлагается на подзадачи до тех пор, пока каждая отдельная подзадача не станет столь простой, что ее реализация становится очевидной. Парным к программированию сверху вниз является *программирование снизу вверх*, при котором уровень языка программирования повышается (например, с помощью определения модулей) до тех пор, пока он не станет настолько высоким и близким к исходной задаче, что ее реализация станет очевидной. Оба метода имеют очевидные достоинства, но имеют и недостатки: при программировании сверху вниз отладка возможна только по завершении всего проектирования, при программировании снизу вверх велик риск создания невостребованных модулей. На практике всегда применяется комбинация этих подходов, благо они не противоречат друг другу.

С.С. Лаврову принадлежит изобретение термина *программирование вширь*,26 когда, начиная с самого первого шага, создается и на всех последующих шагах поддерживается работоспособная версия программы. В терминах программирования сверху вниз это означает форсирование проведение одного пути в дереве последовательных уточнений до конца (ствол дерева), а затем постепенное наращивание дерева вширь. Конечно, в начале работы программа не умеет выполнять (почти) никаких нужных функций, но зато она не содержит и ненужных (ошибочных) функций и постоянно удовлетворяет требованию демонстрируемости. Такой стиль подразумевает проведение тестовых испытаний всех готовых модулей при добавлении или изменении любого модуля,27 но не откладывание отладки на окончание разработки. Это требует дополнительных трудозатрат, но оказывает чрезвычайно благотворное психологическое воздействие на разработчика, и еще более благотворное воздействие такой стиль программирования оказывает на заказчика.

Этот метод конструирования программ хорошо известен, надежен и легко применим во всех случаях. Мы даже рискнем утверждать, что большая часть ныне

---

[26] В современной языковой практике используется термин, являющийся транслитерацией с английского — инкрементальная разработка, которому автор, уступая обстоятельствам непреодолимой силы, уже позволил проскользнуть в текст первой главы.

[27] Т.е. нет никакой автономной отладки, отладка с самого начала комплексная.

используемого программного обеспечения изготовлена именно таким способом. Однако наши личные наблюдения показывают, что в результате структура приложения оказывается соответствующей структуре команды разработчиков, а не исходной задаче. Если в разработке информационной системы отдела кадров задействовано два ведущих разработчика, то будет выделено две основных подсистемы, а если три, то и подсистем окажется три. Это, разумеется не закон природы, но, тем не менее, при пошаговой детализации произвол выделения подсистем неизбежен.

Рассмотрим второй подход. Всякому ясно, что информационная система отдела кадров — это приложение баз данных.[28] При проектировании приложений баз данных первый шаг хорошо известен: нужно составить схему базы данных, то есть определить состав таблиц базы и атрибутов в таблицах, назначить первичные ключи в таблицах и установить связи между таблицами с помощью внешних ключей.

### Схема базы данных

*Схема базы данных* — это описание данных и взаимосвязей между ними в базе данных. В это описание включают, по меньшей мере, следующую информацию:

- таблицы, хранящихся в базе;
- поля записей таблиц;
- первичные ключи таблиц, то есть поля, значение которых однозначно идентифицирует запись в таблице;
- связи, то есть указание на то, как записи одной таблицы связаны с записями другой таблицы.

Часто схему базы данных представляют в графической форме, в виде диаграммы специального вида.

Опять мы не можем привести ни одного аргумента общего характера против такого образа действий, и ограничиваемся частным примером. Среди десятка знакомых нам информационных систем отдела кадров примерно в половине случаев (в том числе в коммерчески продаваемых системах) табельный номер[29] сделан ключом таблицы сотрудников. На первый взгляд такое решение допустимо: табельный номер уникален, однозначно соответствует сотруднику и может быть использован в качестве ключа (в отличии от фамилии сотрудника, которая вполне может не быть уникальной). Однако, полный анализ всего приложения показывает, что это грубая проектная ошибка, порождающая проблемы на позднейших стадиях разработки и заведомую неэффективность в работе системы. Чтобы понять, в чем ошибка, подумайте — как будет выполняться операция "перевод сотрудника"?

Наконец, рассмотрим самый модный объектно-ориентированный подход. Апологеты этого подхода первый шаг проектирования описывают примерно так: нужно выделить словарь предметной области (то есть набор основных понятий),

---

[28] Это предрассудок, не обязательно использовать СУБД для решения таких задач, но предрассудок широко распространенный и укоренившийся.

[29] Грубо говоря, табельный номер — это номер записи в таблице штатного расписания.

сопоставить этим понятиям классы проектируемой системы, определить их атрибуты и операции и дальше все пойдет как по маслу.

## Словарь предметной области

*Словарь предметной области* — часто упоминаемое, но очень плохо определенное понятие объектно-ориентированного анализа и проектирования. Все теоретики объектно-ориентированного подхода единодушно утверждают, что словарь предметной области — важнейший артефакт, лежащий в основе анализа и проектирования. Разумеется, с этим нельзя не согласиться — правильный научный подход к любой задаче требует сначала договориться о терминах. К сожалению, на этом единодушие заканчивается: общепринятых формальных требований к словарю мы не знаем.

Для примера в таблице 2.1 приведен фрагмент словаря для предметной области связанной, с информацией о счетах.

**Таблица 2.1.** Словарь предметной области

| Термин | Категория | Пояснение |
|---|---|---|
| Клиент | Внешняя сущность | Клиент банка |
| Счет | Класс постоянно хранимых объектов | Счет клиента в банке |
| Остаток | Атрибут | Атрибут класса Счет, значение которого является текущей денежной суммой на счете |
| Номер | Атрибут | Атрибут класса Счет, значение которого однозначно идентифицирует объект класса Счет |

И здесь нечего возразить! Если словарь выделен, то действительно, дальше все идет отлично. Но кто выделяет словарь? Если разработчик, то он должен быть оракулом в конкретной предметной области, в противном случае никто не может гарантировать полноту и адекватность словаря, а ошибки при выделении базовых классов относятся к самым тяжелым проектным ошибкам.

Подводя итоги нашему рассмотрению альтернативных подходов к проектированию отметим следующее обстоятельство. Во всех трех рассмотренных случаях первый шаг проектирования сразу выполняется в терминах проектируемой системы, причем однобоко. Действительно, в первом случае за основу берется структура будущего кода, во втором — структура хранимых данных, а в третьем — структура межмодульных интерфейсов. Еще раз оговоримся: мы не видим в бегло упомянутых альтернативных подходах к проектированию никаких непреодолимых пороков. Опытный мастер легко спроектирует информационную систему отдела кадров пользуясь любым подходом. Но новичок может столь же легко допустить при проектировании трудно исправимую ошибку. Запомним это обстоятельство и обратимся к рассмотрению моделирования использования.

### *2.1.3. Преимущества моделирования использования*

Отвлечемся пока от технических деталей нотации диаграмм использования (они рассмотрены в следующем параграфе) и рассмотрим, что предлагается делать на первом шаге моделирования использования.

Наш язык и мышление устроены так, что самой простой, понятной и четкой формой изложения мыслей являются так называемые простые утверждения. Простое утверждение имеет следующую грамматическую форму: подлежащее — сказуемое — прямое дополнение. Или, в логических терминах, субъект — предикат — объект. Например: начальник увольняет сотрудника, директор создает отдел. Но по сути, именно простые утверждения и записаны на диаграмме использования! Действительно, действующее лицо — это субъект, а вариант использования — предикат (вместе с объектом). Таким образом, моделирование использования отличается от рассмотренных подходов тем, что предполагает явное формулирование требований к системе на самом начальном этапе разработки.

Попробуем перечислить те преимущества, который дает этот подход по сравнению с другими.

- *Простые утверждения*. Моделирование использования фактически позволяет переписать исходное техническое задание (или просто записать, если никакой исходной формулировки требований не было) в строгой и формальной, но в тоже время очень простой и наглядной графической форме, как совокупность простых утверждений относительно того, что делает система для пользователей. Конечно, использование такой формы не гарантирует от ошибок (вряд ли гарантия от ошибок вообще возможна), но благодаря простоте и наглядности формы их легче заметить.
- *Абстрагирование от реализации*. Моделирование использования предполагает формулирование требований к системе абсолютно независимо от ее реализации. Другими словами, представление использования описывает только, *что* делает система (но не *как* это делается и не *зачем* это нужно делать). Заметим, что другие подходы, используя на первых шагах термины и понятия реализации (структура программы, структура данных, структура интерфейсов) накладывают невольные ограничения на реализацию, которые не вытекают из существа задачи, а значит могут служить источником неэффективности и ошибок.
- *Декларативное описание*. Каждый вариант использования описывает (а вернее сказать, именует) некоторое множество последовательностей действий, доставляющих значимый для пользователя результат. Однако, никакого императивного описания представление использования не содержит, в модели нет указаний на то, какой вариант использования должен выполняться раньше, а какой позже, то есть нет описания алгоритма, а значит, нет алгоритмических ошибок.
- *Выявление границ*. Представление использование определяет границы системы и постулирует существование во внешнем мире использующих ее агентов. Описание системы в виде черного ящика с определенными интерфейсами кажется очень похожим на представление использования, но здесь есть важное различие, которое часто упускается из вида. Если ограничиться только описанием интерфейсов, то очень легко допустить ошибки следующего типа:

предусмотреть интерфейс, который не нужен, потому что им никто не пользуется. Или, аналогично, забыть интерфейс, который необходим определенной категории пользователей. На диаграмме использования одинокие и покинутые действующие лица и варианты использования обнаруживаются с первого взгляда.

Наш вывод таков: моделирование использования безопаснее и надежнее альтернативных методов, то есть при прочих равных условиях позволяет совершить меньше грубых проектных ошибок на ранних стадиях проектирования. В этом заключается основное преимущество данного метода.

**ЗАМЕЧАНИЕ**

> Не следует думать, что моделирование использования — это панацея от всех болезней разработки программного обеспечения. Существуют области, где моделирование использования практически ничего не дает. Например: разработка автономного компилятора языка программирования.

## 2.2. Диаграммы использования

Диаграмма использования является основным и, по нашему мнению, единственным средством моделирования использования в UML. На диаграмме использования применяются следующие типы сущностей:

- действующие лица;
- варианты использования;
- примечания;
- пакеты.

Между этими сущностями устанавливаются следующие типы отношений:

- ассоциация между действующим лицом и вариантом использования;
- обобщение между действующими лицами;
- обобщение между вариантами использования;
- зависимости (двух стереотипов) между вариантами использования;
- зависимости между пакетами.

На рисунке 2.1. представлена графическая нотация элементов диаграмм использования, справедливая дл версии UML 2.0.

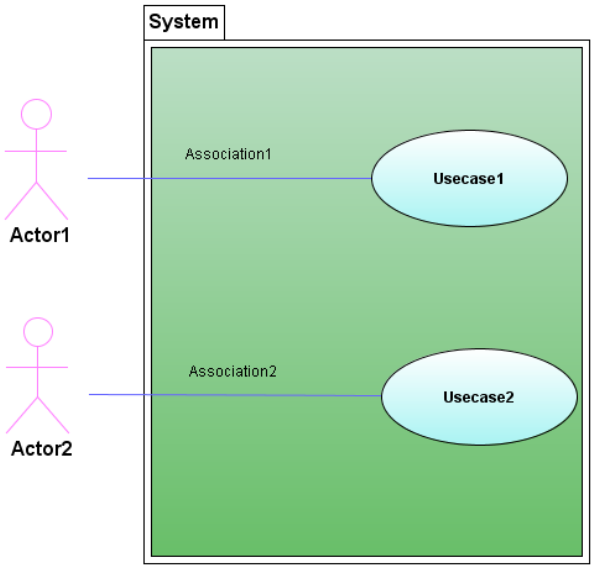



**Рис. 2.1. Нотация диаграмм использования (UML 2.0)**

Применение пакетов для группировки элементов модели унифицировано в UML для всех типов диаграмм, поэтому мы отложим рассмотрение пакетов и зависимостей между ними до главы 5, а остальные элементы рассмотрим по порядку.

### *2.2.1. Действующие лица*

Вопрос о выделении (или идентификации) действующих лиц при составлении модели — один из самых болезненных. Неудачный выбор действующих лиц может отрицательно повлиять на всю модель в целом. Здесь легко впасть в крайность: объявить, что имеется одно действующее лицо (внешний мир), взаимодействующее со всеми вариантами использования или, наоборот, придумать искусственных действующих лиц для каждого варианта использования. Оба экстремальных варианта являются, по существу, моделью черного ящика и сводят к нулю преимущества моделирования использования, рассмотренные в предыдущем параграфе. Формального метода идентификации действующих лиц не существует. Здесь мы перечислим некоторые приемы, которые полезно, по нашему мнению, иметь в виду при выделении действующих лиц и покажем применение этих приемов на нашем сквозном примере. Для начала укажем более детальное определение действующего лица.

С синтаксической точки зрения *действующее лицо* — это стереотип классификатора, который обозначается специальным значком. Для действующего лица указывается только имя, идентифицирующее его в системе. Семантически действующее лицо — это множество логически взаимосвязанных ролей.

*Роль* в UML — это интерфейс, поддерживаемый данным классификатором в данной ассоциации.

С прагматической точки зрения главным является то, что действующие лица находятся *вне* проектируемой системы (или рассматриваемой части системы).
В типовых случаях различные действующие лица назначаются для категорий пользователей (если их удается выделить естественным образом), внешних программных и аппаратных средств (если система взаимодействует с таковыми).

### О термине "действующее лицо"

Мы выбрали для перевода термина actor словосочетание "действующее лицо", исходя из следующей, довольно точной аналогии. Рассмотрим общеизвестное употребление термина "действующее лицо". Имеются литературные сочинения определенного жанра, которые называются пьесы (например, "Трагическая история о Гамлете, принце датском", автор — Вильям Шекспир). В начале пьесы присутствует список действующих лиц, элементы которого имеют примерно такой вид: "Гамлет — сын прежнего и племянник нынешнего короля". Пьеса может быть *поставлена* — например, постановка Юрия Любимова в Театре на Таганке. В конкретный вечер может быть дан *спектакль*, в котором у действующих лиц пьесы имеются конкретные исполнители (например, "в роли Гамлета — Владимир Высоцкий"). Пьеса, постановка и спектакль — это разные вещи. Теперь вернемся к нашей аналогии. Пьеса — это модель (пьеса имеет все характерные признаки модели), постановка — это программа, реализующая модель, спектакль — выполнение этой программы. Так вот, actor в модели UML — это действующее лицо пьесы, но никак не конкретный исполнитель в спектакле. Обратите внимание, что Шекспир не забывает указывать важнейшие интерфейсы действующих лиц своих пьес. То, что действующее лицо — это классификатор, а не экземпляр, особенно заметно, если в конце списка действующих лиц присутствует, например, такая запись: "Лорды, леди, офицеры, солдаты, матросы, вестовые и свитские".

Рассмотрим наш пример с информационной системой отдела кадров. Про внешние программные и аппаратные средства в техническом задании ничего не сказано и этот вопрос пока разумно оставить в стороне. Трудно представить себе организацию, в которой реорганизация внутренней структуры и найм персонала выполняются автоматически, без участия человека, поэтому у нашей системы, очевидно, будут пользователи.
Выделение категорий пользователей происходит, как правило, неформально: из соображений здравого смысла и собственного опыта. Тем не менее, несколько советов мы можем дать. Имеет смысл отнести пользователей к разным категориям, если наблюдаются следующие признаки: пользователи участвуют в разных (независимых) бизнес-процессах; пользователи имеют различные права на выполнение действий и доступ к информации; пользователи взаимодействуют с системой в разных режимах: от случая к случаю, регулярно, постоянно.
Опираясь на собственные советы применительно к нашему примеру мы в первом приближении склонны выделить две категории пользователей:

- менеджер персонала, который работает с конкретными людьми;
- менеджер штатного расписания, который работает с абстрактными должностями и подразделениями.

Бизнес-процесс пользователя первой категории включает в себя не только работу с приложением, но и беседы с конкретными людьми, интервью и тому подобное, чем явно отличается от других бизнес-процессов предприятия.
Пользователи второй категории, очевидно, должны иметь специальные права доступа, поскольку вряд ли допустимо, чтобы кто угодно мог создавать и уничтожать подразделения на предприятии.
На рис. 2.2 мы начинаем формирование представление использования информационной системы отдела кадров. Менеджер персонала имеет имя `Personnel Manager`, а менеджер штатного расписания — `Staff Manager`, в соответствии с используемой дисциплиной имен.

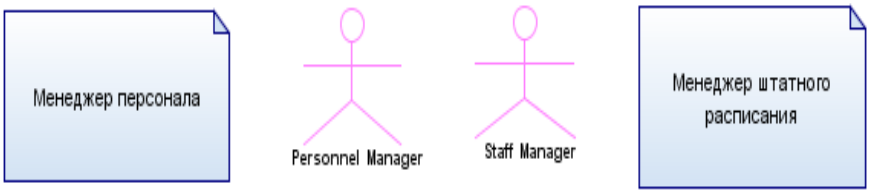



**Рис. 2.2. Действующие лица информационной системы отдела кадров**

## Дисциплина имен

Единственными лексемами языка, которыми программист пользуется совершенно произвольно, являются имена (идентификаторы).[30] Недисциплинированный программист использует свободу выбора идентификаторов для глупых шуток. Умелый программист использует эту свободу для повышения читабельности программы (или модели).

Читабельность программы повышается, если пишущий программу придерживается определенных правил формирования идентификаторов, а читающий программу знает эти правила. Правила формирования идентификаторов в программе называются *дисциплиной имен*. Дисциплина имен должна учитывать по меньшей мере три фактора:

- набор различных характеристик именуемых объектов (и области значений этих характеристик), которые учитываются в данной дисциплине;
- набор правил формирования имен по заданным значениям выбранных характеристик (с учетом возможных лексических ограничений системы программирования);
- набор операций (помимо операции чтения человеком), которые выполняются над множествами имен.

Важным вопросом при формировании имен является выбор набора символов. Поскольку при моделировании и кодировании используются, как правило, разные инструменты, то маловероятно, что они все

[30] Это удивительная особенность языков программирования. Во всех остальных видах человеческой деятельности правила именования объектов более или менее регламентированы.

окажутся локализованными, причем с одинаковым пониманием особенностей русского языка. Отсюда следует, что для идентификаторов в программах обычно используют буквы латинского алфавита. Для условных (кодовых) частей идентификаторов это не важно и даже удобно: иероглиф не должен быть похож на слово.[31] Но для содержательных (семантических) частей желательно, чтобы они являлись узнаваемыми словами или словосочетаниями. Использование транслитерированных русских слов выглядит ужасно.[32] Использование правильных английских слов хорошо, но требует, чтобы все читатели и писатель программы *одинаково* (хорошо или плохо) владели английским (что на практике встречается очень редко). Выход заключается в том, чтобы составить жаргонный словарь из слов, сокращений и аббревиатур и пользоваться только им.[33]

Для UML пока что нет достаточно устоявшейся дисциплины имен, но некоторый набор рекомендаций можно найти в литературе. Мы, по возможности, следуем этим рекомендациям и при случае пересказываем их. В частности, в качестве имен действующих лиц рекомендуется использовать существительное (возможно с определяющим словом), а в качестве имен вариантов использования — глагол (возможно, с дополнением). Эти правила основаны на семантике моделирования использования, которую мы попытались объяснить в начале главы.

### *2.2.2. Варианты использования*

Выделение вариантов использования — ключ ко всему дальнейшему моделированию. На этом этапе определяется функциональность системы, то есть, что она должна делать.

**Отступление о терминологии**

К величайшему сожалению, отечественная практика образования терминов в области информационных технологий неудовлетворительна. Термины либо переводятся (часто неудачно), либо транслитерируются с английского, без учета русских языковых традиций. За примерами ходить далеко не нужно: файл, компьютер, сайт. Еще один характерный пример — использованное нами слово "функциональность" (functionality). Речь идет о множестве функций, которые выполняет система, а называется это именем свойства. На вопрос "какова функциональность системы?" хотелось бы отвечать так: "велика, мала, отсутствует", а приходится отвечать "функциональность состоит из…". Это можно было бы назвать, например, "функциональные обязанности", но "к чему бесплодно спорить с веком".

---

[31] Локализованные версии языков программирования не популярны. Язык программирования далек от естественного и не должен быть к нему близок. Служебные слова языка программирования – это иероглифы, а не слова из букв. Иероглифы легче выучить и использовать, если они ни на что не похожи.

[32] Высший класс – записывать русские слова латинскими буквами, которые по написанию совпадают с кириллическими буквами. К сожалению, таковых немного (например, среди прописных: A, B, C, E, H, K, M, O, P, T, X) и этот метод требует невероятной изобретательности.

[33] Например, в таком стиле: CurRecNum означает "номер текущей записи".

Нотация для варианта использования очень скудная — это просто имя, помещенное в овал (или помещенное под овалом — такой вариант тоже допустим). Другими словами, функции, выполняемые системой, на уровне моделирования использования никак не раскрываются — им только даются имена.

Семантически *вариант использования* — это описание множества возможных последовательностей действий (событий), приводящих к значимому для действующего лица результату.

Каждая конкретная последовательность действий называется *сценарием*. Таким образом, вариант использования является классификатором (см. главу 3), экземплярами которого являются сценарии.

Прагматика варианта использования состоит в том, что среди всех последовательностей действий, могущих произойти при работе приложения, выделяются такие, в результате которых получается явно видимый и достаточно важный для действующего лица (в частности, для пользователя) результат.

Сказанное для действующих лиц уместно повторить и для вариантов использования: выбор вариантов использования сильно влияет на качество модели, а формальные методы предложить трудно — помогают только опыт и чутьё. Если в вашем распоряжении есть техническое задание, пункты которого естественным образом переводятся в варианты использования, знайте, что вам сильно повезло. Если техническое задание представляет собой смесь очевидных пожеланий пользователя, смутных утверждений и конкретных требований (как обычно бывает), то попробуйте поискать в тексте отглагольные существительные и глаголы с прямым дополнением: может статься, что в них зашифрованы варианты использования. Если у вас вовсе нет технического задания, составьте его, пользуясь исключительно простыми утверждениями.

В нашем примере простой анализ текста технического задания выявляет семь вариантов использования:

- прием сотрудника;
- перевод сотрудника;
- увольнение сотрудника;
- создание подразделения;
- ликвидация подразделения;
- создание вакансии;
- сокращение должности;

Опираясь на знание предметной области, которое не отражено в техническом задании (характерный случай), заметим, что термин "вакансия" является сокращением оборота "вакантная должность", то есть должность в некотором особом состоянии. Само же слово "должность" многозначно. Это может быть и обозначение конкретного рабочего места — позиции в штатном расписании, и обозначение совокупности таких позиций, имеющих общие признаки: функциональные обязанности, зарплата и т. п. Например: "в организации различаются должности: программист, аналитик, руководитель проекта" или "в отделе разработки предусмотрены 9 программистов, 3 аналитика и 2 руководителя проектов". Кадровые работники легко различают эти случаи по контексту. Примем рабочую гипотезу о том, что автор технического задания использовал слово

должность в первом смысле и получим набор вариантов использования, представленный на рис. 2.3.

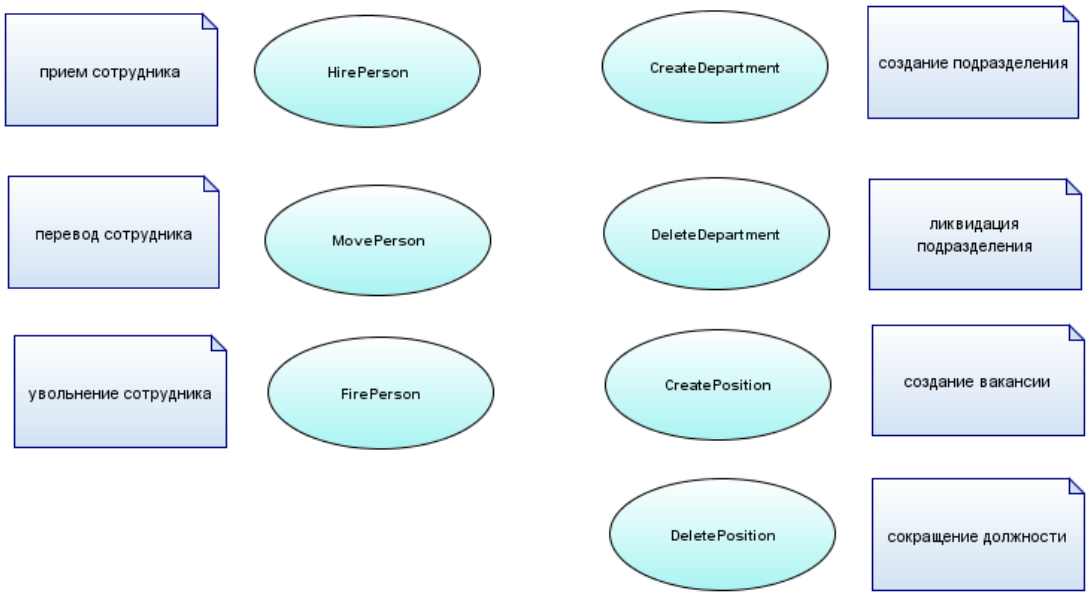



**Рис. 2.3. Варианты использования информационной системы отдела кадров**

### *2.2.3. Примечания*

Третьим типом сущности, применяемым на диаграмме использования, является примечание. Заметим, что примечания являются очень важным средством UML, значение которого часто недооценивается начинающими пользователями. Примечания можно и нужно употреблять на всех типах диаграмм, а не только на диаграммах использования. UML является унифицированным, но никак не универсальным языком — при моделировании проектировщик часто может сказать о моделируемой системе больше, чем это позволяют сделать строгая, но ограниченная нотация UML. В таких случаях наиболее подходящим средством для внесения в модель дополнительной информации является примечание.

В отличии от большинства языков программирования примечания в UML синтаксически оформлены с помощью специальной нотации и выступают на тех же правах, что и остальные сущности. А именно, примечание имеет свою графическую нотацию — прямоугольник с загнутым уголком, к котором находится текст примечания. Примечания могут находиться в отношении соответствия с другими сущностями — эти отношения изображаются пунктирной линией без стрелок.

Примечания содержат текст, который вводит пользователь — создатель модели. Это может быть текст в произвольном формате: на естественном языке, на языке программирования, на формальном логическом языке, например, OCL и т. д. Более того, если возможности инструмента это позволяют, в примечаниях можно хранить гиперссылки, вложенные файлы и другие внешние по отношению к модели артефакты.

В UML последовательно проводится следующий важный принцип: вся информация, которую пользователь соизволил внести в модель, должна быть сохранена инструментом во внутреннем представлении модели и предъявлена по первому требованию, даже если инструмент не умеет обрабатывать эту информацию. Примечания являются важнейшим средством реализации этого принципа.

Примечания могут иметь стереотипы. В UML определены два стандартных стереотипа для примечаний:

- `requirement` — описывает общее требование к системе;
- `responsibility` — описывает ответственность класса.

Примечания первого типа часто применяют в диаграммах использования, а примечания второго типа — в диаграммах классов.

Возвращаясь к нашему примеру, будет совсем не лишним указать, что информацию о состоянии кадров нужно хранить постоянно, т. е. она не должна исчезать после завершения сеанса работы с системой (рис. 2.5).

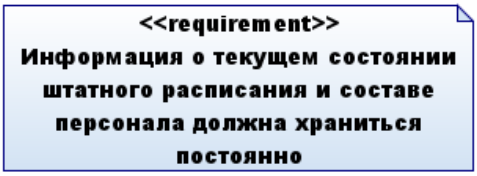



**Рис. 2.4. Требование к информационной системе отдела кадров**

### *2.2.4. Отношения на диаграммах использования*

Как уже было отмечено в начале параграфа, на диаграммах использования применяются следующие основные типы отношений:

- ассоциация между действующим лицом и вариантом использования;
- обобщение между действующими лицами;
- обобщение между вариантами использования;
- зависимости между вариантами использования;

*Ассоциация между действующим лицом и вариантом использования* показывает, что действующее лицо тем или иным способом взаимодействует (предоставляет исходные данные, потребляет результат) с вариантом использования.

Другими словами, эта ассоциация обозначает, что действующее лицо так или иначе, но обязательно непосредственно участвует в выполнении каждого из сценариев, описываемых вариантом использования. Ассоциация является наиболее важным и, фактически, обязательным отношением на диаграмме использования. Действительно, если на диаграмме использования нет ассоциаций между действующими лицами и вариантами использования, то это означает, что система не взаимодействует с внешним миром. Такие системы, равно как и их модели, не имеют практического смысла.

Применительно к нашему примеру в первом приближении можно обозначить ассоциации, представленные на рис. 2.5.

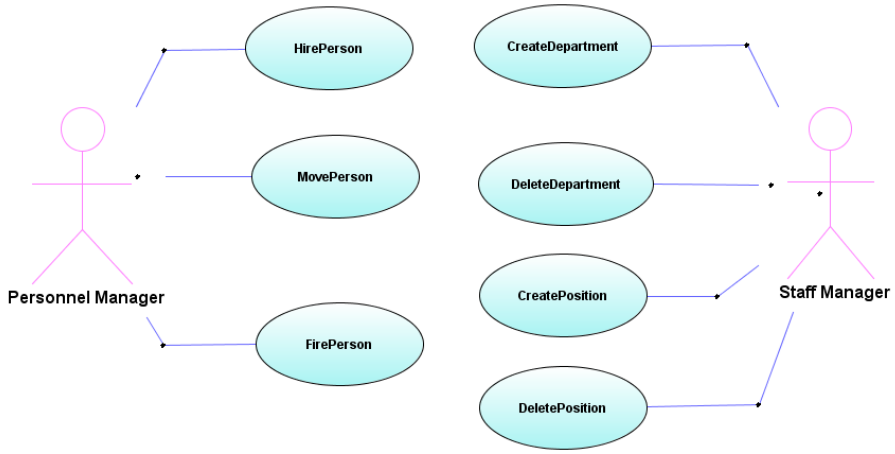



**Рис. 2.5. Ассоциации между действующими лицами и вариантами использования информационной системы отдела кадров**

| |
|---|
| *Обобщение между действующими лицами* показывает, что одно действующее лицо наследует все свойства (в частности, участие в ассоциациях) другого действующего лица. |

Такое обобщение является весьма мощным средством моделирования.
Во-первых, с помощью обобщения между действующими лицами легко показать иерархию категорий пользователей системы, в частности, иерархию прав доступа к выполняемым функциям и хранимым данным. Например, мы можем предположить, что среди пользователей информационной системы отдела кадров имеется категория пользователей (назовем ее высшее руководство), которым разрешен доступ к любым данным и к любым операциям. Этот предположение можно отразить в модели системы так, как показано на рис. 2.6.

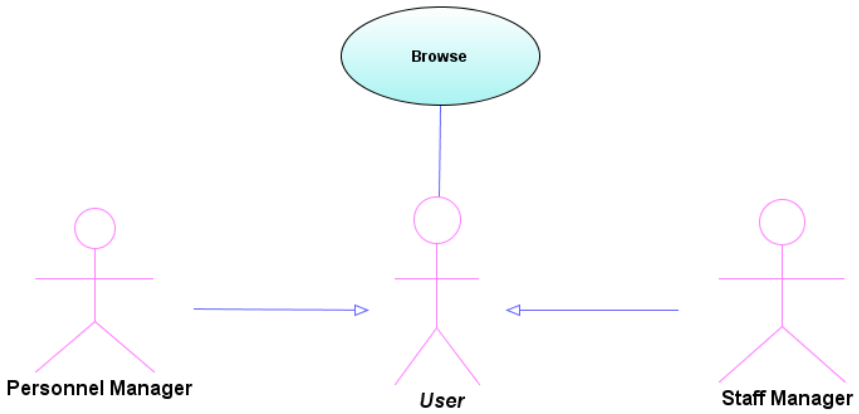



**Рис. 2.6. Иерархия категорий пользователей информационной системы отдела кадров**

Во-вторых, действующее лицо, будучи классификатором, может быть абстрактным классификатором, то есть таким классификатором, который не может иметь непосредственных экземпляров. Введение абстрактных действующих лиц позволяет без потери информации сократить количество непосредственных

ассоциаций в модели, сделав ее более лаконичной, а значит более наглядной и полезной. Например, в информационной системе отдела кадров полезно предусмотреть вариант использования, при котором данные не меняются, а только просматриваются пользователем. Если мы проектируем систему отдела кадров для обычной организации, а не для государственной секретной службы, то разумно предположить, что просматривать данные могут все категории пользователей. В этом случае можно ввести новый вариант использования и установить ассоциации со всеми действующими лицами, а можно поступить так, как показано на рис. 2.7.

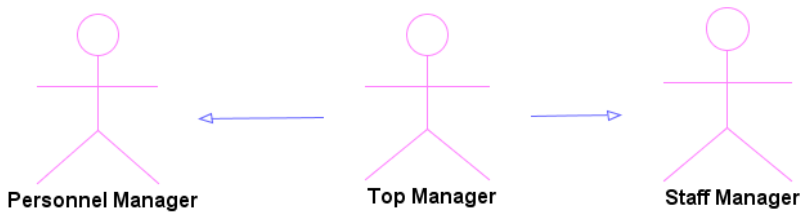



**Рис. 2.7. Абстрактное действующее лицо в информационной системе отдела кадров**

*Обобщение между вариантами использования* показывает, что один вариант использования является частным случаем (подмножеством множества сценариев) другого варианта использования.

Обобщающий вариант использования, будучи классификатором, может быть абстрактным классификатором. Например, такой важный для сотрудника вариант использования, как увольнение, на самом деле является абстракцией: в каждом конкретном случае имеет место ровно один из возможных частных случаев увольнения, которые все приводят к одному и тому же результату с точки зрения менеджера персонала, но весьма различны с точки зрения сотрудника. Допустим, что в нашей информационной системы отдела кадров предусмотрены два типа увольнения: увольнение по инициативе администрации и увольнение по собственному желанию. Данное обстоятельство можно отразить в модели так, как показано на рис. 2.8.

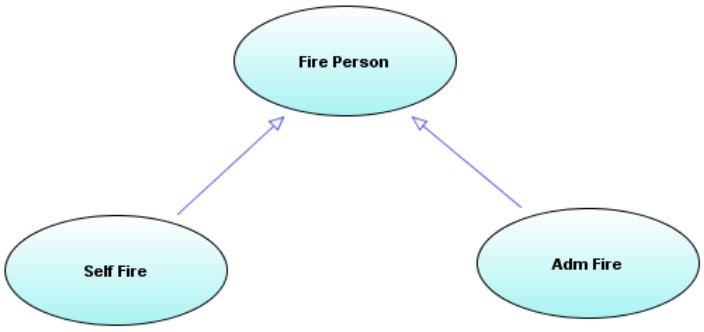



**Рис. 2.8. Обобщение между вариантами использования в информационной системе отдела кадров**

*Зависимость между вариантами использования* показывает, что один вариант использования зависит от другого варианта использования.

В UML имеются два стандартных стереотипа зависимости между вариантами использования, которые в некотором смысле двойственны друг другу:

- `include` — показывает, что сценарий независимого варианта использования включает в себя в качестве подпоследовательности действий сценарий зависимого варианта использования;
- `extend` — показывает, что в сценарий зависимого варианта использования может быть в определенном месте вставлен в качестве подпоследовательности действий сценарий независимого варианта использования.

На первый взгляд не очень понятно, чем отличается семантика этих зависимостей — ведь обе они отражают отношение включения для последовательностей действий. Нам будет удобнее объяснить тонкости семантики этих отношений в первом разделе следующего параграфа, а здесь мы ограничимся примерами.

Рассмотрим еще раз вариант использования увольнение сотрудника (один из самых сложных в реальных системах управления персоналом). Известно, что при увольнении сотрудника следует в целях информационной безопасности удалить (или заблокировать) учетную запись пользователя в локальной сети организации. Причем эта последовательность действий должны быть выполнена в любом сценарии увольнения. С другой стороны, при выполнении определенных условий, при увольнении иногда выплачивается некоторая денежная компенсация (за неиспользованный отпуск, выходное пособие при сокращении и т. д.). Все это примеры последовательностей действий (т. е. вариантов использования), которые вполне могут быть востребованы как при увольнении, так и помимо него. Отношения зависимости между всеми этими вариантами использования могут быть отражены на диаграмме использования, например, так, как показано на рис. 2.9.

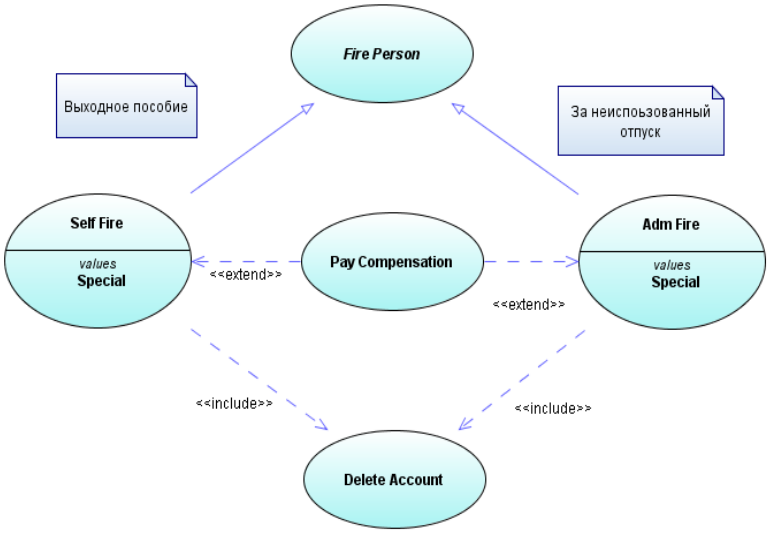



**Рис. 2.9. Зависимости между вариантами использования в информационной системе отдела кадров**

## 2.3. Реализация вариантов использования

После того, как построено представление использования, то есть выделены действующие лица, варианты использования и установлены отношения между ними, встает естественный вопрос: что дальше? То есть, как далее следует продолжать моделирование средствами UML?

Вообще говоря, ответ может быть такой: ничего больше не делать на UML. Представление использования, если оно тщательно продумано и детально прорисовано, является формой технического задания, содержащей достаточно информации для дальнейшего проектирования и реализации любым другим методом. Может статься, что в коллективах программистов, в совершенстве владеющих какой-либо другой методикой проектирования и реализации, такое ограниченное использование UML окажется вполне оправданным.

Однако, UML содержит средства не только для моделирования использования, но и для поддержки всех остальных фаз процесса разработки, и эти средства по меньшей мере не хуже альтернативных. Таким образом, UML полезно знать, даже если он не используется в процессе разработки или используется только частично. Поэтому мы рассмотрим все средства UML в оставшейся части книги. Цель этого параграфа — изложить наше видение возможных путей перехода от моделирования использования к других видам моделирования.

Действующие лица находятся вне системы — с ними ничего делать не нужно. Можно сказать, что действующие лица уже выполнили свою задачу, просто появившись в модели системы. Таким образом, переход от моделирования использования к другим видам моделирования состоит в уточнении, детализации и конкретизации вариантов использования. В представлении использования мы показали, *что* делает система, теперь нужно определить, *как* это делается. Это обычно называется реализацией вариантов использования.

### 2.3.1. Текстовые описания

Исторически замый заслуженный и до сих пор один из самых полярных способов: составить текстовое описание типичного сценария варианта использования. Рассмотрим следующий ниже текст в качестве примера.

- **Увольнение по собственному желанию**
  1. **Сотрудник пишет заявление**
  2. **Начальник подписывает заявление**
  3. **Если есть неиспользованный отпуск,**
     **то бухгалтерия рассчитывает компенсацию**
  4. **Бухгалтерия рассчитывает выходное пособие**
  5. **Системный администратор удаляет учетную запись**
  6. **Менеджер штатного расписания обновляет базу данных**

Казалось бы что здесь неясного? А неясно, например, во что: как должна вести себя система, если на шаге 2 начальник НЕ подписывает заявление. Из текста сценария не только не ясен ответ, но, хуже того, при невнимательном чтении можно и не заметить, что есть вопрос.

Текстовые описания сценариев всем хороши: просты, всем понятны, легко и быстро составляются. Плохи они тем, что могут быть неполны и неточны, и эти недостатки незаметны.

### 2.3.2. Реализация программой на псевдокоде

Повторим еще раз: вариант использования — это описание множества последовательностей действий, доставляющих значимый для действующего лица результат. Наиболее часто используемый метод описания множества последовательностей действий состоит в указании алгоритма, выполнение которого доставляет последовательность действий из требуемого множества.[34] Чаще всего алгоритмы указывают с помощью программ — текстов на некотором языке.

Если программа предназначена для выполнения компьютером, то она должна быть записана на сугубо формальном (и на сегодняшний день довольно примитивном) языке, который называют в этом случае *языком программирования*. Если же программа предназначена исключительно для чтения и, может быть, выполнения человеком, то можно применить менее формальный (и более удобный) язык, который в этом случае обычно называют псевдокодом. Обычно в псевдокод включают смесь общеизвестных ключевых слов языков программирования и неформальные выражения на естественном языке, обозначающие выполняемые действия. Эти выражения должны быть понятны человеку, который пишет (или читает) программу на псевдокоде, но совсем не обязаны быть допустимыми выражениями языка программирования. Текст на псевдокоде *похож* на код программы на языке программирования, но таковым не является — отсюда и название.

---

[34] Последовательность действий при конкретном выполнении алгоритмы называется протоколом этого выполнения. Таким образом, экземпляр варианта использования — сценарий — можно рассматривать как протокол выполнения алгоритма варианта использования.

**Метод пошагового уточнения**

> *Метод пошагового уточнения* при разработке программ был предложен Н. Виртом в период развития структурного программирования. Суть состоит в том, что на первом шаге выписывается (очень короткая) программа на псевдокоде, содержащая обозначения для крупных кусков общего алгоритма. Затем каждое из этих обозначений раскрывается (уточняется) с помощью своего текста на псевдокоде, возможно с введением новых обозначений для более мелких частей алгоритма. Затем вновь введенные обозначения в свою очередь раскрываются и так далее, пока процесс уточнения не дойдет до уровня исполнимого языка программирования.
>
> Метод пошагового уточнения предполагает строгое следование дисциплине структурного программирования и хорошо работает при проектировании сверху вниз. С его помощью получаются понятные и хорошо документированные программы. При разработке программы для отдельного действительно сложного алгоритма этот метод является, видимо, одним из наилучших среди известных методов конструирования программ. Однако метод пошагового уточнения напрямую никак не связан с парадигмой объектно-ориентированного программирования. Современный стиль программирования имеет тенденцию к конструированию программ как очень сложных систем классов, каждый из которых имеет очень простые (с алгоритмической точки зрения) операции. Таким образом, область применимости метода пошагового уточнения сужается.

Первый способ реализации варианта использования — записать алгоритм на псевдокоде. Этот способ хорош тем, что понятен, привычен и доступен любому. Однако в настоящее время вряд ли можно рекомендовать такой способ реализации, как основной, по следующим причинам.

- Реализация на псевдокоде плохо согласуется с современной парадигмой объектно-ориентированного программирования.
- При использовании псевдокода теряются все преимущества использования UML: наглядная визуализация с помощью картинки, строгость и точность языка проектирования и реализации, поддержка распространенными инструментальными средствами.
- Решения на псевдокоде практически невозможно использовать повторно.

Тем не менее, рассмотрим пример реализации вариантов использования на псевдокоде (см. рис. 2.10).

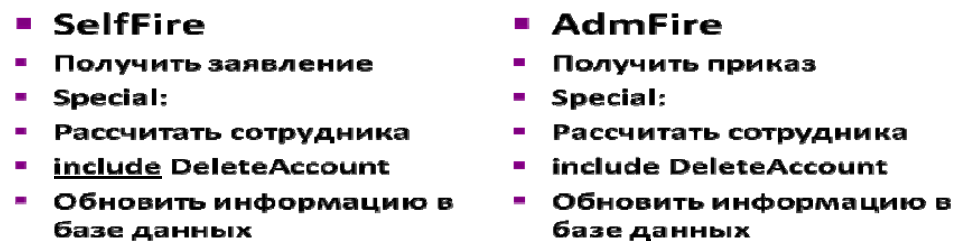



**Рис. 2.10. Реализация вариантов использования при помощи программы на псевдокоде**

Увольнение по собственному желанию запускается по инициативе сотрудника. Увольнение по инициативе администрации начинается с приказа об увольнении.

В этих текстах использовано ключевое слово **`include`**, отражающее наличие зависимостей с таким стереотипом в модели. А именно, это означает, что в этом месте в текст псевдокода для данного варианта использования нужно включить текст псевдокода для варианта использования `DeleteAccount`.

Вариант использования `PayCompensation` запускается, если есть условия для выплаты компенсаций. При этом основные варианты использования не должны знать, каковы эти условия и как рассчитывается компенсация — за это отвечает вариант использования `PayCompensation`. Зависимость со стереотипом **`«extend»`** означает, что псевдокод варианта использования `PayCompensation` должен быть включен в текст основных вариантов использования. При этом вариант использования `PayCompensation` должен знать, в какое место ему нужно включиться. Для этого в основных вариантах использования определена точка расширения — а по сути просто метка в программе.[35]

Мы надеемся, что этот пример в достаточной мере объясняет сходство и различие между зависимостями со стереотипом `include` и `extend`. В обоих случаях речь о включении текста одной программы внутрь текста другой программы. Различие состоит в том, где хранится информация о включении.

В случае стереотипа «`include`» включающая программа знает имя включаемой, а включаемая программа не обязана знать, что ее куда-то включают. В случае стереотипа «`extend`» наоборот, включаемая программа знает имя включающей, а включающая программа не обязана знать, что в нее что-то включают. Поскольку включающая программа знает свою структуру, то при использовании стереотипа «`include`» ей не нужно никакой дополнительной информации о месте включения. Напротив, поскольку включаемая программа не знает структуры включающей программы, то при использовании стереотипа «`extend`» включаемой программе нужна информация о месте включения — имя точки расширения, то есть метки в тексте включающей программы.

---

[35] Для зависимости со стереотипом `include` никакой метки не нужно: место включения определяется тем, где стоит ключевое слово **`include`**.

### *2.3.3. Реализация диаграммами деятельности*

Третий способ реализации варианта использования — описать алгоритм с помощью диаграммы деятельности. С одной стороны, диаграмма деятельности — это полноценная диаграмма UML, с другой стороны, диаграмма деятельности немногим отличается от блок-схемы (а тем самым и от псевдокода). Таким образом, реализация варианта использования диаграммой деятельности является компромиссным способом ведения разработки — в сущности, это проектирование сверху вниз в терминах и обозначениях UML.

Например, в информационной системе отдела кадров прием сотрудника может быть организован так, как показано на рис. 2.11.

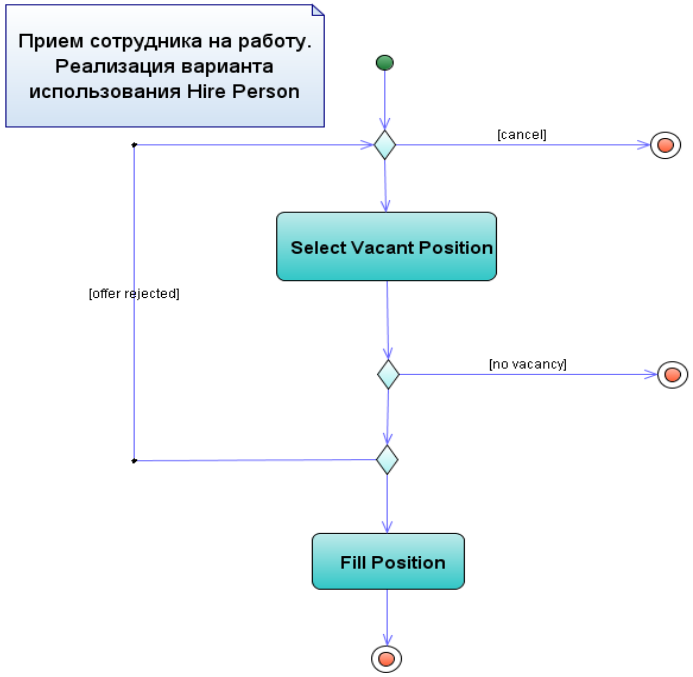



**Рис. 2.11. Реализация варианта использования приема сотрудника информационной системы отдела кадров при помощи диаграммы деятельности**

Приблизило ли нас появление этой диаграммы в модели к завершению работы над системой? С одной стороны, вместо одной сущности, подлежащей реализации (вариант использования `HirePerson`) появилось пять новых: три сторожевых условия и две деятельности, которые в свою очередь теперь нуждаются в реализации. С другой стороны, каждая из этих новых сущностей кажется более простой и понятной, а значит быстрее и надежнее реализуемой. Кроме того, эту диаграмму можно показать заказчику, чтобы проверить, действительно ли проектируемая нами логика работы системы соответствует тому бизнес-процессу, который существует в реальности.

Применение диаграмм активности для реализации вариантов использования не слишком приближает к появлению целевого артефакта — программного кода,[36]

[36] Большинство инструментов не генерируют никакого кода по диаграммам активности.

однако может привести к более глубокому пониманию существа задачи и даже открыть неожиданные возможности улучшения приложения, которые было трудно усмотреть в первоначальной постановке задачи.

Например, рассматривая (чисто формально) схему процесса на рис. 2.11, мы видим что процесс может иметь три исхода.

- Нормальное завершение, которое предполагает обязательное выполнение деятельности `FillPosition`. Резонно предположить, что при выполнении этой деятельности в базу данных будет записана какая-то информация, что, несомненно, является значимым для пользователя результатом.
- Исключительная ситуация, возникающая в том случае, когда процесс не может быть нормально завершен (мы хотели принять человека на работу, и он был согласен, а текущее состояние штатного расписание не позволило этого сделать). Факт возникновения такой ситуации, хотя формально и не является значимым результатом для менеджера персонала, на самом деле может быть весьма важен для высшего руководства или менеджера штатного расписания.
- Завершение процесса без достижения какого-либо видимого результата. Например, менеджер персонала сделал кандидату какие-то предложения, но ни одно из них кандидата не устроило, после чего они расстались, как будто ничего и не было.

Этот простой анализ наталкивает на следующие соображения. Вариант использования *должен* доставлять значимый результат, значит, если результата нет, то что-то спроектировано не так, как нужно. Действительно, все известные нам практические информационные системы отдела кадров обязательно накапливают статистическую информацию о всех *проведенных* кадровых операциях. Такая статистика совершенно необходима для так называемого анализа движения кадров — важной составляющей процесса управления организацией. Однако далеко не все системы позволяют учитывать и *не проведенные* операции. Между тем, учет этой информации, например, учет причин, по которым кандидаты не принимают предложенной работы или, наоборот, причин, по которым организации отвергают кандидатов, может быть весьма полезен для исследования рынка труда и формирования кадровой политики организации.

Немного подумав над этой проблемой (или посоветовавшись с заказчиком), мы можем усовершенствовать нашу диаграмму, например так, как показано на рис. 2.12.

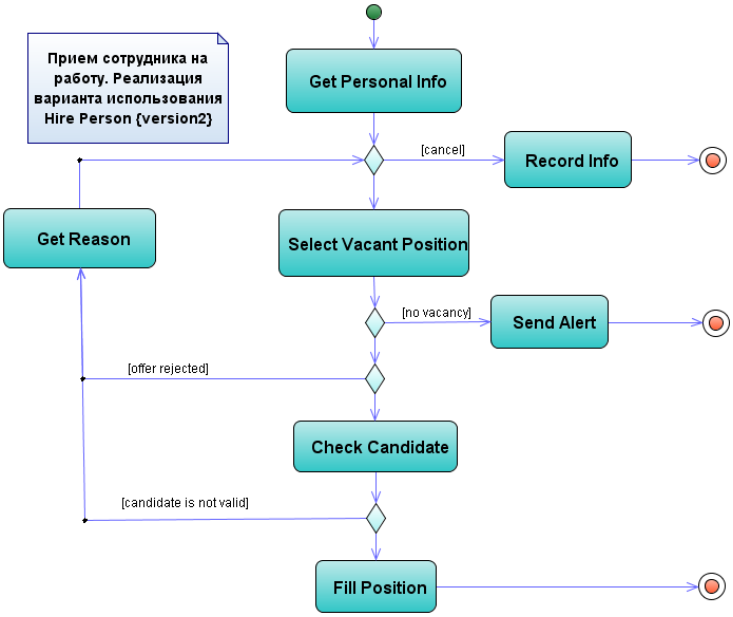



**Рис. 2.12. Усовершенствованная реализация варианта использования приема сотрудника информационной системы отдела кадров**

Вот теперь (формально) все хорошо: информация не теряется. Более того, имеет смысл вернуться к представлению использования и посмотреть, не нужно ли включить в модель новые варианты использования (может быть, как низкоприоритетные и подлежащие реализации в последующих версиях системы). Так реализация вариантов использования может приводить к изменению и усовершенствованию самих вариантов использования. Моделирование имеет итеративный характер, о чем мы уже говорили в главе 1.

### *2.3.4. Реализация диаграммами взаимодействия*

Четвертый из основных способов реализации варианта использования — создать диаграмму взаимодействия (в форме диаграммы кооперации или диаграммы последовательности), которая описывает сценарий данного варианта использования. Этот способ в наибольшей степени соответствует идеологии UML и рекомендуется авторами языка как основной и предпочтительный.

Рассмотрим основные достоинства и недостатки реализации варианта использования диаграммой взаимодействия. Начнем с положительного.

- На диаграмме взаимодействия представлено взаимодействие объектов, т. е. экземпляров некоторых классов. Тем самым построение такой диаграммы с необходимостью приводит к выявлению некоторых классов, которые должны существовать в модели и некоторых операций этих классов (а именно тех, которые появятся в форме сообщений типа вызова операции на диаграмме взаимодействия). Более того, поскольку сообщения передаются вдоль связей (экземпляров ассоциаций), оказываются выявленными и некоторые необходимые ассоциации. Таким образом, реализация варианта использования диаграммой взаимодействия обеспечивает органичный переход от

моделирования использования к моделированию структуры (глава 4) и поведения (глава 5).[37]

- При составлении диаграмм взаимодействия для нескольких вариантов использования могут быть задействованы одни и те же классы, играющие разные роли в различных кооперациях (см. главу 5). Одинаковое поведение и структура, проявляющиеся в разных вариантах использования, оказываются реализованными одним классом. Такой стиль моделирования полностью соответствует объектно-ориентированному подходу и обеспечивает концептуальную целостность проектируемой системы.[38]
- При реализации вариантов использования диаграммами взаимодействия на ранних этапах моделирования появляются сопутствующие фрагменты диаграмм классов (и, может быть, диаграмм реализации). Эти диаграммы гораздо ближе к целевому артефакту — программному коду — нежели диаграммы использования и даже диаграммы деятельности. Все инструменты поддерживают генерацию кода по диаграммам классов, а некоторые — и по диаграммам взаимодействия. Таким образом, появляется возможность автоматизированного построения прототипов (версий системы, обеспечивающих функциональность частично), что полностью соответствует идеологии инкрементальной разработки.

Однако у диаграмм взаимодействия UML есть существенное ограничение. В целом эти диаграммы формулируются на уровне объектов, а потому позволяют реализовать только отдельный сценарий (экземпляр варианта использования). Другими словами, диаграммы взаимодействия позволяют описать протокол выполнения алгоритма, но не сам алгоритм, они не являются универсальной моделью вычислимости. Возникает вопрос: насколько существенным и обременительным оказывается это ограничение при практическом моделировании?

Если алгоритм варианта использования линеен (просто последовательность действий, без ветвлений и циклов), то проблемы нет — линейные алгоритмы суть протоколы выполнения. Для ветвлений и циклов диаграммы взаимодействия содержат некоторые (весьма ограниченные, см. главу 5) средства моделирования. Иногда этих средств может оказаться достаточно.

Что же делать, если все-таки никак не удается построить исчерпывающую диаграмму взаимодействия для варианта использования? В этом случае рекомендуется применять следующие методы. Во-первых, реализовать вариант использования несколькими диаграммами взаимодействия. Каждая диаграмма описывает отдельный сценарий, а все вместе они дают достаточное представление об алгоритме варианта использования. Во-вторых, можно пересмотреть представление использования таким образом, чтобы исключить трудно реализуемые варианты использования. Например, с помощью обобщения выделить

---

[37] Отметим, что в диаграмме деятельности это не так: вы можете использовать в качестве действий операции классов, но не обязаны этого делать. Большинство инструментов даже не поддерживают применения на диаграммах активности элементов модели, определенных в других диаграммах.

[38] Отметим, что применение диаграмм деятельности не обеспечивает концептуальной целостности. Диаграммы деятельности никак не связаны друг с другом. Некоторые инструменты, например, Visio, даже не поддерживают использование деятельности, определенной на одной диаграмме, в другой диаграмме.

частные случаи вариантов использования, которые описывают более однородные множества сценариев и легче реализуются диаграммами взаимодействия. Обычно итеративно применяет оба приема, пока не будет достигнут удовлетворительный результат.

Попробуем проиллюстрировать сказанное на примере реализации диаграммами взаимодействия варианта использования прием сотрудника на работу информационной системы отдела кадров.

Сначала рассмотрим типовой сценарий, когда прием проходит безо всяких осложнений: есть вакантное рабочее место и кандидат готов его занять. Диаграмма последовательности для такого сценария приведена на рис. 2.13.

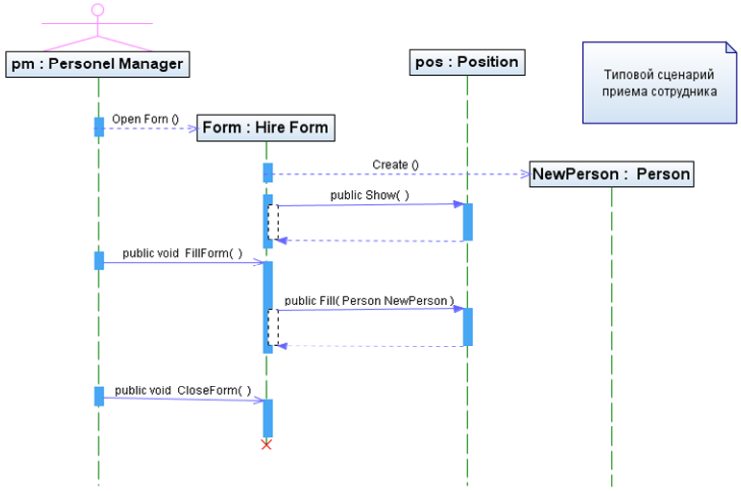



**Рис. 2.13. Диаграмма последовательности для типового сценария приема сотрудника информационной системы отдела кадров**

На приведенной диаграмме (рис. 2.13) последовательность посылаемых сообщений примерно соответствует последовательности действий на диаграмме деятельности (см. рис. 2.12) в том случае, когда поток управления проходит по диаграмме сверху вниз один раз. Таким образом, диаграмма 2.13 до некоторой степени определяет типовой сценарий варианта использования `HirePerson`. Однако, помимо определения последовательности выполняемых действий диаграмма 2.13 содержит и другую информацию, существенную для дальнейшего проектирования. А именно, построив такую диаграмму, мы постулировали существование в системе некоторых классов (возможно, еще не всех), объекты которых должны взаимодействовать для обеспечения требуемого поведения моделируемого варианта использования. Действующее лицо `PersonnelManager` уже было определено при моделировании использования, здесь же в нашей модели появились новые сущности:

- класс `HireForm`, ответственный за интерфейс, необходимый для выполнения варианта использования прием сотрудника;
- класс `Person`, ответственный за хранение данных о конкретном человеке;
- класс `Position`, ответственный за хранение данных и выполнение операций с конкретной должностью.

На этой стадии моделирования список операций указанных классов еще далеко не полон (а атрибуты пока совсем отсутствуют), однако сам факт появления классов в модели является важным архитектурным решением, существенно приближающим нас к реализации системы.

**ЗАМЕЧАНИЕ**

> Большинство инструментов поддерживают следующий режим работы. Составляя некоторую диаграмму, (например, в нашем случае диаграмму последовательности), можно определить в модели некоторые сущности (например, классы), нужные для моделирования в данный момент, не отражая их пока что ни на какой диаграмме. Утрируя, можно представить себе такой стиль моделирования, когда никакие диаграммы вообще не составляются, а, пользуясь средствами инструмента, в модель последовательно добавляются сущности и отношения между ними. Конечно, такой стиль игнорирует один из важнейших аспектов UML — наглядность модели — и на практике не применяется. Однако, приводя это замечание, мы хотим еще раз подчеркнуть важнейшее обстоятельство, часто ускользающее от начинающих пользователей UML: модель не является простой суммой диаграмм, наоборот, каждая диаграмма является не более чем ограниченной проекцией независимо существующей модели.

Возвращаясь к нашему примеру, заметим, что диаграмма 2.13 семантически не полна: она не отражает все сценарии варианта использования, которые предусматриваются, например, диаграммой 2.12. Как уже было сказано, в этом случае можно составить дополнительные диаграммы взаимодействия, реализующие альтернативные сценарии варианта использования. Например, на рис. 2.14 показан сценарий приема сотрудника, соответствующий исключительной ситуации, когда нет вакантных должностей. На этот раз мы описываем сценарий в форме диаграммы кооперации (коммуникации).

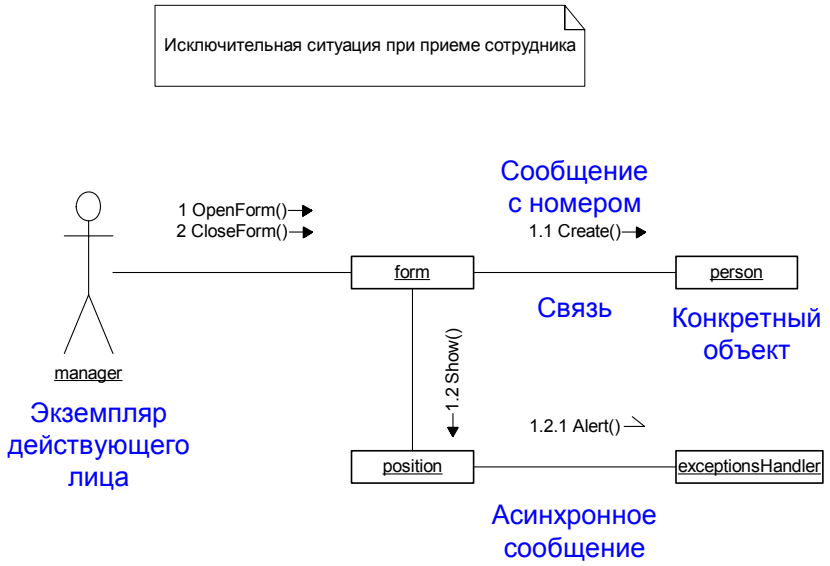



**Рис. 2.14. Диаграмма кооперации для исключительной ситуации при приеме сотрудника информационной системы отдела кадров**

Построение этой диаграммы выявило необходимость включения в модель (по крайней мере) еще одного класса — `ExceptionsHandler`, который несет ответственность за обработку исключительных ситуаций. Конечно, тот способ решения проблемы, который мы предлагаем на диаграмме 2.14 далеко не единственный, и, может быть, не самый лучший из известных вам. Дело не в конкретном способе обработке исключений — это вопрос технологии программирования, а в том, что построение диаграммы взаимодействия выявило сам факт необходимости предусмотреть обработку исключительных ситуаций в системе. Если бы мы не построили диаграмму взаимодействия для альтернативного сценария, мы могли бы упустить из виду это обстоятельство и, тем самым, совершить грубую проектную ошибку.

Авторы UML рекомендуют на этапе перехода от моделирования использования к более детальному проектированию строить диаграммы взаимодействия для всех сценариев вариантов использования (на крайний случай для всех вариантов использования, выбранных для реализации в первой версии системы). Эта рекомендация нам представляется очень полезной и мы советуем ей следовать. Более того, мы можем предложить неформальный критерий для управления процессом разработки, основанный на этой рекомендации. После того, как диаграммы взаимодействия построены, нужно сопоставить набор выявленных классов с неформальным словарем предметной области (см. раздел 2.1.2). Если наблюдается значительное совпадение, скажем, если большинство понятий словаря предметной области оказалось среди классов, выявленных при реализации вариантов использования, то это означает, что мы на правильном пути и можно смело переходить к более детальному проектированию. Если же пересечение словаря и множества выделенных классов незначительно, скажем, если оно составляет менее половины объема словаря, то нужно приостановить проектирование и вернуться на фазу анализа (привлечь сторонних экспертов в предметной области, заново выполнить моделирование использования, проконсультироваться с заказчиком и т. д.).

Из материала этого раздела мы делаем следующий вывод: реализация вариантов использования диаграммами взаимодействия является наиболее трудоемким и сложным методом, но этот метод лучше всего согласован с объектно-ориентированным подходом и в наибольшей мере приближает нас к конечной цели.

## 2.4. Выводы

- Составление диаграмм использования — это первый шаг моделирования.
- Основное назначение диаграммы использования — показать, что делает система во внешнем мире.
- Диаграмма использования не зависит от программной реализации системы и поэтому не обязана соответствовать структуре классов, модулей и компонентов системы.
- Идентификация действующих лиц и вариантов использования — ключ к дальнейшему проектированию.
- В зависимости от выбранной парадигмы проектирования и программирования применяются различные способы реализации вариантов использования.

# Тема 3. Моделирование структуры

- Что такое моделирование структуры?
- В чем заключаются особенности объектно-ориентированного моделирования структуры?
- Какие элементы применяются для моделирования структуры?
- Как и для чего создаются диаграммы классов и объектов?
- В каких случаях применяются диаграммы компонентов и размещения?

## 3.1. Объектно-ориентированное моделирование структуры

Моделируя структуру, мы описываем составные части системы и отношения между ними. UML является объектно-ориентированным языком моделирования, поэтому не удивительно, что основным видом составных частей, из которых состоит система, являются объекты. В каждый конкретный момент функционирования системы можно указать конечный набор конкретных объектов и связей между ними, образующих систему. Однако в процессе работы этот набор не остается неизменным: объекты создаются и уничтожаются, связи устанавливаются и теряются. Число возможных вариантов наборов объектов и связей, которые могут иметь место в процессе функционирования системы, если и не бесконечно, то может быть необозримо велико. Представить их все в модели практически невозможно, а главное бессмысленно, поскольку такая модель из-за своего объема будет недоступна для понимания человеком, а значит бесполезна при разработке системы. Каким же образом можно строить компактные (полезные) модели необозримых (потенциально бесконечных) систем?

Метод построения конечных (и небольших) моделей бесконечных (или очень больших) систем известен человечеству испокон веков и пронизывает всё современное знание в разных формах и под разными названиями. Мы уверены, что читатель легко сможет самостоятельно привести тому множество примеров из знакомых ему областей знания. Заметьте, что в подобных случаях кроме самого компактного описания, подразумевается известным набор правил интерпретации описания, позволяющих построить по описанию множества любой его элемент. Сами правила и способ их задания различны в разных случаях, но принцип один и тот же.

В UML этот принцип формализован в виде понятия *дескриптора*. Дескриптор имеет две стороны: это само описание множества (intent) и множество значений, описываемых дескриптором (extent). Антонимом для дескриптора является понятие *литерала*. Литерал описывает сам себя. Например, тип данных `integer` является дескриптором: он описывает множество целых чисел, потенциально бесконечное (или конечное, но достаточно большое, если речь идет о машинной арифметике). Изображение числа `1` описывает само число "один" и более ничего — это литерал. Почти все элементы моделей UML являются дескрипторами — именно поэтому средствами UML удается создавать представительные модели достаточно сложных систем. Рассмотренные в предыдущей главе варианты использования и действующие лица — дескрипторы, рассматриваемые в этой главе классы, ассоциации, компоненты, узлы — также дескрипторы. Примечание же является литералом — оно описывает само себя.

Эта глава посвящена объяснению немногочисленных, но достаточно содержательных дескрипторов, которые в UML применяются для моделирования структуры.

### *3.1.1. Объектно-ориентированное программирование*

Мы считаем необходимым предпослать дальнейшему рассмотрению набор замечаний по поводу парадигмы объектно-ориентированного программирования в целом. Наши замечания не претендуют на полноту исчерпывающего изложения — основные идеи объектно-ориентированного подхода считаются известными читателю. Тем не менее, наши довольно общие замечания, хотя и не являются истиной в последней инстанции, могут, как нам кажется быть полезными при обсуждении объектно-ориентированного моделирования на UML.

Начнем с небольшого исторического обзора.

Вопрос повышения (изначально очень низкой) продуктивности программирования является предметом технологии программирования. Технология программирования находится в центре внимания ведущих практических программистов, теоретиков и начальников от программирования начиная с конца 60-х годов, со времен так называемой первой революции в программировании. Хотя ресурсы, инвестированные в технологию программирования, огромны и достижения значительны, вопрос далек от окончательного разрешения. Подтверждением тому служит разнообразие парадигм программирования, которые постоянно возникают, не вытесняя друг друга.

| |
|---|
| *Парадигма программирования* — это собрание основополагающих принципов, которые служат методической основой конкретных технологий и инструментальных средств программирования. |

Исторически первой оформленной парадигмой принято считать так называемое *структурное программирование*. Выявленная[39] Э. Дейкстрой обратная зависимость между долей неограниченных операторов перехода `Go To` и качеством (надежностью, скоростью отладки) программы привела к бурному развитию концепции структурного программирования, которое большинством рядовых программистов было понято как "программирование без `Go To`".

*Программирование без* `Go To` основано на том простом факте, что любая структура управления может быть функционально эквивалентно выражена суперпозицией последовательного выполнения, ветвления по условию и цикла с предусловием (рис. 3.1).

[39] Примечательно, что эта зависимость была выявлена не спекулятивным, а правильным научным методом, а именно, статистической обработкой представительной выборки текстов реальных программ.

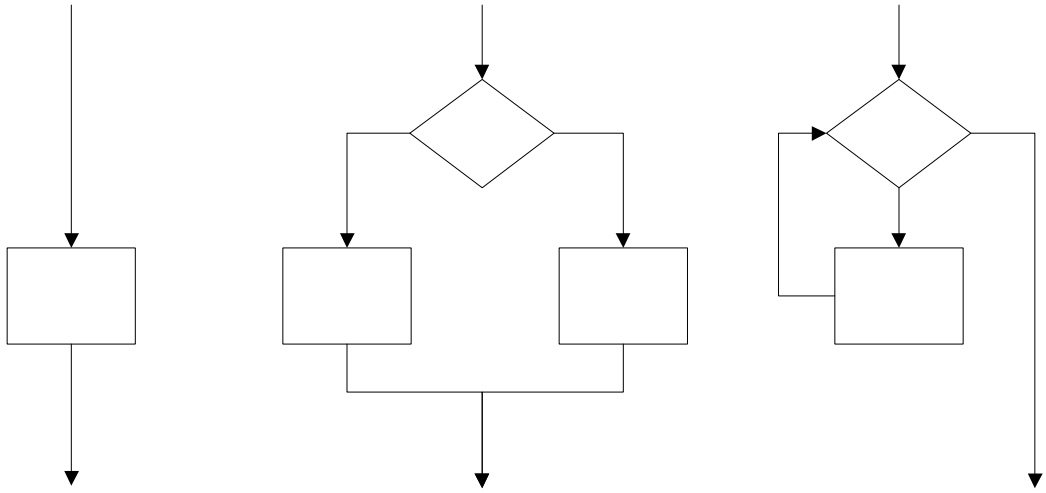


**Рис. 3.1. Базовые структуры управления программирования без Go To.**

Каждая из базовых структур обладает свойством один вход — один выход; этой свойство сохраняется при суперпозиции двух базовых структур; следовательно, свойством один вход — один выход обладает любая суперпозиция и, таким образом, любая структура обладает таким свойством. Наличие свойства один вход — один выход ценно тем, что позволяет установить простое соответствие между статическим текстом программы и динамическим протоколом ее выполнения. Для линейных программ взаимно-однозначное соответствие очевидно; для линейно ветвящихся программ очевидно однозначное соответствие из текста в протокол, для циклических программ соответствие устанавливается, если добавить к естественной координате в тексте (от начала к концу) еще по одной целочисленной координате (обычно называемой счетчиком цикла) для каждого уровня вложенности циклов. Таким образом, программирование без `Go To` позволяет разрешить (до некоторой степени) основное противоречие программирования — между статическим конечным текстом программы и динамическим потенциально бесконечным процессом ее выполнения. Действительно, с одной стороны, в распоряжении программиста находится код программы, который существенно конечен и существенно статичен и на который программист может воздействовать произвольным образом. С другой стороны, целью программирования является получение разворачивающегося во времени процесса выполнения программы, причем этот процесс потенциально бесконечен (или необозримо велик), существенно динамичен и не подлежит прямому управлению со стороны программиста. Таким образом, программирование по существу является процессом косвенного отложенного управления (планирования), причем цепочки прямых и обратных связей очень длинны и запутаны в современных программно-аппаратных компьютерных системах (программа обращается к функциям системы программирования, которые обращаются к функциям административной системы времени выполнения, которые обращаются к функциям операционной системы, которые обращаются к функциям аппаратуры и обратно в том же или даже более сложном порядке). Коротко говоря, программист работает с конечным статическим текстом программы, а представляющим интерес результатом является бесконечный динамический процесс выполнения, и

установление прямого соответствия между этими сущностями является основной проблемой программирования.[40]

**ЗАМЕЧАНИЕ**

> Различного рода синтаксический сахар (Switch, Repeat, For Next, For Each и т. д.) не меняет сути дела, потому что принцип один вход — один выход сохраняется. Поэтому наличие или отсутствие структур управления в языке почти не влияет на программирование. Наличие или отсутствие в языке ограниченных операторов перехода (break, exit, signal и т. д.) также не меняет сути дела по той же причине. Ограниченные переходы полезны, потому что позволяют писать более короткие и эффективные программы той же степени структурности.[41]

Структурное программирование было первым, но не осталось единственным. Возникли и продолжают развиваться другие парадигмы, инспирированные иными моделями вычислимости, нежели машина Тьюринга и иными архитектурами компьютеров, нежели архитектура фон Неймана.

## Компьютерные архитектуры

> Было предложено и реализовано "в металле" множество принципиально различных способов организации работы компьютеров, которые обычно называют архитектурой компьютера. Исторически одной из первых и в то же время до сих пор наиболее распространенной архитектурой компьютеров является *архитектура фон Неймана*. В этой архитектуре компьютер имеет (линейную) адресуемую память, состоящую из ячеек (как правило, одинаковых), и процессор, дискретно выполняющий команды. В памяти хранятся как обрабатываемые данные, так и выполняемые команды. Команды оперируют с ячейками памяти (произвольными), указывая их адреса.

Из соображений эффективности выполнения программ средства программирования ориентируются на подразумеваемую архитектуру. Во всех наиболее распространенных системах программирования имеются два фундаментальных понятия, индуцированные архитектурой машины фон Неймана — это понятие переменной, которая имеет имя и может менять значение (абстракция адресуемой памяти ячеек) и понятие управления (абстракция дискретного последовательного процессора), то есть определяемой программистом последовательности выполнения команд программы. Программирование, в котором используются оба этих ключевых понятия, называется *процедурным* программированием.[42] Программирование, в котором одно или оба понятия не используется, называется *непроцедурным*. Программирование, в котором используется явное понятие управления, называется *императивным*, в противоположность *декларативному* программированию.

---

[40] Приведенный абзац является кратким упрощенным пересказом некоторых положений знаменитого письма «GO TO statements considered harmful», которое написал в начале 60-х годов Э. Дейкстра редактору журнала Datamation и которое положило начало технологии программирования, как самостоятельному направлению программистской мысли.

[41] Это наглядно продемонстрировано в известной статье Д. Кнута «Structured programming with Go To».

[42] Характеристическим признаком процедурного программирования является наличие явного оператора присваивания, а не понятие процедуры, как можно было бы подумать.

В силу целого ряда объективных и субъективных (в основном, исторических) причин непроцедурные парадигмы программирования пока сравнительно редко применяются при решении интересующих нас задач разработки приложений для бизнеса и наиболее популярной в данный момент является парадигма объектно-ориентированного программирования, на использование которой ориентирован UML.

Объектно-ориентированное программирование является консервативным расширением структурного программирования, основанным на следующей идее. Программирование без `Go To` (вкупе с необходимыми расширениями, такими как структурные переходы, обработка исключительных ситуаций и модульность) удовлетворительным образом решает проблему структурирования управления. Однако, как известно, процедурные программы определяются связями не только по управлению, но и по данным. Неограниченное присваивание столь же чревато ошибками, как и неограниченный оператор `Go To`, потому что порождает те же проблемы: трудности динамического отслеживания истории переменной и количественные трудности отождествления переменных (их получается слишком много ввиду сугубой бедности структур данных в традиционных языках программирования). Интуитивно очевидно, что решение должно обладать тем же ключевым свойством локальности: для каждой переменной существует единственный модуль, имеющий к ней доступ. Локализация переменной в модуле не является решением, поскольку процедурное программирование не может обходиться без переменных, время жизни которых выходит за пределы времени активизации процедур их обработки, т. е. нужны переменные, доступные процедуре, которые существуют не только во время вызова этой процедуры. Глобальные же переменные допускают неограниченное присваивание. Паллиативные приемы, подобные описателю `own` в Алголе и именованным общим блокам в Фортране оказались неудобными и ненадежными решениями.

Центральной идеей парадигмы объектно-ориентированного программирования является *инкапсуляция*, т. е. структурирование программы на структуры особого вида, объединяющего данные и процедуры их обработки, причем внутренние данные структуры не могут быть обработаны иначе, кроме как предусмотренными для этого процедурами. В разных вариациях объектно-ориентированного программирования эту структуру называли по-разному: объект, класс, абстрактный тип данных, модуль, кластер и др. В настоящее время структуру из данных и процедур их обработки, существующую в памяти компьютера во время выполнения программы, чаще всего называют *объектом*, а описание множества однотипных объектов к тексте программы называют *классом*.

| *Объект* – структура из данных и процедур их обработки, существующая в памяти компьютера во время выполнения программы. |
|---|

| *Класс* – описание множества однотипных объектов к тексте программы. |
|---|

За редкими исключениями в современных объектно-ориентированных системах программирования классы являются дескрипторами (см. начало данного параграфа). Каждый такой класс имеет внутреннюю часть, называемую

реализацией (или представлением) и внешнюю часть, называемую интерфейсом. Доступ к реализации возможен только через интерфейс. Обычно в интерфейсе различают *свойства* (которые синтаксически выглядят как переменные) и *методы* (которые синтаксически выглядят как процедуры или функции). Мы будем использовать собирательное название *составляющие*[43] для свойств и методов класса.

Кроме основной идеи инкапсуляции, с объектно-ориентированным программированием принято ассоциировать также понятия наследования и полиморфизма. *Наследование* — это способ структуризации описаний многих классов, который позволяет сократить текст программы, сделав его тем самым более обозримым, а значит более надежным и удобным. Если класс А наследует классу В, то говорят, что класс А является *суперклассом* класса В, а класс В — *подклассом* класса А. Подкласс содержит все составляющие своего суперкласса и удовлетворяет принципу подстановочности (см. разд. 1.4.2). Отношение наследования транзитивно. Полиморфизм — это способ идентификации процедур при вызове. В классических процедурных системах программирования процедуры идентифицируются просто по именам. В объектно-ориентированном программировании конкретный метод идентифицируется не только по имени, но и по объекту, которому метод принадлежит, типам и количеству аргументов (т. е. по полной сигнатуре). Таким образом, сходные по назначению и смыслу, но различающиеся в деталях реализации методы разных классов могут быть названы одинаково.

### Хронология развития объектно-ориентированного программирования

Интересно и поучительно проследить хронологию развития объектно-ориентированного программирования: все перечисленные понятия и идеи известны с 1968 года (СИМУЛА-67), в рафинированном виде были оформлены в теоретических работах к 1978 году (Б. Лисков и многие другие авторы), получили общепризнанное языковое выражение к 1988 году (C++) и стали основным и массовым инструментом для профессиональных программистов к 1998 году.

Из сказанного следует важный вывод: объектная ориентированность является атрибутом стиля программирования, присущего конкретному программисту, а не предопределяется используемым языком или системой программирования. В любой системе программирования, даже в той, где нет никаких специальных средств поддержки объектов и классов, можно создавать объектно-ориентированные программы, используя все преимущества объектно-ориентированного программирования, равно как и в самой современной объектно-ориентированной системе никто не застрахован от безнадежно недисциплинированного манипулирования объектами. Более того, в некоторых случаях слишком автоматизированные средства объектно-ориентированного программирования *мешают* объектно-ориентированному программированию, поскольку *навязывают* программисту конкретные решения. Например, в

[43] Мы не используем термин "член класса", хотя он часто встречается в литературе. По нашему мнению, поскольку класс — это множество объектов, членом класса мыло бы естественно назвать элемент множества, т. е. объект — экземпляр класса, а отнюдь не составную часть описания.

большинстве современных систем объектно-ориентированного программирования требуется, чтобы метод был отнесен в точности к *одному* классу, хотя это отнюдь не всегда естественно и удобно.[44]

С парадигмой объектно-ориентированного программирования связано развитие и практическое воплощение важной мысли (уже упомянутой в разд. 1.2.3.) о неразрывной связи проектирования и кодирования и о примате проектирования. Человек мыслит реальный мир состоящим из объектов.[45] Кодированию *всегда*[46] предшествует проектирование, важнейшей частью которого является моделирование, т. е. вычленение в предметной области задачи объектов и связей, которые существенны для решения задачи. Объектно-ориентированное программирование позволяет установить прямое соответствие между программными конструкциями и объектами реального мира через объекты модели. Таким образом, объектно-ориентированная система программирования поддерживает не только кодирование, но и (до некоторой степени) проектирование. При написания объектно-ориентированной программы основные усилия программист тратит на определение состава классов, их методов и свойств. Эта информация является в чистом виде наложенной структурой, поскольку явно в исполняемом коде программы никак не отражается. Кодирование тел методов (то, что транслируется в конечном счете в машинные команды) в хорошо спроектированной объектно-ориентированной программе оказывается почти тривиальным.

Некоторые экстремисты объектно-ориентированного подхода считают, что понятие класса вообще является единственным понятием, необходимым и достаточным для описания структуры (да и поведения заодно) любых программных систем. Мы не разделяем такой крайней точки зрения, но согласны с тем, что классы занимают первостепенное место в объектно-ориентированном моделировании структуры.

### *3.1.2. Назначение структурного моделирования*

Рассмотрим более детально, какие именно структуры нужно моделировать и зачем. Мы выделяем следующие структуры:

- структура связей между объектами во время выполнения программы;
- структура хранения данных;
- структура программного кода;
- структура компонентов в приложении;
- структура используемых вычислительных ресурсов;
- структура сложных объектов, состоящих из взаимодействующих частей

---

[44] Распространен предрассудок, что только синтаксис вида `Object.Method(Arguments)` является допустимым в объектно-ориентированном программировании. Синтаксис `Method(pObject,Arguments)` ничуть не хуже, а зачастую лучше. Например, если объект `Человек` имеет свойство `Семейное_положение` и метод `Вступить_в_брак(супруг:Человек)`, то обеспечить необходимую целостность данных не просто. Эта проблема исчезает, если имеется служба (без памяти!) `ЗАГС`, у которой есть метод `Регистрация_брака(муж,жена:Человек)`.

[45] Это свойство нашего мышления, но не свойство реального мира.

[46] У неумелых программистов неявно и бессознательно, у умелых – явно и осознанно.

- структура артефактов в проекте.
    - Диаграммы компонентов

Наша классификация может быть не совсем полна и уж совсем не ортогональна (упомянутые структуры не являются независимыми, они связаны друг с другом), но в целом соответствует сложившейся практике разработки приложений, поскольку позволяет фиксировать основные решения, принимаемые в процессе проектирования и реализации. В этом разделе кратко мы обсудим назначение перечисленных структур и укажем средства UML, предназначенные для их моделирования.

*Структура связей между объектами во время выполнения программы*. В парадигме объектно-ориентированного программирования процесс выполнения программы состоит в том, что программные объекты взаимодействуют друг с другом, обмениваясь сообщениями. Наиболее распространенным типом сообщения является вызов метода объекта одного класса из метода объекта другого класса. Для того чтобы вызвать метод объекта, нужно иметь доступ к этому объекту. На уровне программной реализации этот доступ может быть обеспечен самыми разнообразными механизмами. Например, в объекте содержащем вызывающий метод может хранится указатель (ссылка) на объект, содержащий вызываемый метод. Еще вариант: ссылка на объект с вызываемым методом может быть передана в качестве аргумента вызывающему методу. Возможно, используется какой-либо системный механизм удаленного вызова процедур, обеспечивающий доступ к объектам (например, такие как CORBA или DCOM) по идентификаторам. Если свойства объектов представлены записями в таблице базы данных, а методы — хранимыми процедурами СУБД (нередкий вариант реализации), то идентификация объектов осуществляется по первичному ключу таблицы. Как бы то ни было, во всех случаях имеет место следующая ситуация: один объект "знает" другие объекты и, значит, может вызвать открытые методы, использовать и изменять значения открытых свойств и т. д. В этом случае, мы говорим что объекты связаны, т. е. между ними есть *связь*. Для моделирования структуры связей в UML используются отношения ассоциации на диаграмме классов.

### Технологии доступа к объектам

Существуют, активно применяются и постоянно появляются новые технологии, обеспечивающие доступ к объектам в распределенных и гетерогенных (неоднородных) сетях. Одной из самых популярных является технология CORBA, разработанная группой OMG (Object Management Group). Основу этой технологии составляет так называемый брокер запросов к объектам (Object Request Broker, ORB), который управляет взаимодействием клиентов и серверов распределенного приложения. ORB должен быть установлен на каждом компьютере, где исполняются программы, использующие технологию CORBA. ORB реализован для всех основных аппаратных и программных платформ, поэтому распределенные приложения, использующие технологию CORBA, могут выполняться в гетерогенных сетях.

Другим примеров является распределенная компонентная модель объектов DCOM (Distributed Component Object Model), разработанная корпорацией Microsoft. Эта технология является развитием технологии

COM (Component Object Model), определяющей методы создания и взаимодействия программных объектов различных типов. Технология COM отличается от технологии CORBA тем, что взаимодействие между объектами устанавливается непосредственно, а не с помощью промежуточного агента.

*Структура хранения данных*. Программы обрабатывают данные, которые хранятся в памяти компьютера. В парадигме объектно-ориентированного программирования для хранения данных во время выполнения программы предназначены свойства объектов, которые моделируются в UML атрибутами классов. Однако большая часть приложений для автоматизации делопроизводства устроена так, что определенные данные (не все) должны хранится в памяти компьютера не только во время сеанса работы приложения, но постоянно, т. е. между сеансами. Объекты, которые сохраняют (по меньшей мере) значения своих свойств даже после того, как завершился породивший их процесс, мы будем называть *хранимыми*. В UML для моделирования данного свойства объектов и их составляющих применяется стандартное именованное значение `persistence`, которое может быть назначено классификатору, ассоциации или атрибуту и может принимать одно из двух значений:

- `persistent` — экземпляры классификатора, ассоциации или значения атрибута, соответственно, должны быть хранимыми;
- `transient` — противоположно предыдущему — сохранять экземпляры не требуется (значение по умолчанию).

В настоящее время самой распространенным способом хранения объектов является использование системы управления базами данных (СУБД). При этом хранимому классу соответствует таблица базы данных, а хранимый объект (точнее говоря, набор значений хранимых атрибутов) представляется записью в таблице. Вопрос структуры хранения данных является первостепенным для приложений баз данных. К счастью, известны надежные методы решения этого вопроса (см. врезки "Схема базы данных" в разд. 2.1.2 и "Диаграммы сущность-связь" в разд. 2.1). Эти же методы (с точностью до обозначений) применяются и в UML в форме ассоциаций с указанием кратности полюсов.

*Структура программного кода*. Не секрет, что программы существенно отличаются по величине — бывают программы большие и маленькие. Удивительным является то, насколько велики эти различия: от сотен строк кода (и менее) до сотен миллионов строк (и более). Столь большие количественные различия не могут не проявляться и на качественном уровне. Действительно, для маленьких программ структура кода практически не имеет значения, для больших — наоборот, имеет едва ли не решающее значение. Поскольку UML не является языком программирования, модель не определяет структуру кода непосредственно, однако косвенным образом структура модели существенно влияет на структуру кода. Большинство инструментов поддерживает полуавтоматическую генерацию кода для одного или нескольких объектно-ориентированных языков программирования. В большинстве случаев классы модели транслируются в классы (или эквивалентные им конструкции) целевого языка. Кроме того, многие инструменты учитывают структуру пакетов в модели и транслируют ее в соответствующие "надклассовые" структуры целевой системы

программирования. Таким образом, если задействовано средство автоматической генерации кода, то структура классов и пакетов в модели фактически полностью моделирует структуру кода приложения.

*Структура сложных объектов, состоящих из взаимодействующих частей*. Для моделирования этой структуры применяется новое средство UML 2 — диаграмма внутренней структуры классификатора. Мы не рассматриваем такие структуры в нашем сквозном примере детально, а потому ограничимся одним примером, приведенным на рис. 3.2

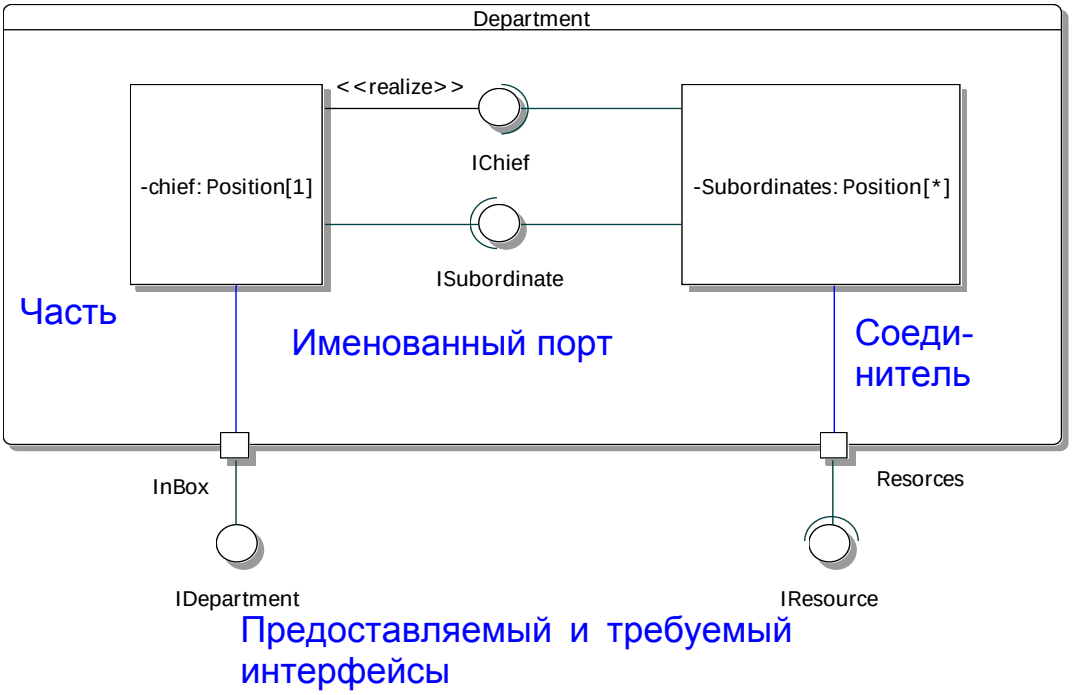



**Рис. 3.2 Диаграмма внутренней структуры классификатора**

На диаграмме внутренней структуры классификатора показывается классификатор, внутри которого имеются *части*. Частей может быть несколько. Каждая часть является экземпляром некоторого друго классификатора. Части могут взаимодействовать друг с другом. Это обозначается с помощью *соединителей*. Если нужно показать взаимодействие с внешним миром, то на кранице структурированного классификатора изображается именованный *порт*.

*Структура артефактов в проекте*. Только самые простые приложения состоят из одного артефакта — исполнимого кода программы. Большинство насчитывает в своем составе десятки, сотни и тысячи различных компонентов: исполнимых двоичных файлов, файлов ресурсов, исходного кода, различных сопровождающих документов, справочных файлов, файлов с данными и т. д. Для большого приложения важно не только иметь точный полный список всех артефактов, но и указать, какие именно компоненты входят в конкретный экземпляр системы. Дело в том, что для больших приложений в проекте сосуществуют разные версии одной и той же компоненты. Это исчерпывающим образом моделируется диаграммами

компонентов UML, где предусмотрены стандартные стереотипы для описания компонентов разных типов.

*Структура компонентов в приложении*. Приложение, состоящее из одной исполнимой компоненты, имеет тривиальную структуру компонентов, моделировать которую нет нужды. Но большинство современных приложений состоят из многих компонентов, даже если и не являются распределенными. Компонентная структура предполагает описание двух аспектов: во-первых, как классы распределены по компонентам, во-вторых, как (через какие интерфейсы) компоненты взаимодействуют друг с другом. Оба эти аспекта моделируются диаграммами компонентов UML.

*Структура используемых вычислительных ресурсов*. Многокомпонентное приложение, как правило, бывает распределенным, т. е. различные компоненты выполняются на разных компьютерах. Диаграммы размещения и компонентов позволяют включить в модель описание и этой структуры.

### *3.1.3. Классификаторы*

Важнейшим типом дескрипторов являются классификаторы. *Классификатор* — это дескриптор множества однотипных объектов. Из этого определения непосредственно вытекает основное и характеристическое свойство классификатора: классификатор (прямо или косвенно) может иметь экземпляры.

В UML используются следующие классификаторы:

- класс (см. разд. 2.2.1 и далее);
- интерфейс (см. разд. 3.2.7);
- тип данных (см. разд. 3.2.8);
- узел (см. разд. 3.3.5);
- компонент (см. разд. 3.3.2);
- действующее лицо (см. разд. 2.2.1);
- вариант использования (см. разд. 2.2.2);
- подсистема (см. разд. 5.2.1).

Все классификаторы имеют некоторые общие свойства, которые мы опишем в этом разделе и будем использовать в дальнейшем.

Во-первых, классификаторы (как и все элементы модели) имеют имена. Имя служит для идентификации элемента модели и потому должно быть уникально в данном пространстве имен (см. разд. 2.2.1.). Классификатор может быть абстрактным (т. е. не могущим иметь прямых экземпляров) и в этом случае его имя выделяется курсивом. Классификатор может быть конкретным (может иметь прямые экземпляры) и в этом случае его имя записывается прямым шрифтом.

Во-вторых, как уже было сказано, классификатор может иметь экземпляры. Экземпляры бывают прямые и косвенные.

Если некоторый объект непосредственно порожден с помощью конструктора классификатора *A*, то этот объект называется *прямым экземпляром A*.

Если классификатор *A* является обобщением классификатора *B* или если классификатор *B* является реализацией классификатора *A*, то все экземпляры классификатора *B* являются *косвенными экземплярами* классификатора *A*.

Данное свойство является транзитивным: если классификатор *B* является обобщением классификатора *C*, то все экземпляры *C* также являются косвенными экземплярами *A*. Именно это обстоятельство позволяет рассматривать абстрактные классификаторы. Действительно, абстрактный классификатор — это такой дескриптор множества объектов, в котором нет прямого описания элементов множества, но данный классификатор связан отношением обобщения с другими классификаторами и объединение множеств их экземпляров считается множеством экземпляров данного абстрактного классификатора. Другими словами, множество определяется не прямо, а через совокупность подмножеств. Например, интерфейс, будучи абстрактным классом, не может иметь непосредственных экземпляров, но реализующий его класс может, стало быть, интерфейс является классификатором.
В-третьих, классификатор (как и другие элементы модели) имеет видимость.

*Видимость* определяет, может ли свойство одного классификатора (в том числе имя) использоваться в другом классификаторе.

Другими словами, если в определенном контексте нечто доступно и может быть как-то использовано, то оно является видимым (в этом контексте). Если же оно не видимо, то и не может быть использовано. Видимость является свойством всех элементов модели (хотя не для всех элементов это свойство является существенным). Видимость может иметь одно из трех значений:

- открытый (обозначается знаком + или ключевым словом `public`);
- защищенный (обозначается знаком # или ключевым словом `protected`);
- закрытый (обозначается знаком – или ключевым словом `private`).

Открытый элемент модели является видимым везде, где является видимым содержащий его контейнер. Например, открытый атрибут класса виден везде, где виден сам класс.
Защищенный элемент модели виден в своем контейнере и во всех контейнерах, для которых данный является обобщением. Например, защищенный атрибут класса виден в этом классе и во всех наследующих классах.
Закрытый элемент модели виден только в своем контейнере. Например, закрытый атрибут класса виден только в этом классе.

**ЗАМЕЧАНИЕ**

> Для видимости в UML нет значения по умолчанию. Например, если для атрибута класса не указано значение видимости, то это не значит, что атрибут по умолчанию открытый или, наоборот, закрытый. Это означает, что видимость для данного атрибута в модели не указана и в этом аспекте модель не полна.

В-четвертых, все составляющие классификатора имеют область действия.

*Область действия* определяет, как проявляет себя составляющая классификатора в экземплярах, т. е. имеют экземпляры свои значения составляющей или совместно используют одно значение.

Область действия имеет два возможных значения:

- экземпляр — никак специально не обозначается, поскольку подразумевается по умолчанию;
- классификатор — описание составляющей классификатора подчеркивается.

Если областью действия составляющей является экземпляр, то каждый экземпляр классификатора имеет свое значение составляющей. Например, областью действия атрибута по умолчанию является экземпляр. Это означает, что каждый объект — экземпляр класса — имеет свое собственное значение атрибута, которое может меняться независимо от значений данного атрибута других объектов, экземпляров этого же класса. Если областью действия составляющей является классификатор, то все экземпляры классификатора совместно используют одно значение составляющей. Например, конструктор обычно имеет областью действия классификатор, поскольку является процедурой, общей для всех создаваемых объектов.

В-пятых, классификатор имеет *кратность*, т. е. ограничение на количество экземпляров.[47] Возможны следующие варианты.

- У классификатора нет экземпляров — такой классификатор называется *службой*. Все составляющие службы имеют областью действия классификатор. Хранение информации, обрабатываемой службой, обеспечивают объекты использующие службу. Типичный пример — набор процедур общего пользования, скажем, библиотека математических функций. Службы используются в приложениях достаточно часто, поэтому есть даже стандартный стереотип (`utility`), определяющий класс как службу.
- Классификатор имеет ровно один экземпляр. Такой классификатор называется *одиночкой*. В сущности, между службой и одиночкой различия незначительны, но иногда одиночку использовать удобнее, например, для хранения глобальных переменных приложения.
- Классификатор имеет фиксированное число экземпляров. Такой вариант не часто, но встречается. Например, порты в концентраторе.
- По умолчанию классификатор имеет произвольное число экземпляров. Поскольку этот вариант встречается чаще всего, он никак специально не указывается.

В-шестых, классификаторы (и только они!) могут участвовать в отношении обобщения *обобщение* (см. разд. 3.2.5)

Сказанное в этом разделе мы закончим диаграммой классов из метамодели UML,[48] описывающей понятие классификатора (рис. 3.3)

---

[47] Кратность может быть не только у классификаторов, но также у атрибутов и полюсов ассоциаций (см. разд. 3.2.1).

[48] Данная и последующие диаграммы с фрагментами метамодели UML не являются заимствованиями из стандарта — эти диаграммы составлены автором. Хотя в стандарте *нет* именно таких диаграмм, какие здесь приводятся, это, тем не менее, диаграммы метамодели UML (с точностью до возможных ошибок автора).

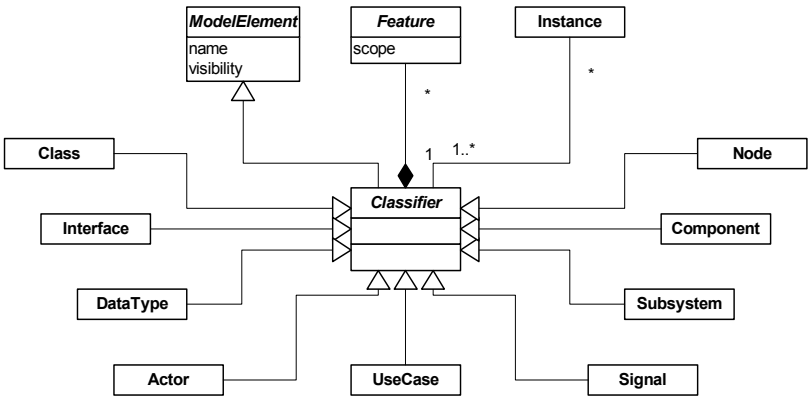



**Рис. 3.3 Метамодель классификатора**

**ЗАМЕЧАНИЕ**

Диаграмма на рис. 3.3 и последующие диаграммы с фрагментами метамодели UML не являются заимствованиями из стандарта — эти диаграммы составлены автором. Хотя в стандарте *нет* именно таких диаграмм, какие здесь приводятся, это, тем не менее, диаграммы метамодели UML (с точностью до возможных ошибок автора). Нам представляется, что возможность пересказывать стандарт "своими словами" не отклоняясь от него — еще один весомый довод в пользу гибкости и выразительности UML.

### *3.1.4. Идентификация классов*

Повторим еще раз: описание классов и отношений между ними является основным средством моделирования структуры в UML. Но прежде чем переходить к технике описания классов полезно обсудить вопрос, как выделяются классы, подлежащие описанию. По этому вопросу в литературе приводится множество соображений, советов, рекомендаций и даже принципов. Само разнообразие подходов свидетельствует о том, что среди них нет универсального и применимого во всех случаях. Мы выбрали три приема выделения классов, самых простых (можно сказать, даже примитивных), а потому, по нашему мнению, самых действенных и широко применимых:

- словарь предметной области;
- реализация вариантов использования;
- образцы проектирования.

*Словарь предметной области* — это набор основных понятий (сущностей) данной предметной области.

Более точного определения мы дать не можем (см. также вставку "Словарь предметной области" в разд. 2.1.2), зато мы можем дать легко выполнимый совет. Рассмотрите внимательно текст технического задания (или иного документа, лежащего в основе проекта) и выделите в содержательной части имена существительные — все они являются кандидатами на то, чтобы быть названиями классов (или атрибутов классов) проектируемой системы. Разумеется, после этой простой операции к полученному списку нужно применить фильтр здравого смысла и опыта, отсекая ненужное.

Рассмотрим пример информационной системы отдела кадров. В тексте технического задания (см. разд. 2.1.1) первый вводный абзац можно отбросить сразу — это общие слова. Суть заключена в нумерованных пунктах. Но в этом тексте вообще все слова являются существительными (кроме союзов). Выпишем их в том порядке, как они встречаются, но без повторений:

- прием;
- перевод;
- увольнение;
- сотрудник;
- создание;
- ликвидация;
- подразделение;
- вакансия;
- сокращение;
- должность.

Заметим, что некоторые их этих слов по сути являются названиями действий (и по форме являются отглагольными существительными). Фактически, это глаголы, замаскированные особенностями родного языка. Это ясно видно, если переписать текст технического задания в форме простых утверждений, как рекомендуется в разд. 2.1.3. Отбросим замаскированные глаголы. Таким образом, остается список из четырех слов:
сотрудник,
подразделение;
вакансия;
должность.

При анализе технического задания в разд. 2.1.3 мы отметили (опираясь на знание предметной области), что вакансия — это должность в особом состоянии. Таким образом, это слово в списке лишнее и у нас остались три кандидата, которые мы оставляем в словаре (и заодно присваиваем им английские идентификаторы).

- Сотрудник (`Person`);
- Подразделение (`Department`);
- Должность (`Position`).

Рассмотрим теперь, как выявляются классы в процессе реализации вариантов использования (см. разд. 2.3). Если при реализации вариантов использования применяются диаграммы взаимодействия (разд. 2.3.3), то в этом процессе в качестве побочного эффекта выделяются некоторые классы непосредственно, поскольку на диаграммах кооперации и последовательности основными сущностями являются объекты, которые по необходимости нужно отнести к определенным классам. Использование диаграмм деятельности (разд. 2.3.2) также может подсказать, какие классы нужно определить в системе, особенно если на диаграмме деятельности указывается поток объектов. Однако, если сценарии вариантов использования описываются на псевдокоде (разд. 2.3.1), то выделить классы значительно труднее. Фактически, если варианты использования реализуются на псевдокоде или диаграммами деятельности вне связи с объектами, то выявление объектной структуры системы просто откладывается "на потом". Иногда это может быть вполне оправдано — например, архитектор,

моделирующий систему, прежде чем начать проектирование основной структуры классов, хочет более глубоко вникнуть в логику бизнес-процессов незнакомой ему предметной области.

Обсудим это на примере информационной системы отдела кадров. Реализация вариантов использования `SelfFire` и `AdmFire` в разделе 2.3.1 не дает практически никакой информации для выделения классов. Появление двух новых существительных ("приказ" и "заявление") наталкивает на мысль, что в системе может появиться класс `Document`, но сразу ясно, что это класс будет пассивным хранилищем информации, не наделенным собственным поведением и это полезное наблюдение никак не приближает к решению основной задачи: выявить ключевые классы в системе. Реализация варианта использования `HirePerson` с помощью диаграмм деятельности в разд. 2.3.2 существенно углубляет наши знания о процессе приема на работу, но никак не приближает нас к структуре классов приложения. Наиболее значительный прогресс в этом направлении дают диаграммы последовательности и взаимодействия для этого же варианта использования, приведенные в разд. 2.3.3. Два класса — `Person` и `Position` — те же, что нам подсказал анализ словаря предметной области. Очевидно, что если одни и те же выводы получены разными способами, доверие к ним возрастает. Полезный класс `ExceptionHandler`, однако, вряд ли мог появиться из словаря: описывая предметную область, люди склонны закрывать глаза на возможные ошибки, исключительные ситуации и прочие неприятности. Особого обсуждения заслуживает появление класса `HireForm`.

На самом деле этот класс появился в модели как результат применения образца проектирования.

> *Образец проектирования* — это стандартное решение типичной задачи в конкретном контексте.

Синтаксически образец в UML является кооперацией, роли объектов в которой рассматриваются как параметры, а аргументами являются конкретные классы модели. В данном случае применен образец `View-Model Separation`, который предполагает, что класс, ответственный за интерфейс с пользователем не должен прямо манипулировать данными, а класс, ответственный за хранение и обработку данных, не должен содержать интерфейсных элементов. Таким образом, класс `HireForm` появился как недостающий аргумент кооперации `View-Model Separation` (мы еще вернемся к обсуждению применения образцов в главе 5).

Подведем итоги: мы полагаем, что классы в модели идентифицируются в результате проведения трех частично независимых процессов: анализа предметной области, согласования уже построенной модели и применения теоретических соображений. Мы уверены, что ни одним из этих методов нельзя пренебрегать, но одновременно утверждаем, что ни один из них не является универсальным и самодостаточным: решающим является здравый смысл и опыт архитектора, составляющего модель.

## 3.2. Диаграммы классов

Диаграмма классов является основным средством моделирования структуры UML. Это совсем не удивительно и вполне естественно, поскольку UML является "сильно" объектно-ориентированным языком. Класс в UML является основной

структурной единицей. Диаграммы классов наиболее информационно насыщены по сравнению с другими типами канонических диаграмм UML, инструменты генерируют код в основном по описанию классов, структура классов точнее всего соответствует окончательной структуре кода приложения.
На диаграммах классов в качестве сущностей применяются прежде всего классы, как в своей наиболее общей форме, так и в форме многочисленных стереотипов и частных случаев: интерфейсы, типы данных, процессы и др. Кроме того, в диаграмме классов могут использоваться (как и везде) пакеты и примечания.
Сущности на диаграммах классов связываются главным образом отношениями ассоциации (в том числе агрегирования и композиции) и обобщения. Отношения зависимости и реализации на диаграммах классов применяются реже.

### *3.2.1. Классы*

Класс — один из самых "богатых" элементов моделирования UML. Описание класса может включать множество различных элементов, и чтобы они не путались, в языке предусмотрено группирование элементов описания класса по *разделам*. Стандартных разделов три:

- раздел имени — наряду с обязательным именем может содержать также стереотип, кратность и список свойств;
- раздел атрибутов — содержит список описаний атрибутов класса;
- раздел операций — содержит список описаний операций класса.

Как и все основные сущности UML, класс обязательно имеет имя, а стало быть раздел имени не может быть опущен. Прочие разделы могут быть пустыми. Наряду со стандартными разделами, описание класса может содержать и произвольное количество дополнительных разделов. Семантически дополнительные разделы эквиваленты примечаниям. Если инструмент умеет что-то делать с информацией в дополнительных разделах, пусть делает. В любом случае инстумент обязан сохранить эту информацию в модели.
Нотация классов очень проста — это всегда прямоугольник. Если разделов более одного, то внутренность прямоугольника делится горизонтальными линиями на части, соответствующие разделам. Содержимым раздела в любом случае является текст.[49] Текст внутри стандартных разделов должен иметь определенный синтаксис.

**ЗАМЕЧАНИЕ**

> Некоторые инструменты допускают использование нескольких альтернативных вариантов синтаксиса для текстов в разделах. Например, синтаксис описания атрибутов в стиле, рекомендованном UML или же в стиле целевого языка программирования данного инструмента. Такие вариации допускаются стандартом при условии, что варианты синтаксиса семантически эквиваленты и могут быть преобразованы друг в друга без потери информации. В данной книге применяется стандартный синтаксис.

Раздел имени класса в общем случае имеет следующий синтаксис.

```
«стереотип» ИМЯ {свойства} кратность
```

[49] Некоторые инструменты позволяют помещать в разделы класса не только тексты, но также фигуры и значки.

В табл. 3.1 перечислены стандартные стереотипы классов.

**Таблица 3.1.** Стандартные стереотипы классов

| Стереотип | Описание |
|---|---|
| `actor` | действующее лицо |
| `enumeration` | перечислимый тип данных |
| `exception` | сигнал, распространяемый по иерархии обобщений |
| `implementation Class` | реализация класса |
| `interface` | нет атрибутов и все операции абстрактные |
| `metaclass` | экземпляры являются классами |
| `powertype` | метакласс, экземплярами которого являются все наследники данного класса |
| `process, thread` | активные классы |
| `signal` | класс, экземплярами которого являются сообщения |
| `stereotype` | стереотип |
| `type (datatype)` | тип данных |
| `utility` | нет экземпляров = служба |

Обязательное имя класса может быть выделено курсивом и в этом случае данный класс является абстрактным, т. е. не могущим иметь непосредственных экземпляров.

**ЗАМЕЧАНИЕ**

> Если имя подчеркнуто, то это уже не имя класса, а имя объекта.

Класс, а также отдельные элементы его описания могут иметь произвольные заданные пользователем ограничения и именованные значения.

Кратность класса задается по общим правилам (см. раздел 3.1.3). Наиболее распространенный случай неограниченной кратности (т. е. класс может иметь произвольное значение экземпляров) подразумевается по умолчанию и никак не отражается на диаграмме классов. Другой распространенный случай — нулевая кратность — обычно представляется с помощью стандартного стереотипа «`utility`» (см. табл. 3.1).

Рассмотрим пример раздела имени класса для нашей информационной системы отдела кадров.

Если мы предполагаем, что проектируемая информационная система отдела кадров будет использоваться на одном предприятии, то целесообразно определить служебный класс `Company` со стереотипом `utility` для хранения глобальных атрибутов и операций информационной системы отдела кадров. Раздел имени такого класса показан на рис. 3.4.


**Рис. 3.4. Раздел имени службы**

## *3.2.2. Атрибуты*

*Атрибут* — это именованное место (или, как говорят, *слот*), в котором может храниться значение.

Атрибуты класса перечисляются в разделе атрибутов. В общем случае описание атрибута имеет следующий синтаксис.

```
видимость ИМЯ кратность : тип = начальное_значение {свойства}
```

Видимость, как обычно, обозначается знаками +, –, #. Еще раз подчеркнем, что если видимость не указана, то никакого значения видимости по умолчанию не подразумевается.

Если имя атрибута подчеркнуто, то это означает, что областью действия данного атрибута является класс, а не экземпляр класса, как обычно. Другими словами, все объекты — экземпляры этого класса совместно используют одно и тоже значение данного атрибута, общее для всех экземпляров. В обычной ситуации (нет подчеркивания) каждый экземпляр класса хранит свое индивидуальное значение атрибута.

**ЗАМЕЧАНИЕ**

Подчеркивание описания атрибута соответствует описателю `static` в языке C++.

Кратность, если она присутствует, определяет данный атрибут как массив (определенной или неопределенной длины).

Тип атрибута — это либо примитивный (встроенный) тип, либо тип определенный пользователем (см. раздел 3.2.8).

Начальное значение имеет очевидный смысл: при создании экземпляра данного класса атрибут получает указанное значение. Заметим, что если начальное значение не указано, то никакого значения по умолчанию не подразумевается. Если нужно, чтобы атрибут обязательно имел значение, то об этом должен позаботиться конструктор класса.

Как и любой другой элемент модели, атрибут может быть наделен дополнительными свойствами в форме ограничений и именованных значений.

У атрибутов имеется еще одно стандартное свойство: *изменяемость*. В табл. 3.2 перечислены возможные значения этого свойства.

**Таблица 3.2.** Значения свойства изменяемости атрибута

| Значение | Описание |
|---|---|
| `changeable` | Никаких ограничений на изменение значения атрибута не накладывается. Данное значение имеет место по умолчанию, поэтому указывать в модели его излишне. |
| `addOnly` | Это значение применимо только к атрибутам, кратность которых больше единицы. При изменении значения атрибута новое значение добавляется в массив значений, но старые значения не меняются и не исчезают. Такой атрибут "помнит" историю своего изменения. |
| `frozen` | Значение атрибута задается при инициализации объекта и не может меняться. |

Например, в информационной системе отдела кадров класс Person, скорее всего, должен иметь атрибут, хранящий имя сотрудника. В табл. 3.3 приведен список примеров описаний такого атрибута. Все описания синтаксически допустимы и могут быть использованы в соответствии с текущим уровнем детализации модели.

**Таблица 3.3.** Примеры описаний атрибутов

| Пример | Пояснение |
|---|---|
| `name` | Минимальное возможное описание — указано только имя атрибута |
| `+name` | Указаны имя и открытая видимость — предполагается, манипуляции с именем будут производится непосредственно |
| `-name : String` | Указаны имя, тип и закрытая видимость — манипуляции с именем будут производится с помощью специальных операций |
| `-name [1..3] : String` | Указана кратность (для хранения трех составляющих; фамилии, имени и отчества) |
| `-name : String = "Novikov"` | Указано начальное значение |
| `+name : String {frozen}` | Атрибут объявлен не меняющим своего значения после начального присваивания и открытым[50] |

### *3.2.3. Операции*

*Операция* — это описание способа выполнить какие-то действия с объектом: изменить значения его атрибутов, вычислить новое значение по информации хранящейся в объекте и т. д.

Выполнение действий, определяемых операцией, инициируется *вызовом* операции. При выполнении операция может, в свою очередь, вызывать операции этого и других классов.

Описания операций класса перечисляются в разделе операций и имеют следующий синтаксис.

```
видимость ИМЯ (параметры) : тип {свойства}
```

Здесь слово `параметры` обозначает последовательность описаний параметров операции, каждое из которых имеет вид следующий формат.

```
направление ПАРАМЕТР : тип = значение
```

Начнем по порядку. Видимость, как обычно, обозначается с помощью знаков +, –, #.[51] Подчеркивание имени означает, что область действия операции — класс, а не

[50] В данном случае это, видимо, одно из наилучших решений: атрибут `name` инициализируется конструктором при создании объекта класса `Person` и после этого не меняет своего значения. В тоже время к атрибуту открыт доступ (фактически только на чтение) и нет нужды в операциях для работы со значением атрибута.

[51] Вообще говоря, видимость можно обозначать с помощью ключевых слов `private`, `public`, `protected`, что, например, встречается в инструменте Sun Java Studio Enterprise 8.

объект. Например, конструкторы имеют область действия класс. Курсивное написание имени означает, что операция абстрактная, т. е. в данном классе ее реализация не задана и должна быть задана в подклассах данного класса. После имени в скобках может быть указан список описаний параметров. Описания параметров в списке разделяются запятой. Для каждого параметра обязательно указывается имя, а также могут быть указаны направление передачи параметра, его тип и значение аргумента по умолчанию.

### Формальные параметры и фактические аргументы

Исторически сложилось так, что в отечественной литературе по программированию при описании параметров и аргументов процедур и функций наблюдается терминологическая путаница и разнобой. Слова "параметр" и "аргумент" часто используются как синонимы, а чтобы различать, о чем идет речь: о параметре, который назван при описании процедуры или о значении, которое должно быть подставлено вместо этого параметра при выполнении процедуры, применяются различные неуклюжие обороты, такие как "формальный параметр", "фактический параметр", "фактическое значение формального параметра" и др. Некоторое время тому назад мы (автор и близкие коллеги) приняли соглашение, согласно которому параметры всегда формальные, а аргументы всегда фактические. Таким образом, в операторе описания процедуры после имени процедуры в скобках идут имена параметров, а в операторе вызова процедуры после имени процедуры в скобках идут значения аргументов. Фактических параметров, равно как и формальных аргументов не бывает. Нам было очень приятно заметить, что в спецификации UML используется аналогичное соглашение, хотя это и не подчеркнуто явно. Данное соглашение устраняет всякую путаницу и мы придерживаемся его в этой книге и всех других своих книгах. Надеемся, что у нас найдутся последователи.

### Способы передачи параметров

Процедуры в языках программирования имеют параметры. Наличие параметров позволяет многократно применить одну и ту же процедуру для обработки или вычисления различных данных. Для того чтобы использовать эту возможность, при вызове процедуры нужно каким-то образом указать, откуда в данном конкретном вызове следует брать входные данные и куда нужно поместить результаты. Механизм такого указания называется *способом передачи параметров*.

В различных языках и системах программирования используются разные (иногда довольно замысловатые) способы передачи параметров в процедуры и функции. Мы опишем здесь два наиболее распространенных, которые в той или иной форме используется практически во всех языках и системах программирования: передача параметров по ссылке и по значению.

Если переменная[52] передается в процедуру по ссылке, то процедуре будет передан адрес этой переменной в памяти. При этом происходит

[52] Обычно системы программирования запрещают передавать по ссылке константы. Это вызвано тем, что процедура может изменить значение константы, и, тем самым, внести ошибку в ход дальнейших вычислений.

отождествление формального параметра процедуры и переданного фактического аргумента. Тем самым вызываемая процедура, изменяя значение аргумента, изменяет значение переданной ей переменной. Такой способ передачи подходит для выходных параметров.

Если же переменная (или константа) передается по значению, то компилятор создает временную копию этой переменной и именно адрес этой переменной-копии передается процедуре. Тем самым вызываемая процедура, изменяя значение аргумента, изменяет значение переменной-копии (но не самой переменной), которая будет уничтожена после завершения работы процедуры. Такой способ передачи подходит для входных параметров.

Направление передачи параметра в UML описывает семантическое назначение параметров, не конкретизируя конкретный механизм передачи. Как именно следует трактовать указанные в модели направления передачи параметров зависит от используемой системы программирования. Возможные значения направления передачи приведены в табл. 3.4.

**Таблица 3.4.** Ключевые слова для описания направления передачи параметров

| Ключевое слово | Назначение параметра |
|---|---|
| `In` | Входной параметр — аргумент должен быть значением, которое используется в операции, но не изменяется |
| `Out` | Выходной параметр — аргумент должен быть хранилищем, в которое операция помещает значение |
| `Inout` | Входной и выходной параметр — аргумент должен быть хранилищем, содержащим значение. Операция использует переданное значение аргумента и помещает в хранилище результат |
| `Return` | Значение, возвращаемое операцией. Никакого аргумента не требуется |

Типом параметра операции, равно как и тип возвращаемого операцией значения может быть любой встроенный тип или определенный в модели класс.

Все вместе (имя операции, параметры и тип результата) обычно называют *сигнатурой*. Стандарт предлагает считать сигнатурой имя операции плюс количество, порядок и типы параметров (т. е. направление передачи параметров и их имена, а также тип результата не входят в сигнатуру). Но это точка вариации семантики — в конкретном инструменте может быть реализовано другое понятие сигнатуры. Если сигнатуры различны, то и операции различны (даже если совпадают имена). В одном классе не может быть двух операций с одной сигнатурой — модель считается противоречивой. Если в подклассе определена операция с той же самой сигнатурой, то возможны два случая. Если описание операции в подклассе в точности то же самое или если оно является непротиворечивым расширением (например, в классе не был указан тип результата, а в подклассе он указан), то это повторное описание той же самой операции. Если же описание операции с совпадающей сигнатурой в подклассе противоречит описанию в классе (например, явно указаны различные направления передачи параметров), то модель считается противоречивой.

Операция имеет два важных свойства, которые указываются в списке свойств как именованные значения.

Во-первых, это `concurrency` — свойство, определяющее семантику одновременного (параллельного) вызова данной операции. В приложениях, где имеется только один поток управления, никаких параллельных вызовов быть не может. Действительно, если операция вызвана, то выполнение программы приостанавливается в точке вызова до тех пор, пока не завершится выполнение вызванной операции. В однопоточных приложениях в каждый момент времени управление находится в одной определенной точке программы и выполняется ровно одна определенная операция. Рекурсивный вызов (т. е. вызов операции из нее самой) не считается параллельным, поскольку при рекурсивном вызове выполнение операции, как обычно, приостанавливается и, таким образом, всегда выполняется только один экземпляр рекурсивной операции. Не так обстоит дело в приложениях, где имеется несколько потоков управления. В таком случае операция может быть вызвана из одного потока и в то время, пока ее выполнение еще не завершилось, вызвана из другого потока. Значение свойства `concurrency` определяет, что будет происходить в этом случае. Возможные варианты и их описания даны в табл. 3.5.

**Таблица 3.5.** Значения свойства `concurrency`

| Значение | Описание |
|---|---|
| `sequential` | Операция не допускает параллельного вызова (не является повторно-входимой). Если параллельный вызов происходит, то дальнейшее поведение системы не определено. |
| `Guarded` | Параллельные вызовы допускаются, но только один из них выполняется — остальные блокируются и их выполнение задерживается до тех пор, пока не завершится выполнение данного вызова.[53] |
| `concurrent` | Операция допускает произвольное число параллельных вызовов и гарантирует правильность своего выполнения. Такие операции называются повторно-входимыми (reenterable). |

Во-вторых, операция имеет свойство `isQuery`, значение которого указывает, обладает ли операция *побочным эффектом*. Если значение данного свойства `true`, то выполнение операции не меняет состояния системы — операция только вычисляет значения, возвращаемые в точку вызова.[54] В противном случае, т. е. при значение `false`, операция меняет состояние системы: присваивает новые значения атрибутам, создает или уничтожает объекты и т. п. По умолчанию операция имеет свойство `{isQuery=false}`. Поэтому, если нужно указать, что данная операция — это функция без побочных эффектов, то достаточно написать `{isQuery}`.

Рассмотрим примеры описания возможных операций класса Person информационной системы отдела кадров (табл. 3.7).

---

[53] Такая семантика может породить состояние взаимной блокировки (или тупика), в которой два или более процессов взаимно блокируют друг друга и ни один не может продолжить выполнение.

[54] Такие операции традиционно называют функциями.

**Таблица 3.7.** Примеры описания операций

| Пример | Пояснение |
|---|---|
| `move` | Минимальное возможное описание — указано только имя операции |
| `+move(in from, in to)` | Указаны видимость операции, направления передачи и имена параметров |
| `+move(in from : Dpt, in to : Dpt)` | Подробное описание сигнатуры: указаны видимость операции, направления передачи, имена и типы параметров |
| `+getName() : String {isQuery}` | Функция, возвращающая значение атрибута и не имеющая побочных эффектов |
| `+setPwd(in pwd : String = "password")` | Процедура, для которой указано значение аргумента по умолчанию |

### *3.2.4. Зависимости*

Всего в UML определено 17 стандартных стереотипов отношения зависимости, которые можно разделить на 6 групп:

- между классами и объектами на диаграмме классов;
- между пакетами;
- между вариантами использования;
- между объектами на диаграмме взаимодействия;
- между состояниями автомата;
- между подсистемами и моделями.

Здесь рассматриваются зависимости первой группы, которые перечислены в табл. 3.7.

**Таблица 3.7.** Стандартные стереотипы зависимостей на диаграмме классов

| Стереотип | Применение |
|---|---|
| `bind` | Подстановка параметров в шаблон. Независимой сущностью является шаблон (класс с параметрами), а зависимой — класс, который получается из шаблона заданием аргументов. См. пример в разд. 3.2.9. |
| `derive` | Буквально означает "может быть вычислен по". Зависимость с данным стереотипом применяется не только к классам, но и к другим элементам модели: атрибутам, ассоциациям и др. Суть состоит в том, зависимый элемент может быть восстановлен по информации, содержащейся в независимом элементе. Таким образом, данная зависимость показывает, что зависимый элемент, вообще говоря, излишен и введен в модель из соображений удобства, наглядности и т. д. |
| `friend` | Назначает специальные права видимости. Зависимый класс имеет доступ к составляющим независимого класса, даже если по общим правилам видимости он не имеет на это прав. |
| `instanceOf` | Указывает, что зависимый объект (или класс) является экземпляром независимого класса (метакласса). |
| `instantiate` | Указывает, что операции независимого класса создают экземпляры |

| | |
|---|---|
| | зависимого класса. |
| `powertype` | Показывает, что экземплярами зависимого класса являются подклассы независимого класса. Таким образом, в данном случае зависимый класс является метаклассом. См. пример в разд. 3.2.5. |
| `refine` | Указывает, что зависимый класс уточняет (конкретизирует) независимый. Данная зависимость показывает, что связанные классы концептуально совпадают, но находятся на разных уровнях абстракции. |
| `use` | Зависимость самого общего вида, показывающая, что зависимый класс каким-либо образом использует независимый класс. |

Повторим еще раз, что зависимости на диаграммах классов используются сравнительно редко, потому что имеют более расплывчатую семантику по сравнению с ассоциациями и обобщением. В нашей (достаточно простой) модели информационной системы отдела кадров не нашлось естественных примеров использования зависимостей на диаграмме классов, поэтому мы ограничимся немногочисленными ссылками на примеры в табл. 3.8.

### *3.2.5. Обобщение*

Отношение обобщения (разд. 1.4.2) часто применяется на диаграмме классов. Действительно, трудно представить себе ситуацию, когда между объектами в одной системе нет ничего общего. Как правило, общее есть и это общее целесообразно выделить в отдельный класс. При этом общие составляющие, собранные в суперклассе, автоматически наследуются подклассами. Таким образом, сокращается общее количество описаний, а значит, уменьшается вероятность допустить ошибку. Использование обобщений не ограничивает свободу проектировщика системы, поскольку унаследованные составляющие можно переопределить в подклассе, если нужно. При обобщении выполняется принцип подстановочности (см. разд.1.4.2.). Фактически это означает увеличение гибкости и универсальности программного кода при одновременном сохранении надежности, обеспечиваемой контролем типов. Действительно, если, например, в качестве типа параметра некоторой процедуры указать суперкласс, то процедура будет с равным успехом работать в случае, когда в качестве аргумента ей передан объект любого подкласса данного суперкласса. Суперкласс может быть конкретным, идентифицированным одним из методов, описанных в разд. 3.1.4, а может быть абстрактным, введенным именно для построения отношений обобщения.

Рассмотрим пример. В информационной системе отдела кадров мы выделили классы `Position`, `Department` и `Person` (см. разд. 3.1.3). Резонно предположить, что все эти классы имеют атрибут, содержащий собственное имя объекта, выделяющее его в ряду однородных. Для простоты положим, что такой атрибут имеет тип `String`.[55] В таком случае можно определить суперкласс, ответственный за хранение данного атрибута и работу с ним, а прочие классы связать с суперклассом

[55] Существуют организации, в которых подразделения не именуются, а нумеруются (или даже имеют шифрованные имена), например, из соображений секретности. Но наша информационная система отдела кадров не предназначена для организаций со столь строгим режимом.

отношением обобщения. Однако более пристальный анализ предметной области наводит на мысль, что работа с собственным именем для выделенных классов производится не совсем одинаково. Действительно, назначение и изменение собственных имен подразделениям и должностям находится в пределах ответственности информационной системы отдела кадров, но назначение (тем паче изменение) собственного имени сотрудника явно выходит за эти пределы.[56] Исходя из этих соображений, мы приходим к структуре обобщений, представленной на рис. 3.6. Обратите внимание, что суперкласс `Unit` мы определили как абстрактный, т. е. не могущий иметь непосредственных экземпляров, поскольку не предполагаем иметь в системе объекты данного класса. Класс `Unit` в данном нужен только для того, чтобы свести описания одного атрибута и двух операций в одно место и не повторять их дважды.

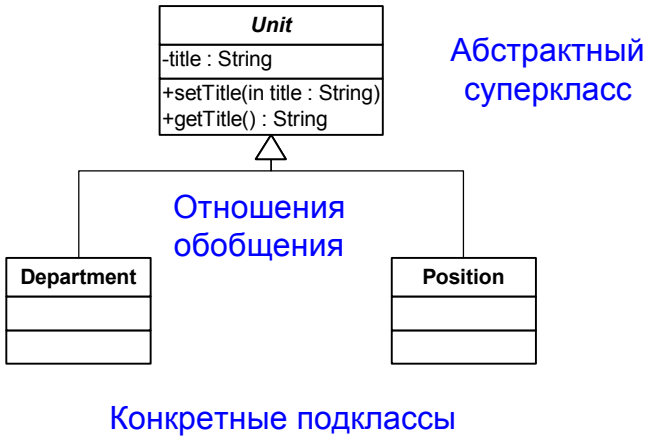



**Рис. 3.5. Отношение обобщения**

Отношения обобщения можно задавать в UML довольно свободно, но не совсем произвольно. Обобщения в модели должны образовывать строгий частичный порядок.

## Отношения порядка

Антисимметричное транзитивное отношение ◊ (см. разд. 2.1, вставка "Множества, отношения и графы") называется отношением *порядка*. Если свойство полноты не выполняется, то порядок называется *частичным*, а если, напротив, выполняется, то *полным*. Если вдобавок выполняется свойство антирефлексивности, то порядок называется строгим, в противном случае — нестрогим. Отношения порядка встречаются очень часто и играют важную роль во многих областях. Например, отношение < на множестве чисел является строгим полным порядком, а отношение ≤ — нестрогим полным порядком. Отношение "быть потомком" на множестве людей является строгим частичным порядком. Отношение включения ⊆ на множестве подмножеств данного множества является нестрогим частичным порядком.

[56] Бывают случаи, когда собственные имена сотрудников также назначаются — обычно такие имена называют кличками, но наша информационная системы отдела кадров также не предназначена для подобных организаций.

> Нетрудно показать, что орграфе, соответствующем отношению порядка, нет контуров (разве что петли, если порядок нестрогий). Отсюда немедленно следует, что в конечном упорядоченном множестве есть минимальные и максимальные элементы. В орграфе им соответствуют узлы, в которые не входят дуги (источники) и узлы, из которых не исходят дуги (стоки). Минимальных и максимальных элементов может быть несколько. Если в упорядоченном множестве ровно один максимальный и ровно один минимальный элемент, то это конечное вполне упорядоченное множество. Если же имеется ровно один минимальный элемент и несколько максимальных, то это дерево (иерархия).

Таким образом, в UML допускается, чтобы класс был подклассом нескольких суперклассов (множественное наследование), не требуется, чтобы у базовых классов был общий суперкласс (несколько иерархий обобщения) и вообще не накладывается никаких ограничений, кроме частичной упорядоченности (т. е. отсутствия циклов в цепочках обобщений). Нарушение данного условия является синтаксической ошибкой, однако не все инструменты проверяют это условие — цикл может быть незаметен, потому что отдельные дуги цикла обобщений могут быть показаны на разных диаграммах. При множественном обобщении возможны конфликты: суперклассы содержат составляющие, которые невозможно включить в один подкласс, например, атрибуты с одинаковыми именами, но разными типами. В UML конфликты при множественном обобщении считаются нарушением правил непротиворечивости модели (разд. 1.8.2).

Рассмотрим пример с описанием свойств личности. Для информационной системы отдела кадров этот пример несколько искусственен, но в сложных системах встречаются и более изощренные ситуации. Заодно здесь иллюстрируется использование стереотипа `metaclass`. Допустим, мы моделируем такие свойства личности, как пол и отношение к собственности.

Тогда данную ситуацию можно описать с помощью диаграммы, приведенной на рис. 3.7. Классы `Male` и `Female` являются экземплярами метакласса `Gender`. Классы `Owner` и `Employe` являются экземплярами метакласса `Status`. Это видно из имен подмножеств обобщений, указанных возле пунктирных линий, выделяющих подмножества. Классификация по полу является завершенной и дизъюнктной. Классификация по отношению собственности, напртив, не является ни завершенной, ни дизъюнктной.

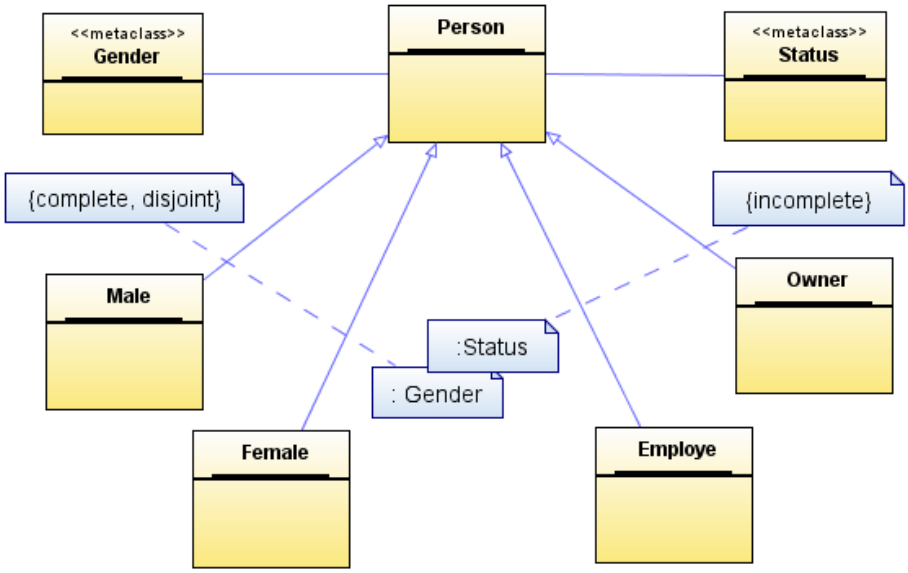



**Рис. 3.7. Подмножества обобщений и метаклассы**

Иногда бывает важно указать, что некоторый объект является экземпляром конкретного класса. Это делается с помощью стереотипа `instanceOf` (рис. 3.8). Далее, обычно конструктор объектов класса определяется в классе. Но иногда конструктор помещают в другой класс, который в таком случае называют *фабрикой*. На рис. 3.8 объект (единственный) класса `Company` является фабрикой объектов классов `Department` и `Person`, а объект класса `Department` является фабрикой объектов класса `Position`

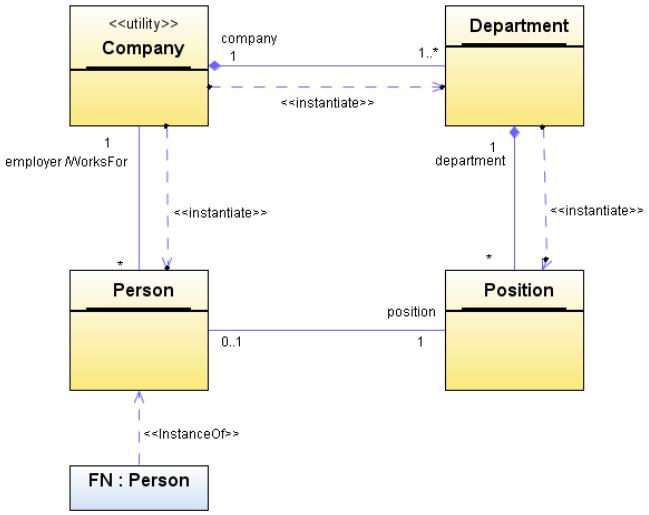



**Рис. 3.8. Инстанциирование и спецификация экземпляра**

## *3.2.6. Ассоциации*

Отношение ассоциации является, видимо, самым важным на диаграмме классов. В общем случае ассоциация, которая обозначается сплошной линией, соединяющей

классы, означает, что экземпляры одного класса связаны с экземплярами другого класса. Поскольку экземпляров может быть много, и каждый может быть связан с несколькими, ясно, что ассоциация является дескриптором, который описывает множество связанных объектов. В UML ассоциация является классификатором, экземпляры которого называются связями.

Связь между объектами (экземплярами классов) в программе может быть организована самыми разными способами. Например, в объекте одного класса может хранится указатель на объект другого класса. Другой вариант: объект одного класса является контейнером для объектов другого класса. Связь не обязательно является непосредственно хранимым физическим адресом. Этот адрес может динамически вычисляться во время выполнения программы на основании другой информации. Например, если объекты представлены как записи в таблице базы данных, то связь означает, в записи одного объекта имеется поле, значением которого является первичный ключ записи другого объекта (из другой таблицы). Еще пример: использование какого-либо механизма динамического связывания по имени (уникальному идентификатору) объекта. При моделировании на UML техника реализации связи между объектами не имеет значения. Ассоциация в UML подразумевает лишь то, что связанные объекты обладают достаточной информацией для организации взаимодействия. Возможность взаимодействия означает, что объект одного класса может послать сообщение объекту другого класса, в частности, вызвать операцию или же прочитать или изменить значение открытого атрибута. Поскольку в объектно-ориентированной программе такого рода действия и составляют суть выполнения программы, моделирование структуры взаимосвязей объектов (т. е. выявление ассоциаций) является одной из ключевых задач при разработке.

Как уже было сказано, базовая нотация ассоциации (сплошная линия) позволяет указать, что объекты ассоциированных классов могут взаимодействовать во время выполнения. Но это только малая часть того, что можно моделировать с помощью отношения ассоциации. Для ассоциации в UML предусмотрено наибольшее количество различных дополнений, которые мы сначала перечислим, а потом рассмотрим по порядку. Дополнения, как обычно, не являются обязательными: их используют при необходимости, в различных ситуациях по разному. Если использовать все дополнения сразу, то диаграмма становится настолько перегруженной, что ее трудно читать. Итак, для ассоциации определены следующие дополнения:

- имя ассоциации (возможно, вместе с направлением чтения);
- кратность полюса[57] ассоциации;
- вид агрегации полюса ассоциации;
- роль полюса ассоциации;
- направление навигации полюса ассоциации;
- упорядоченность объектов на полюсе ассоциации;
- изменяемость множества объектов на полюсе ассоциации;

[57] Напомним, что полюсом называется конец линии ассоциации. Обычно используются двухполюсные ассоциации, но могут быть и многополюсные. Пример приведен в конце раздела (см. рис. 3.22).

- квалификатор полюса ассоциации;
- класс ассоциации;
- видимость полюса ассоциации;
- многополюсные ассоциации.

Начнем по порядку. *Имя ассоциации* указывается в виде строки текста над (или под, или рядом с) линией ассоциации. Имя не несет дополнительной семантической нагрузки, а просто позволяет различать ассоциации в модели. Обычно имя не указывают, за исключением многополюсных ассоциаций или случая, когда одна и та же группа классов связана несколькими различными ассоциациями. Например, в информационной системе отдела кадров, если сотрудник занимает должность, то соответствующие объектов классов `Person` и `Position` должны быть связаны. Дополнительно можно указать направление чтения имени ассоциации. Фрагмент графической модели, приведенный на рис. 3.9, фактически можно прочитать вслух: `Person occupies position`.

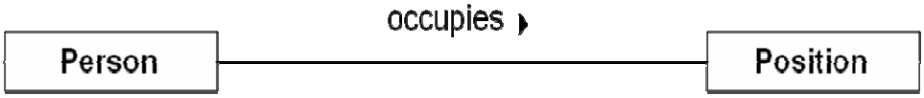



**Рис. 3.9. Имя ассоциации**

*Кратность полюса ассоциации* указывает, сколько объектов данного класса (со стороны данного полюса) участвуют в связи. Кратность может быть задана как конкретное число, и тогда в каждой связи со стороны данного полюса участвуют ровно столько объектов, сколько указано. Более распространен случай, когда кратность указывается как диапазон возможных значений, и тогда число объектов, участвующих в связи должно находится в пределах указанного диапазона. При указании кратности можно использовать символ `*`, который обозначает неопределенное число. Например, если в информационной системе отдела кадров не предусматривается дробление ставок и совмещение должностей, то (работающему) сотруднику соответствует одна должность, а должности соответствует один сотрудник или ни одного (должность вакантна). На рис. 3.10. приведен соответствующий фрагмент диаграммы UML.

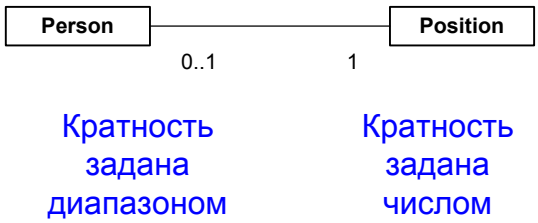



**Рис. 3.10. Кратность полюсов ассоциации**

Более сложные случаи также легко моделируются с помощью кратности полюсов. Например, если мы хотим предусмотреть совмещение должностей и хранить

информацию даже о неработающих сотрудниках, то диаграмма примет вид, приведенный на рис. 3.11 (запись `*` эквивалентна записи `0..*`).

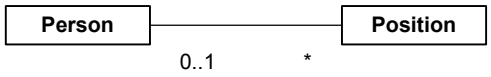



**Рис. 3.11. Использование неопределенной кратности**

*Агрегация и композиция*. В UML используются два частных, но очень важных случая отношения ассоциации, которые называются агрегацией и композицией. В обоих случаях речь идет о моделировании отношения типа «часть – целое». Ясно, что отношения такого типа следует отнести к отношениям ассоциации, поскольку части и целое обычно взаимодействуют.

*Агрегация* от класса `A` к классу `B` означает, что объекты (один или несколько) класса `A` входят в состав объекта класса `B`.

Это отмечается с помощью специального графического дополнения: на полюсе ассоциации, присоединенному к «целому», т. е., в данном случае, к классу `B`, изображается ромб. Например, на рис. 3.12 указано, что подразделение является частью компании.

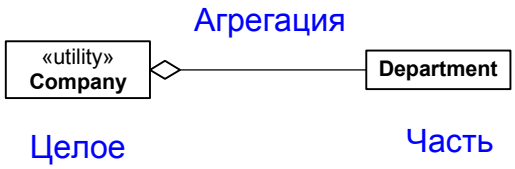



**Рис. 3.12. Агрегация**

При этом никаких дополнительных ограничений не накладывается: объект класса `A` (часть) может быть связан отношениями агрегации с другими объектами (т. е. участвовать в нескольких агрегациях), создаваться и уничтожаться независимо от объекта класса `B` (целого).

*Композиция* накладывает более сильные ограничения: композиционно часть может входить только в одно целое, часть существует только пока существует целое и прекращает свое существование вместе с целым.

Графически отношение композиции отображается закрашенным ромбом. Для примера на рис. 3.13 приведены два возможных взгляда на отношения между подразделениями и должностями в информационной системе отдела кадров. В первом случае (вверху), мы считаем, что в организации принята жесткая («армейская») структура: каждая должность входит ровно в одно подразделение, в каждом подразделении есть по меньшей мере одна должность (начальник). Во втором случае (внизу) структура организации более аморфна: возможны «висящие в воздухе» должности, бывают «пустые» подразделения и т. д.

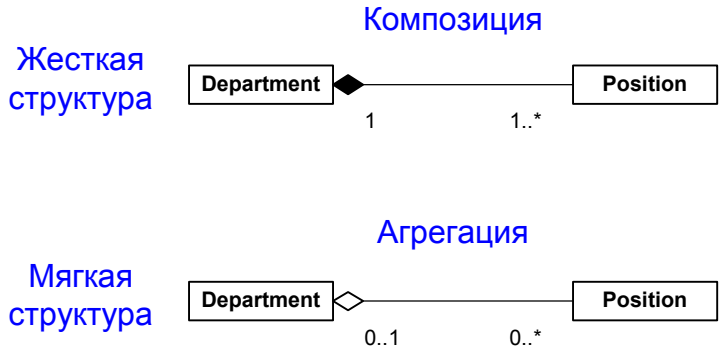



**Рис. 3.13. Сравнение композиции (вверху) и агрегации (внизу)**

В комбинации с указанием кратности, отношения ассоциации, агрегации и композиции позволяют лаконично и полно отобразить структуру связей объектов: что из чего состоит и как связано. На рис. 3.13 приведен пример одного из вариантов такой структуры для информационной системы отдела кадров.

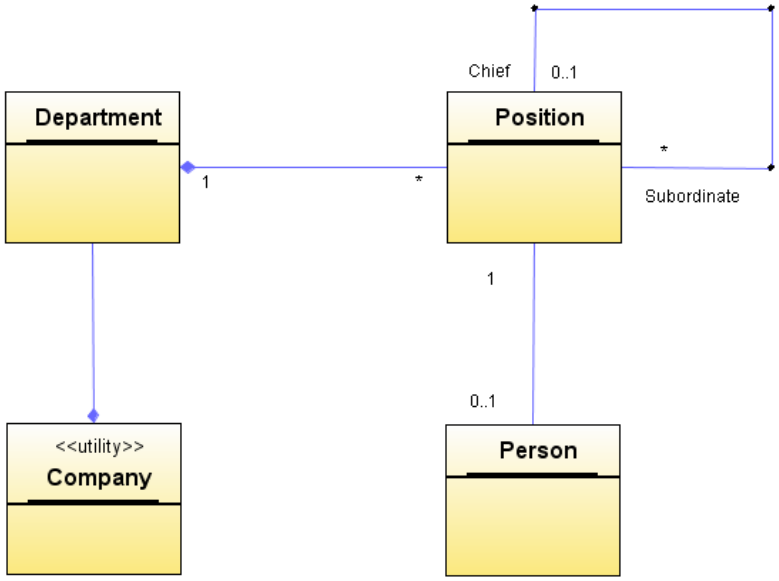



**Рис. 3.14. Структура связей классов информационной системы отдела кадров**

Мы хотим закончить объяснение агрегации и композиции некоторой не совсем универсальной и точной, но понятной программистской интерпретацией. Допустим, что в классе `B` имеется атрибут класса `A`. В этом случае естественно считать, что объект класса `A` является частью объекта класса `B`. Если значением атрибута является ссылка на объект класса `A` (указатель), то это агрегация, а если значением атрибута является непосредственно экземпляр объекта класса `A` (встраивание в смысле Visual Basic), то это композиция.[58]

[58] Некоторые инструменты автоматически помещают на диаграмму классов соответствующую линию композиции или агрегации при определении таких атрибутов.

*Роль полюса ассоциации*, называемая также *спецификатором интерфейса* — это способ указать, как именно участвует классификатор (присоединенный к данному полюсу ассоциации) в ассоциации. В общем случае данное дополнение имеет следующий синтаксис:

```
видимость ИМЯ : тип
```

Имя является обязательным, оно называется *именем роли* и фактически является собственным именем полюса ассоциации, позволяющим различать полюса. Если рассматривается одна ассоциация, соединяющая два различных класса, то в именах ролей нет нужды: полюса ассоциации легко можно различить по именам классов, к которым они присоединены. Однако, если это не так, т. е. если два класса соединены несколькими ассоциациями, или же если ассоциация соединяет класс с самим собой, то указание роли полюса ассоциации является необходимым. Такая ситуация отнюдь не является надуманной, как может показаться. Вернемся к рис. 3.14, где мы, забегая вперед, начали разбор примера на эту тему. На этом рисунке изображена ассоциация класса `Position` с самим собой. Эта ассоциация призвана отразить наличие иерархии подчиненности должностей в организации. Однако из рис. 3.14 видно только, что объекты класса `Person` образуют некоторую иерархию (каждый объект связан с некоторым количеством нижележащих в иерархии объектов и не более чем с одним вышележащим объектом), но не более того. Используя спецификацию интерфейсов и, заодно, (опять несколько забегая вперед) отношения реализации и интерфейсы можно описать субординацию в информационной системе отдела кадров достаточно лаконично, но точно. Например, на рис. 3.15 указано, что в иерархии субординации каждая должность может играть две роли. С одной стороны, должность может рассматриваться как начальственная (`chief`), и в этом случае она предоставляет интерфейс `IChief`[59] имеющий операцию `petition` (начальнику можно подать служебную записку). С другой стороны, должность может рассматриваться как подчиненная (`subordinate`), и в этом случае она предоставляет интерфейс `ISubordinate`, имеющий операцию `report` (от подчиненного можно потребовать отчет). У начальника может быть произвольное количество подчиненных, в том числе и 0, у подчиненного может быть не более одного начальника.

[59] Сложилась устойчивая традиция начинать имена интерфейсов с прописной буквы I.

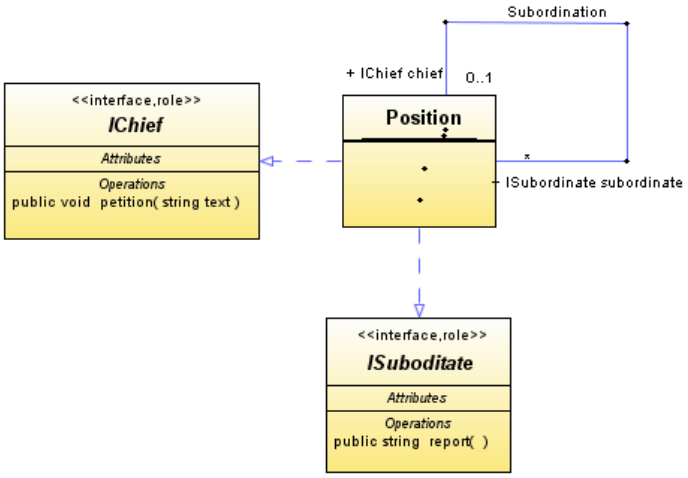



**Рис. 3.15. Роли полюсов ассоциации**

Имя в виду ту же самую цель, можно поступить и по-другому: непосредственно включить в описание класса составляющие, ответственные за обеспечение нужной функциональности (рис. 3.16). Однако такое решение "не в духе" UML: во-первых, оно слишком привязано к реализации, разработчику не оставлено никакой свободы для творческих поисков эффективного решения; во-вторых оно менее наглядно — неудачный выбор имен атрибутов способен замаскировать семантику отношения для читателя модели, потеряны информативные имена ролей и ассоциации; в-третьих, оно менее надежно — в модели на рис. 3.15 подчиненный синтаксически не может потребовать отчета от начальника, а в модели на рис. 3.16 — может, и нужно предусматривать дополнительные средства для обработки этой ошибки.

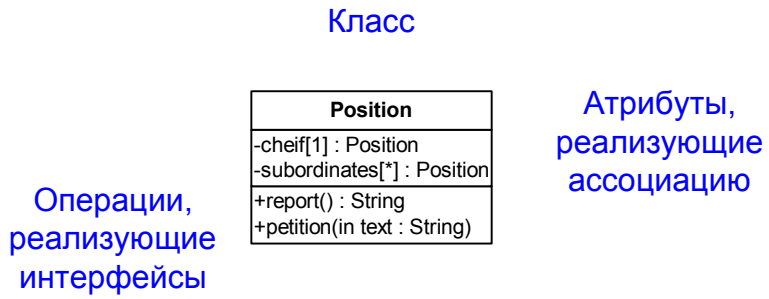



**Рис. 3.16. Атрибуты, обеспечивающие реализацию ролей**

| |
|---|
| *Направление навигации* полюса ассоциации — это свойство полюса, имеющее значение типа `Boolean`, и определяющее, можно ли получить с помощью данной ассоциации доступ к объектам класса, присоединенному к данному полюсу ассоциации. |

По умолчанию это свойство имеет значение `true`, т. е. доступ возможен. Если все полюса ассоциации (обычно их два) обеспечивают доступ, то это никак не отражается на диаграмме (потому, что данный случай наиболее распространенный и предполагается по умолчанию). Если же навигация через некоторые полюса

возможна, а через другие нет, то те полюса, через которые навигация возможна, отмечаются стрелками на концах линии ассоциации. Таким образом, если никаких стрелок не изображается, то это означает, что подразумеваются стрелки во всех возможных направлениях. Если же некоторые стрелки присутствуют, то это означает, что доступ возможен только в направлениях, указанных стрелками. Отсюда следует, что в UML невозможно изобразить случай, когда навигация вдоль ассоциации невозможна ни в каком направлении — но это и не нужно, т. к. такая ассоциация не имеет смысла.

Рассмотрим следующий пример (который может быть полезен и в информационной системе отдела кадров, если понадобиться расширить ее функциональность защитой данных на уровне пользователя). Допустим, имеются два класса: `User` (содержит информацию о пользователе) и `Password` (содержит пароль — информацию, необходимую для аутентификации пользователя). Мы хотим отразить в модели следующую ситуацию: имеется взаимно однозначное соответствие между пользователями и паролями, зная пользователя можно получить доступ к его паролю, но обратное неверно: по имеющемуся паролю нельзя определить, кому он принадлежит. Такая ситуация может быть промоделирована, как показано на рис. 3.17.

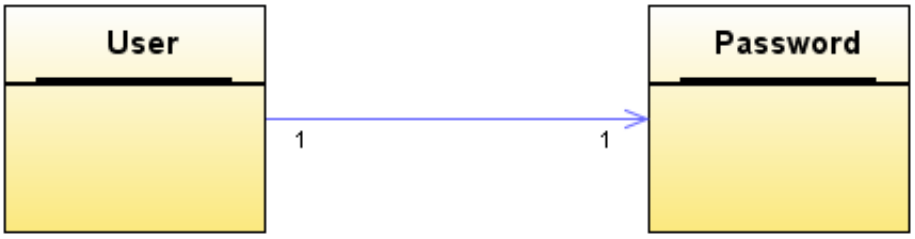



**Рис. 3.17. Направление навигации**

*Видимость полюса ассоциации* — это указание того, является ли классификатор присоединенный к данному полюсу ассоциации, видимым для других классификаторов вдоль данной ассоциации, помимо тех классификаторов, которые присоединены к другим полюсам ассоциации.

Пожалуйста не думайте, что мы хотим вас запутать с помощью подобных формальных определений. На самом деле все очень просто. Продолжим рассмотрение предыдущего примера. Допустим, у нас есть третий класс — `Workgroup` — ответственный за хранение информации о группах пользователей. Тогда очевидно, что зная группу, мы должны иметь возможность узнать, какие пользователи входят в группу, и обратно, для каждого пользователя можно определить в какую группу (или в какие группы) он включен. Но не менее очевидно, что доступ к группе не должен позволять узнать пароли пользователей этой группы. Другими словами, мы хотим ограничить доступ к объектам через полюс ассоциации. В UML для этого нужно явно указать имя роли полюса ассоциации, и перед именем роли поставить соответствующий символ (или ключевое слово) видимости (рис. 3.18).

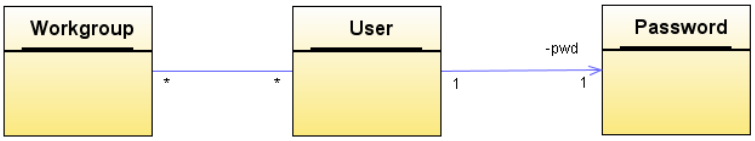



**Рис. 3.18. Ограничение видимости полюса ассоциации**

Обратите внимание, что, согласно семантическим правилам UML, ограничение видимости полюса `pwd` распространяется на объекты класса `Workgroup`, но не на объекты класса `User` — непосредственно связанные объекты всегда видят друг друга. Объект класса `User` может использовать интерфейс `pwd`, предоставляемый классом `Password` — стрелка навигации разрешает ему это, но объект класса `Workgroup` не может использовать интерфейс `pwd` — значение видимости `private` (знак –) запрещает ему это.

Следующая два дополнения встречаются сравнительно редко, но иногда без них не обойтись. Они не имеют специальной графической нотации, а записываются в виде стандартных ограничений на полюсах ассоциации.

*Упорядоченность объектов на полюсе ассоциации*. Если кратность полюса лежит в диапазоне `0..1`, то проблемы упорядоченности не возникает — на данном полюсе связи, возможно, имеется один объект, но не более того. При иных кратностях объектов может быть несколько, т. е. полюс связи присоединен к множеству объектов. По умолчанию множество объектов, присоединенных к данному полюсу связи, считается неупорядоченным (как и любое множество, если не оговорено противное). Если необходимо указать, что это множество упорядочено (см. пример ниже и рис. 3.19), то нужно наложить соответствующее ограничение на полюс ассоциации. В качестве ограничения используется одно из двух ключевых слов:

- `ordered` (или `sorted`) если множество упорядочено;
- `unordered`, если не упорядочено.

Обычно считается, что множество объектов на полюсе связи может изменяться произвольным образом (в пределах специфицированной кратности). Например, в один момент работы информационной системы отдела кадров в данном подразделении может быть одних 10 должностей, а в другой — 20 других. Совершенно аналогично, значение атрибута обычно может произвольным образом меняться в процессе жизни объекта (в пределах указанного типа атрибута). Однако иногда необходимо определенным образом ограничить изменяемость атрибута (см. разд. 3.2.2 и табл. 3.3). Аналогично иногда нужно ограничить *изменяемость* состава множества объектов присоединенных к полюсу связи (экземпляру ассоциации). Для этого применяется тот же самый набор стандартных значений (см. еще раз табл. 3.3). В информационной системе отдела кадров мы не нашли походящего примера, и для иллюстрации двух последних понятий рассмотрим элементарный пример из вычислительной геометрии. Допустим, что у нас есть класс `Point`, экземплярами которого являются точки (на плоскости). Многоугольник (`Polygon`) можно определить, как упорядоченное множество точек (вершин многоугольника), причем резонно предположить, что состав вершин данного многоугольника, после того, как он определен, не может меняться. Модель данной ситуации приведена на рис. 3.19.


**Рис. 3.19. Упорядоченность и изменяемость множества объектов на полюсе связи**

Рассмотрим ситуацию, когда два класса связаны ассоциацией "один ко многим" (именно этот случай имеет место в данном примере). Такая ассоциация доставляет для каждого экземпляра класса, находящегося на полюсе "один" множество (иногда большое) объектов класса, находящегося на полюсе "много". Например, по экземпляру многоугольника можно получить множество его вершин. Однако часто возникают ситуации, когда нужно получить не все множество ассоциированных объектов, а некоторое небольшое подмножество, чаще всего один конкретный объект. Чтобы выделить из множества один конкретный элемент, нужно располагать информацией, однозначно идентифицирующей этот элемент. Такую информацию принято называть ключом. Например, номер вершины в многоугольнике можно считать ключом и тогда задача ставится так: по многоугольнику и номеру вершины получить вершину. Всегда можно применить следующее тривиальное решение: хранить ключ в самом объекте в качестве атрибута и, получив множество объектов, перебирать их все последовательно до тех пор, пока не найдется тот, который имеет искомое значение ключа. Такой прием называется *линейным поиском*. Для многоугольников, у которых не очень много вершин, данный способ может быть вполне приемлемым. Но в других случаях, когда к полюсу "много" присоединено действительно *много* объектов, линейный поиск слишком неэффективен. Известно множество программных решений, позволяющих эффективно выделить (найти в множестве) объект по ключу: сортированные массивы, таблицы расстановки (хэш-таблицы), деревья сортировки, внешние индексы и др. Эти приемы обобщены в UML понятием квалификатора.

*Квалификатор полюса ассоциации* — это атрибут (или несколько атрибутов) ассоциации, значение которого (которых) позволяет выделить один (или несколько) объектов класса, присоединенного к данному полюсу ассоциации.

Квалификатор изображается в виде небольшого прямоугольника на полюсе ассоциации, примыкающего к прямоугольнику класса. Внутри этого прямоугольника (или рядом с ним) указывается имена и, возможно, типы атрибутов квалификатора. Описание квалифицирующего атрибута ассоциации имеет такой же синтаксис, что и описание обычного атрибута класса, только оно не может содержать начального значения. Квалификатор может присутствовать только на том полюсе ассоциации, который имеет кратность "много", поэтому, если на полюсе ассоциации с квалификатором задана кратность, то она указывает не допустимую мощность множества объектов, присоединенных к полюсу связи, а допустимую мощность того подмножества, которое определяется при задании

значений атрибутов квалификатора. Например, на рис. 3.20 приведен фрагмент модели для примера с многоугольниками, в котором использован квалификатор (в данном случае с именем `number`). Обратите внимание, что на полюсе ассоциации с квалификатором, присоединенном к классу `Point`, указана кратность `0..1` (а не `*`, как на рис. 3.19). Действительно, само наличие квалификатора указывает на кратность много, а кратность `0..1` указывает сколько вершин с конкретным номером связано с данным многоугольником: 1, если задан допустимый номер вершины, или 0 в противном случае.

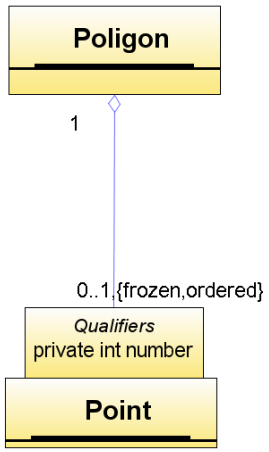



**Рис. 3.20. Квалификатор**

Ассоциация имеет экземпляры (связи), стало быть, является классификатором и может обладать соответствующими свойствами и составляющими классификатора. В распространенных случаях, рассмотренных выше, ассоциации не обладали никакими собственными составляющими. Грубо говоря, ассоциация между классами *A* и *B* — это просто множество пар (*a*,*b*), где *a* — объект класса *A*, а *b* — объект класса *B*. Подчеркнем еще раз, что это именно множество: двух одинаковых пар (*a*,*b*) быть не может. Однако возможны и более сложные ситуации, когда ассоциация имеет собственные атрибуты (и даже операции), значения которых хранятся в экземплярах ассоциации — связях. В таком случае применяется специальный элемент моделирования — класс ассоциации.

*Класс ассоциации* — это сущность, которая имеет как свойства класса, так и свойства ассоциации.

Класс ассоциации изображается в виде символа класса, присоединенного пунктирной линией к линии ассоциации.

Вернемся к информационной системе отдела кадров и рассмотрим следующий пример. Допустим, что имеет место более сложное отношение между должностями и сотрудниками, нежели те, что приведены на рис. 3.10 и 3.11. А именно, допускается не только совмещение должностей (один сотрудник может работать на нескольких должностях), но и дробление ставок (одну должность могут занимать несколько сотрудников — полставки, четверть ставки и т. п.). Использую уже

разобранную нотацию ассоциации, мы можем констатировать, что между классами `Person` и `Position` имеет место ассоциация "многие ко многим". Однако этого недостаточно: необходимо указать, какую долю данной должности занимает данный сотрудник. Эту информацию нельзя отнести ни к должности, ни к сотруднику — это атрибут класса ассоциации между ними (рис. 3.21).

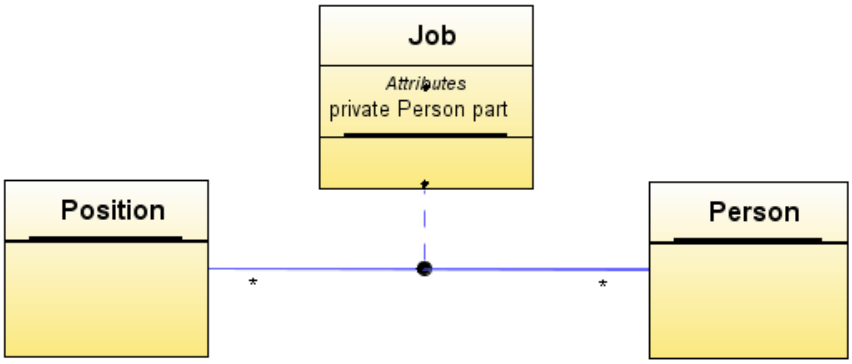



**Рис. 3.21. Класс ассоциации**

Может показаться, что понятие класса ассоциации в UML является надуманным и излишним. Действительно, можно применить стандартный прием нормализации, который часто используется при проектировании схем баз данных: систематическим образом избавиться от отношений "многие ко многим" путем введения дополнительной сущности и двух отношений "один ко многим". Применительно к данному примеру такой прием дает решение, приведенное на рис. 3.22.

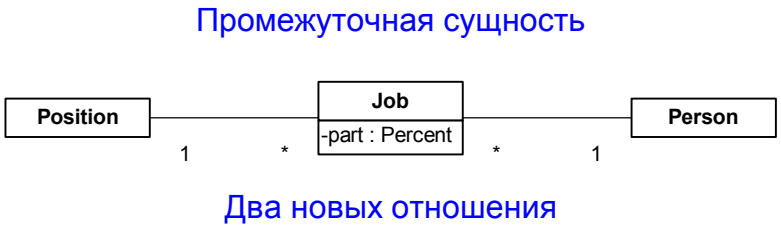



**Рис. 3.22. Элиминация отношения "многие ко многим" с помощью введения дополнительной сущности**

На первый взгляд, модели на рис. 3.21 и 3.22 выглядят семантически эквивалентными, однако это не так — здесь есть тонкое различие. В случае использования класса ассоциации `Job`, который по определению является множеством пар должность–сотрудник, не может быть двух одинаковых пар. То есть не может быть так, чтобы Новиков занимал полставки доцента и еще (отдельно) четверть той же ставки. В случае же промежуточного класса `Job` это, вообще говоря, вполне возможно, что не совсем естественно. Опытные проектировщики баз данных нам могут возразить, что это вполне поправимо: нужно ввести в классах `Position` и `Person` атрибуты, которые будут уникальными

ключами, идентифицирующими объекты этих классов, а в классе `Job` ввести пару атрибутов, значениями которых будут ключи классов `Position` и `Person` и потребовать, чтобы эта пара атрибутов была уникальным составным ключом класса `Job`. Мы не будем спорить — это действительно так и делается при традиционном проектировании схем баз данным. Нам представляется, что разбор данного примера уже является достаточным обоснованием полезности элегантного понятия класса ассоциации в UML: один рис. 3.21 понятнее и точнее рис. 3.22 с неизбежными добавлениями и пояснениями про уникальные ключи.

Чаще всего используются бинарные ассоциации, отражающие связи между объектами двух классов. В UML определены также *многополюсные ассоциации*, отражающие связи между большим числом объектов. С формальной точки зрения многополюсные ассоциации излишни, поскольку их можно выразить через комбинацию бинарных ассоциаций введением дополнительных сущностей. Действительно, упорядоченную тройку объектов (*a*, *b*, *c*) — элемент трехполюсной ассоциации — можно представить как упорядоченную пару (*a*, *d*), где *d* — новый объект, представляющий упорядоченную пару (*b*, *c*). Однако на практике (в некоторых случаях) многополюсные ассоциации бывают буквально незаменимы. Рассмотрим следующий пример из информационной системы отдела кадров. Допустим, что в организации применяется современная организационная форма управления и помимо иерархии подразделений и должностей существует структура выполняемых проектов, "пронизывающих" организацию.[60] Один и тот же сотрудник может участвовать во многих проектах, выполняя различные обязанности (т. е. занимая различные должности), в каждом проекте (частично) заняты многие сотрудники и размер оплаты труда зависит от того, сколько конкретно времени проработал данный сотрудник в данной должности в данном проекте. Такую замысловатую (но весьма жизненную!) ситуацию очень легко отобразить, используя многополюсные ассоциации и классы ассоциации (рис. 3.23). Чтобы полностью оценить полезность многополюсных ассоциаций и классов ассоциаций, мы советуем читателям проделать весьма поучительное упражнение: попытаться построить модель, семантически эквивалентную рис. 3.23, но не использующую многополюсных ассоциаций и классов ассоциаций (подсказка: придется ввести дополнительные сущности и отношения).

[60] Речь идет о модной матричной структуре управления.

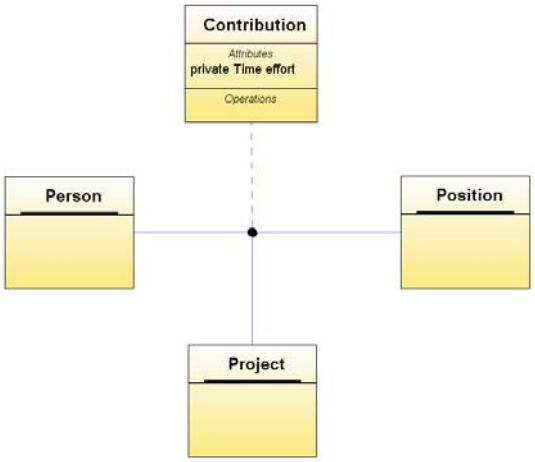



**Рис. 3.23. Многополюсная ассоциация**

Мы хотим подвести итог этому пространному разделу, обобщив сказанное с помощью диаграммы метамодели ассоциации, представленной на рис. 3.24.

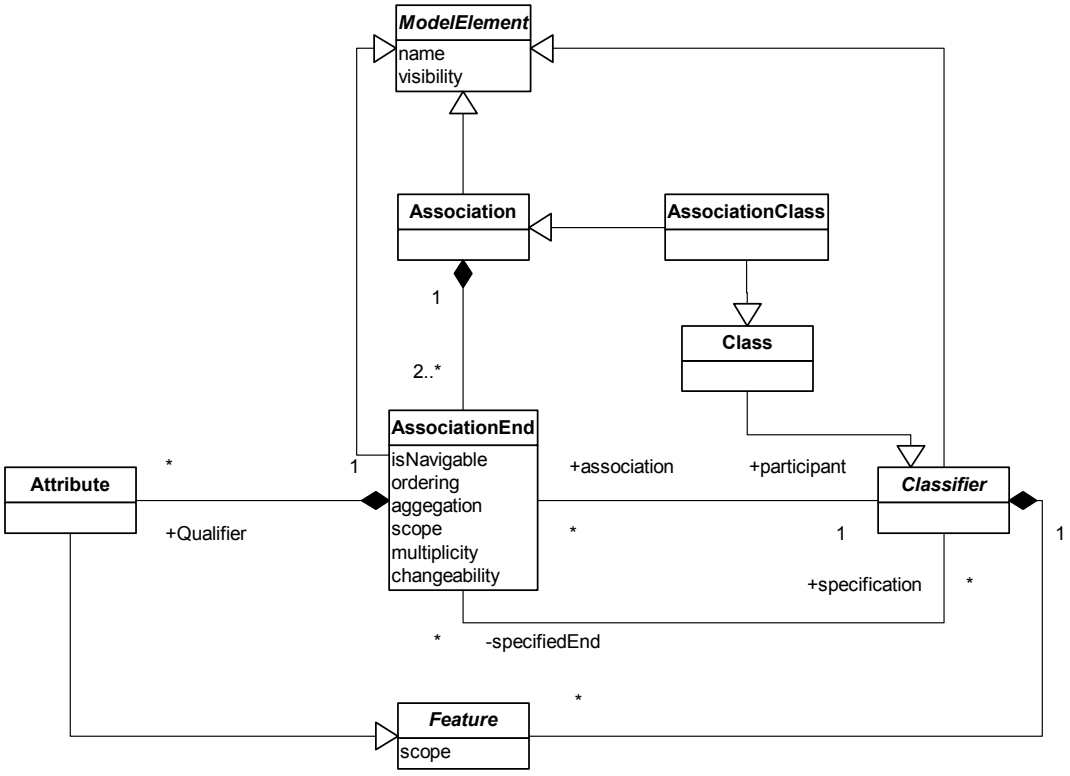



**Рис. 3.24. Метамодель ассоциации**

## *3.2.7. Интерфейсы и роли*

В UML имеется несколько частных случаев классификаторов, которые, подобно классам, предназначены для моделирования структуры, но обладают рядом

специфических особенностей. Наиболее важными их них являются интерфейсы, которые рассматриваются в этом разделе и типы данных, рассмотренные в следующем разделе.

*Интерфейс* — это именованный набор абстрактных операций.

Другими словами, интерфейс — это абстрактный класс, в котором нет атрибутов и все операции абстрактны. Поскольку интерфейс — это абстрактный класс, он не может иметь непосредственных экземпляров.

Между интерфейсами и другими классификаторами, в частности классами, на диаграмме классов применяются два отношения:

- классификатор (в частности, класс) использует интерфейс — это показывается с помощью зависимости со стереотипом «call»;
- классификатор (в частности, класс) реализует интерфейс — это показывается с помощью отношения реализации.

Никаких ограничений на использование отношения реализации не накладывается: класс может реализовывать много интерфейсов, и наоборот, интерфейс может быть реализован многими классами. Нет ограничений и на использование зависимостей со стереотипом «call» — класс может вызывать любые операции любых видимых интерфейсов. Семантика зависимости со стереотипом «call» очень проста — эта зависимость указывает, что в операциях класса, находящегося на независимом полюсе, вызываются операции класса (в частности, интерфейса) находящегося на зависимом полюсе.

Разобравшись с интерфейсами, их реализацией и использованием, нам представляется уместным еще раз повторить определение понятия, которое несколько раз уже проскальзывало (см. разд. 2.2.1, 3.1.3).

*Роль* — это интерфейс, который предоставляет классификатор в данной ассоциации.

Мы надеемся, что туман неясности по поводу понятия "роль" в UML, порожденный нашими постоянными, но неизбежными забеганиями вперед, теперь полностью развеялся.

Продолжая пример информационной системы отдела кадров, начатый на рис. 3.15, дополним его иллюстрацией использования зависимости со стереотипом «call». Допустим, что класс `Department` для реализации операций связанных с движением кадров, использует операции класса `Position`, позволяющие занимать и освобождать должность — другие операции класса `Position` классу `Department` не нужны. Для этого, как показано на рис. 3.25 можно определить соответствующий интерфейс `IPosition` и связать его отношениями с данными классами.

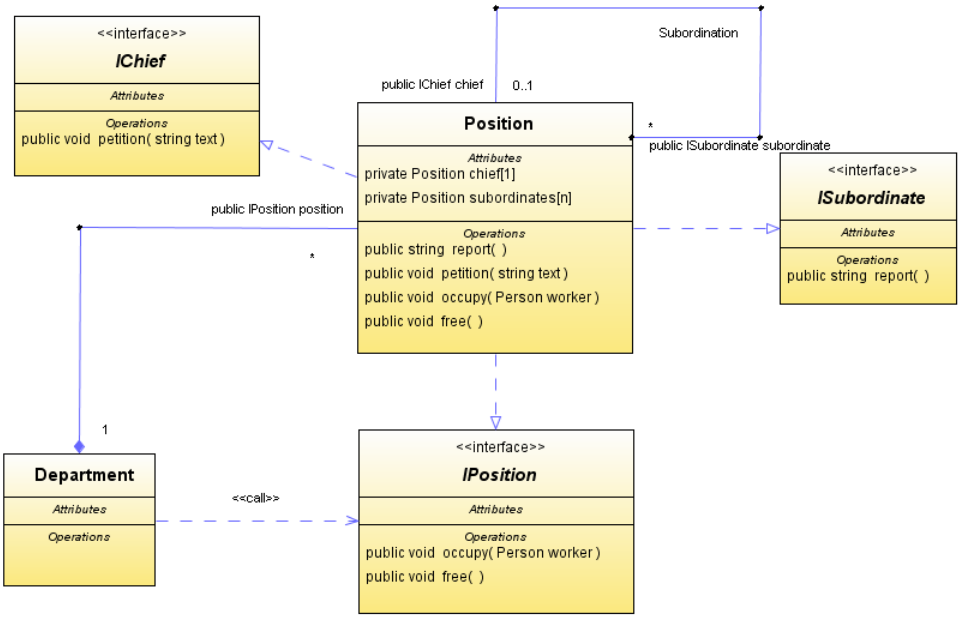



**Рис. 3.25. Отношения реализации и использования интерфейсов**

Используя нотацию «чупа-чупс», появившуюся в UML 2.0, эту же модель можно изобразить лаконично, сиимметрично и просто, как показано на рис. 3.26.

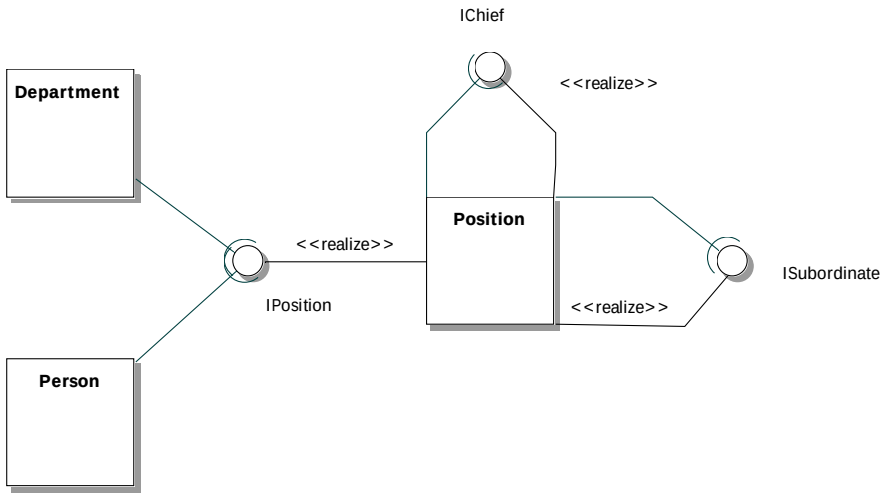



**Рис. 3.25. Использование нотации «чупа-чупс»**

## *3.2.8. Типы данных*

Понятие типа данных и все связанное с ним является одной из самых заслуженных и важных концепций программирования.

*Тип данных* — это совокупность двух вещей: множества значений (может быть очень большого или даже потенциально бесконечного) и конечного множества операций, применимых к данным значениям.

Чтобы понять, в чем важность данного понятия для программирования, нужно вернуться назад, к началу истории развития программирования и спуститься вниз, на уровень реализации и представления данных в памяти компьютера. Допустим, что речь идет о программе на машинном языке, т. е. последовательности команд, которые может выполнять процессор компьютера. Эти команды работают с ячейками памяти, в которых хранятся последовательности битов. Разработаны и используются методы представления в виде последовательностей битов данных любой природы: чисел, символов, графики и т. д. Так вот, в наиболее распространенных в настоящее время компьютерных архитектурах по последовательности битов в ячейке нельзя сказать, данные какой природы закодированы в ней: это может быть и код числа, и коды нескольких символов, и оцифрованный звук.[61] Поскольку для процессора все коды в ячейках "на одно лицо" он выполнит любую допустимую команду над любыми данными. Например, выполнит команду умножения чисел над кодами символов. Такая ситуация возможна по двум причинам:

- либо (наиболее вероятно) это ошибка программиста — он указал не ту команду или не тот операнд;
- либо это хитроумный программистский трюк.

### Программистские трюки

Молодые, в особенности способные, программисты любят применять в своих программах трюки: необычные или нестандартные приемы, требующие изобретательности и особых знаний. Рассмотрим, например, такой оператор на языке C:

```
a = ( a < b ) ? a : ( b = ( a = a ^ b ) ^ b ) ^ a ;
```

Даже хорошо знающему язык C человеку нелегко с первого взгляда определить, что делает этот оператор. Между тем, он выполняет простое и полезное преобразование: если значение переменной `a` меньше значения переменной `b`, то ничего не делается, в противном случае переменные меняются значениями. Это типичный трюк. Начинающие программисты гордятся своими трюками (и стыдятся своих ошибок). Между тем, с точки зрения стороннего потребителя, между трюком и ошибкой нет существенной разницы: и то и другое уменьшает надежность программы, затрудняет ее понимание и препятствуют повторному использованию. К счастью, с опытом склонность к трюкачеству обычно проходит.

Чтобы предупредить нежелательные последствия применения команд к неподходящим данным, в языках программирования, особенно в языках высокого уровня, используется концепция типа данных: элементам языка, ответственным за хранение и представление данных, в частности, переменным, приписывается тип.

---

[61] Следует сделать важную оговорку: это так в большинстве компьютерах, но не во всех. Уже давно существуют и применяются компьютеры, архитектура которых такова, что природа данных хранимых в памяти однозначно определяется по самим этим данным. Как правило, это достигается тем, что вместе с данными в качестве их неотъемлемой части хранится информация (называемая тегом) о типе этих данных.

Идея состоит в том, что элемент данного типа может содержать значения только этого типа, и обрабатываться только процедурами данного типа.
По способу приписывания типа языки программирования подразделяются на языки со статической типизацией, когда тип элемента языка (переменной) не может меняться во время выполнения программы и языки с динамической типизацией, в которых тип переменной может меняться по ходу выполнения.

### Статический и динамический контроль типов

> В языках со статической типизацией возможен *статический контроль типов*. Суть состоит в том, что статически, т. е. до начала выполнения программы, компилятор проверяет, что к данным определенного типа применяются только соответствующие операции этого типа. В языках с динамической типизацией возможен только *динамический контроль типов*, т. е. проверка допустимости выполнения операции производится динамически, во время выполнения программы. Понятно, что динамический контроль типов отрицательно сказывается на эффективности программы, поэтому статический контроль типов предпочтительнее. Однако, многие конструкции распространенных языков программирования, например, указатели `void*` или динамические массивы не позволяют проводить полный статический контроль типов. Более того, используются такие конструкции, которые вообще не позволяют автоматически проверять тип, например, `union` в C или `record case` в Паскале. Таким образом, типизация и контроль типов являются, несомненно, полезными концепциями, повышающими надежность программ, но они не всеобъемлющи и не дают полной гарантии от ошибок.

UML не является сильно типизированным языком: например, в модели можно указывать типы атрибутов классов и параметров операций, но это не обязательно. Инструмент может проверять соответствие типов, если они указаны, но не обязан этого делать. (Контроль типов — еще один пример точки вариации семантики в языке). Такое решение принято в расчете на то, что UML используется совместно с разными языками программирования, использующими различные концепции типизации и типового контроля, и навязывание одной конкретной модели ограничило бы применение UML.
Здесь уместно дать точные ответы на два важных вопроса.

- Для каких элементов модели можно указать тип?
- Что можно использовать в качестве указания типа?

Ответ на первый вопрос разбросан по тексту книги. Сконцентрируем здесь необходимые ссылки. В UML типизированы могут быть:

- атрибуты классов (см. раздел 3.2.2), в том числе классов ассоциаций (см. раздел 3.2.6);
- параметры операций, в том числе тип возвращаемого значения (см. раздел 3.2.3);
- роли полюсов ассоциаций (см. раздел 3.2.6);
- квалификаторы полюсов ассоциаций (см. раздел 3.2.6);
- параметры шаблонов (см. раздел 3.2.9).

Ответ на второй вопрос — что же можно указать в качестве типа — с одной стороны, очень лаконичен, а, с другой стороны, требует дополнительного

обсуждения. Лаконичный ответ звучит так: тип указывается с помощью классификатора. Обсудим это определение. Если типы составляющих одного классификатора указываются с помощью других классификаторов, то возможны два варианта: либо мы имеем замкнутую систему взаимно рекурсивных определений, которые не нуждаются ни в каких внешних сущностях, либо мы имеем некоторый набор заранее определенных классификаторов, которые используются как базовые для определения остальных.

Первый подход (абсолютно все определяется в рамках одной системы) кажется соблазнительным, но, к сожалению, он никуда не ведет. Подробное обсуждение этого факта, хотя и поучительное с теоретической точки зрения, увело бы нас слишком далеко от основной темы книги. Мы сошлемся на авторитет: в распространенных языках программирования так не делают.

В UML, также как в распространенных языках программирования и других формальных системах, имеется набор базовых классификаторов, с помощью которых определяются типы элементов модели, в частности типы составляющих других классификаторов. Это типы данных. В модели UML можно использовать три вида типов данных.

- Примитивные типы, которые считаются предопределенными в UML — таковых, как минимум, три: строка, целое число и значение даты/времени. Инструменты вправе расширять этот набор и использовать подходящие названия.
- Типы данных, которые определены в языке программирования, поддерживаемым инструментом. Это могут быть как названия встроенных типов, так и сколь угодно сложные выражения, доставляющие тип, если таковые допускаются языком.
- Типы данных, которые определены в модели. В стандарте UML предусмотрен только один конструктор типов данных: перечислимый тип, который определяется с помощью стереотипа «`enumeration`».[62]

В частности, тип `Boolean` определен в UML как перечислимый тип со значениями `true` и `false`. Приведем пример: допустим, в нашем приложении нужно использовать не обычную двузначную логику, а трехзначную. Тогда соответствующий можно определить так, как показано на рис. 3.26.

[62] По нашему мнению, и этого конструктора не было бы в UML, если бы не нужда в нем при определении метамодели.

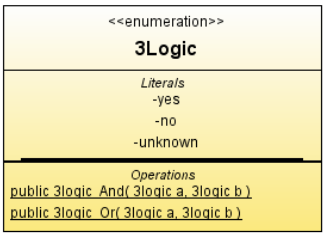



**Рис. 3.26. Перечислимый тип данных «Трехзначная логика»**

Наряду со стандартным стереотипом «`enumeration`» многие инструменты допускают использование стереотипа «`datatype`», который означает построение типа данных с помощью не специфицированного конструктора типов.
Возникает вопрос: чем же типы данных отличаются от прочих классификаторов UML?

*Тип данных* (в UML) — это классификатор, экземпляры которого не обладают индивидуальностью (identity).[63]

Это довольно тонкое понятие, которое мы попробуем объяснить на примере.
Рассмотрим какой-нибудь встроенный тип данных в обычном языке программирования, например, тип `integer` в языке Паскаль. Значения этого типа (экземпляры классификатора) изображаются обычным образом, например, `3`. Что будет, если число "три" используется в нескольких местах программы? Отличатся ли чем-нибудь экземпляры изображения `3`? Очевидно, нет. Экземпляры типа `integer` не обладают индивидуальностью, мы вправе считать, что написанные в разных местах изображения числа `3` суть одно и то же число, а компилятор вправе использовать для хранения представления числа "три" одну и ту же ячейку, сколько бы изображений числа `3` ни присутствовало в программе. Далее, программисту не нужно предусматривать никаких инициализирующих действий, для того, чтобы воспользоваться числом `3` — не нужно определять никаких объектов, не нужно вызывать никаких конструкторов. Можно считать, что все значения типа данных уже определены и всегда существуют, независимо от программы. Более того, с числом "три" ничего не может случиться — чтобы ни происходило в программе, число "три" останется числом "три" и никогда не станет числом "пять".[64]
Сопоставим сказанное с обычным классом, экземпляры которого обладают индивидуальностью. Допустим, в классе `Cinteger` есть только один атрибут, который хранит целое значение. На первый взгляд, такой класс ничем не отличается от типа данных `integer` — экземпляры данного класса вполне можно

[63] Мы используем термин "индивидуальность", хотя семантически он не точен. Более точным является неологизм "самость", но это слово частенько используется в оккультной и мистической литературе и мы воздержимся от его употребления.

[64] В самых ранних реализациях языка Фортран это было не так. Там можно было определить подпрограмму `SUBROUTINE S ( INTEGER X ) X = X + 2 END`, а в другом месте программы вызвать ее `CALL S ( 3 )`, после чего число "три" стало бы числом "пять" со всеми вытекающими ошибками и неприятностями, которые бывало очень трудно обнаружить.

использовать как целые числа. Но это поверхностное впечатление: между типом данных `integer` и классом `Cinteger` много существенных отличий. Во-первых, экземпляры класса `Cinteger` должны быть явно созданы и инициализированы, прежде чем их можно будет использовать в программе. Во-вторых, экземпляр класса `Cinteger`, который в данный момент хранит число "три", через некоторое время выполнения программы может хранить число "пять", оставаясь при этом *тем же самым* экземпляром, поскольку он обладает индивидуальностью. В-третьих, в программе может быть определено несколько экземпляров класса `Cinteger`, которые хранят одно и то же число "три" и это будут разные объекты (компилятор разместит их в разных областях памяти), поскольку они обладают индивидуальностью.

Отсутствие индивидуальности[65] экземпляров типа данных влечет некоторые общепринятые ограничения на операции типа данных.

- Было бы нелепо, если бы операция сложения для числа "три" работала бы по иному алгоритму, нежели операция сложения для числа "пять". Поэтому областью действия операций типа всегда является классификатор, а не экземпляр (в модели они подчеркнуты, см. рис. 3.26).
- Естественно считать, что операции типа данных не имеют побочных эффектов, т. е. их применение не меняет состояния системы. В принципе можно допустить, что операция сложения чисел помимо вычисления значения суммы делает какое-то невидимое волшебное действие, например, меняет значение какой-то глобальной переменной. Но такие операции, как нам кажется, не дают ничего, кроме ненужных сложностей и трудностей.[66] Поэтому операции типа данных всегда обладают свойством `isQuery`.
- Типы данных и их операции — это базовые, элементарные конструкции языков программирования. Разумно предположить, что они реализованы предельно эффективно.[67] С точки зрения моделирования их выполнение можно считать мгновенным и атомарным действием. Довольно странно требовать от операции сложения, чтобы она обладала способностью параллельно и одновременно со сложением одной пары чисел складывать и другие пары. Поэтому операции типов данных считаются не повторно входимыми и обладают свойством `sequential`.

Есть еще одно замечание относительно операций типов данных, которое, однако не является общепринятым, а отражает авторские предпочтения. Операции типа данных принадлежат типу в целом, а не отдельным экземплярам (значениям) типа. Поэтому мы считаем целесообразным явным образом передавать в качестве аргументов все объекты, над которыми выполняется операция типа данных, и не

---

[65] В этом контексте ясно видно, почему слово "индивидуальность" не является в данном случае семантически точным переводом термина identity. Каждое значение типа данных обладает индивидуальностью в общепринятом смысле этого слова: число "три" это не число "пять".

[66] В нашем распоряжении есть только один очень частный контр пример: операция задержки типа данных дата/время, которая ничего не вычисляет, а имеет побочный эффект в виде строго определенного времени своего выполнения. Но исключения только подтверждают правило.

[67] Для обычных типов данных языков программирования и распространенных архитектур компьютера операция — это, как правило, одна машинная команда.

использовать объект `this`. С нашей точки зрения выражение `or(a,b)` лучше выражения `a.or(b)`. Поэтому операции на рис. 3.26 имеют по два параметра, а не по одному.[68]

### *3.2.9. Шаблоны*

Еще одной сущностью, специфической для диаграмм классов, являются шаблоны. *Шаблон* — это класс с параметрами. Параметром может быть любой элемент описания класса — тип составляющей, кратность атрибута и т. д. На диаграмме шаблон изображается с помощью прямоугольника класса, к которому в правом верхнем углу присоединен пунктирный прямоугольник с параметрами шаблона. Описания параметров перечисляются в этом прямоугольнике через запятую. Описание каждого параметра имеет вид:

`ИМЯ : тип`

Сам по себе шаблон не может непосредственно использоваться в модели. Для того, чтобы на основе шаблона получить конкретный класс, который может использоваться в модели, нужно указать явные значения аргументов. Такое указание называется *связыванием*. В UML применяются два способа связывания:

- *явное связывание* — зависимость со стереотипом «`bind`», в которой указаны значения аргументов;
- *неявное связывание* — определение класса, имя которого имеет формат

  `имя_шаблона < аргументы >`

Рассмотрим пример (рис. 3.27). Здесь определен шаблон `Array`, имеющий два параметра: `n` типа `Integer` и `T`, тип которого не указан. Этот шаблон применяется для создания массивов определенной длины, содержащих элементы определенного типа. В данном случае с помощью явного связывания определен класс `Triangle` как массив из трех элементов типа `Point`. С помощью неявного связывания определен аналогичный по смыслу класс с именем `Array<3,Point>`.

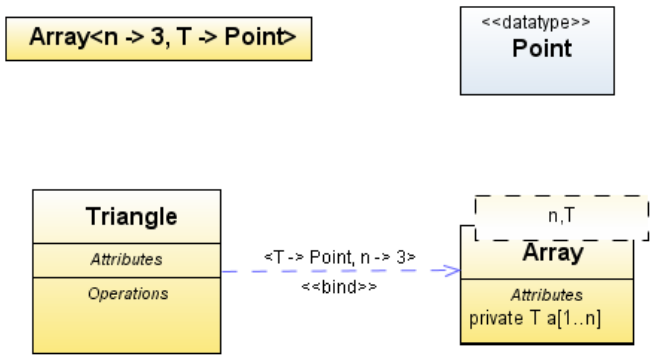



**Рис. 3.27. Связывание шаблона: неявное (вверху), явное (внизу)**

Назначение и область применения шаблонов понятны — шаблоны нужны, чтобы определить некоторую общую параметрическую конструкцию класса один раз, и

[68] Если бы это был обычный класс, а не тип данных, наши операции имели бы по одному параметру.

затем использовать ее многократно, подставляя конкретные значения аргументов. Явное связывание более наглядно, неявное связывание менее наглядно, зато записывается короче.

Нам хочется закончить обсуждение диаграмм классов — важнейшего средства описания структуры в UML — серией элементарных советов по практическому моделированию. Как видно из предыдущих разделов, диаграммы классов содержат множество деталей. Для практически значимых систем диаграммы классов в конечном итоге получаются довольно сложными. Пытаться прорисовать сложную диаграмму классов сразу "на всю глубину" нерационально — слишком велик риск "утонуть" в деталях. Мы полагаем, что удачная модель структуры сложной системы создается за несколько (может быть даже за несколько десятков) итераций, в которых моделирование структуры перемежается моделированием поведения. Вот наши советы.

- Описывать структуру удобнее параллельно с описанием поведения. Каждая итерация должна быть небольшим уточнением как структуры, так и поведения.
- Не обязательно включать в модель все классы сразу. На первых итерациях достаточно идентифицировать очень небольшую (10%) долю всех классов системы.
- Не обязательно определять все свойства класса сразу. Начните с имени — операции и атрибуты постепенно выявятся в процессе моделирования поведения.
- Не обязательно показывать на диаграмме все свойства класса. В процессе работы диаграмма должна легко охватываться одним взглядом.
- Не обязательно определять все отношения между классами сразу. Пусть класс на диаграмме "висит в воздухе" — ничего с ним не случится.

## 3.3. Диаграммы реализации

*Диаграммы реализации* — это обобщающее название для диаграмм компонентов и диаграмм размещения. Название объясняется тем, что данные диаграммы приобретают особую важность на позднейших фазах разработки — на фазах реализации и перехода. На ранних фазах разработки — анализа и проектирования — эти диаграммы либо вообще не используются, либо имеют самый общий, не детализированный вид. Диаграммы компонентов и размещения имеют больше общего, чем различий, поскольку предназначены для моделирования двух теснейшим образом связанных структур:

- структуры компонентов в приложении;
- структуры используемых вычислительных ресурсов.

Мы начнем с диаграмм компонентов, как более общей конструкции, а затем рассмотрим, какие дополнительные (по сравнению с диаграммами компонентов) специальные средства применяются на диаграммах размещения.

### *3.3.1. Диаграмма компонентов*

*Диаграмма компонентов* предназначена для перечисления и указания взаимосвязей артефактов моделируемой системы (см. разд. 1.5.9).

На диаграмме компонентов применяются следующие основные типы сущностей:

- компоненты;
- интерфейсы;
- классы;
- объекты.

На диаграмме компонентов обычно используются отношения следующих типов:

- зависимость;
- ассоциация (главным образом в форме композиции);
- реализация.

Основные элементы нотации диаграмм компонентов приведены на рис. 1.14. Прежде чем переходить к более детальным примерам, нам необходимо обсудить основную сущность используемую на диаграммах данного типа, а именно — компонент.

### *3.3.2. Компонент*

| Компонент — это физически существующий и заменяемый артефакт системы. |
|---|

Это определение подчеркивает основную прагматику, на которую авторы UML, видимо, ориентировались при разработке семантики диаграмм компонентов. Речь идет о компонентом программировании.

#### Компонентное программирование

Идея *компонентного* или *сборочного* программирования также стара как само программирование. Действительно затраты на тиражирование программ любой величины и сложности неизмеримо малы по сравнению с затратами на их разработку. Затратами на тиражирование можно просто пренебречь.[69] Отсюда возникает естественное желание не создавать каждую новую программу "с нуля", а использовать ранее созданные программы в качестве готовых компонентов, благо затраты на копирование столь невелики. Оставляя в стороне экономические вопросы, связанные с правами собственности, сосредоточимся на техническом аспекте: насколько реально компонентное программирование? Допустим, нужные по функциональности программные компоненты доступны для копирования: как можно их использовать при разработке новой программы? Ответ на этот вопрос имеет огромное значение, прежде всего экономическое. Для решения данного вопроса были предприняты беспрецедентные усилия, особенно в последние годы. Наша оценка текущего состояния данного вопроса состоит из трех утверждений. Первое: компонентное программирование является непреложным требованием практики, без него прогресс в индустрии программного обеспечения невозможен. Второе: в последние годы в этой области наблюдается значительный прогресс, причем налицо тенденция к ускорению. Третье: современные технологии компонентного программирования все еще далеки от идеала и не в полной мере отвечают требованиям практики. Поясним наше видение следующей аналогией. Автомобиль можно

[69] Это очень важная особенность индустрии программного обеспечения. Действительно, в других отраслях промышленности, затраты на производство экземпляра продукта, хотя и меньше затрат на проектирования, но все же не могут рассматриваться как пренебрежимо малые.

> рассматривать состоящим из компонентов: двигатель, трансмиссия, ходовая часть, кузов… Можно ли собрать автомобиль из готовых компонентов? Ответ: да, если компоненты были изготовлены в расчете на то, что из них будет собрана конкретная модель автомобиля с применением конкретной технологии сборки. В противном случае сборка проблематична. Однако в каждом достаточно большом гаражном кооперативе можно найти умельца, который с помощью хорошего набора ручных инструментов, природной смекалки и технического чутья может собрать нечто, способное двигаться, из самых неожиданных компонентов. Пока что, по нашему мнению, работа программистов, использующих готовые компоненты, больше напоминает шаманство гаражного умельца, нежели современное автоматизированное производство.[70]

Что же является компонентом в смысле UML? Прежде всего, это компонент в смысле сборочного программирования: особым образом оформленная часть программного кода, которая может работать в составе более сложной программы. К сожалению, такое определение слишком расплывчато: отдельная строка исходного кода также может рассматриваться как компонент, но это, видимо, не совсем то, чего мы хотим. На самом деле абсолютно формальное определение компонента в UML дать невозможно. При выделении компонентов применяются следующие неформальные критерии.

- Компонент нетривиален. Это нечто более сложное и объемное, чем фрагмент кода или одиночный класс.
- Компонент независим, но не самодостаточен. Он содержит все, что нужно для функционирования, но предназначен для работы во взаимодействии с другими компонентами.
- Компонент однороден. Он выполняет несколько взаимосвязанных функций, которые могут быть естественным образом охарактеризованы как единое целое в контексте более сложной системы.
- Компонент заменяем. Он поддерживает строго определенный набор интерфейсов и может быть без ущерба для функционирования системы заменен другим компонентом, поддерживающим те же интерфейсы.

Компоненты понимаются в UML в наиболее общем смысле: это не только исполнимые файлы с кодами программы, но и исходные тексты программ, веб-страницы, справочные файлы, сопроводительные документы, файлы с данными и вообще любые артефакты, которые тем или иным способом используются при работе приложения и входят в его состав.

Для того чтобы как-то отражать такое разнообразие типов артефактов, являющихся компонентами, в UML предусмотрены стандартные стереотипы компонентов перечисленные в таблице 3.9. Помимо стандартных стереотипов, многие инструменты поддерживают дополнительные стереотипы компонентов, часто со

---

[70] Некоторые авторские соображения о влиянии UML на компонентное программирование изложены в главе 6.

специальными значками и фигурами,[71] обеспечивающими высокую наглядность диаграмм компонентов.

**Таблица 3.9.** Стандартные стереотипы компонентов

| Стереотип | Описание |
|---|---|
| «executable» | Выполнимая программа любого вида. Подразумевается по умолчанию, если никакой стереотип не указан. |
| «document» | Документ любого типа, например, файл с документацией к программе. |
| «file» | Файл с исходным кодом программы или с данными, которые программа использует. |
| «library» | Статическая или динамическая библиотека. |
| «table» | Таблица базы данных. |

Нам осталось рассмотреть последний аспект, относящийся к компонентам в UML. В терминологии UML компонент — это классификатор, т. е. дескриптор, описывающий множество однотипных объектов, и как всякий классификатор, компонент может иметь экземпляры. Такой взгляд на компоненты может показаться несколько надуманным, но он имеет свое обоснование в необыкновенной легкости копирования электронных артефактов, о которой мы говорили выше. Действительно, если на компакт-диске с поставляемой системой записан, например, файл документации, то можно считать, что это описание множества файлов с документацией, потому что при развертывании системы этот файл может быть легко скопирован на все рабочие станции в необходимом числе экземпляров. Поскольку компонент — это классификатор, его имя, в соответствии с общими механизмами UML (см. разд. 1.7.3), не подчеркивается на диаграмме. Заметим, что копии электронных артефактов неотличимы друг от друга и можно считать, что они не обладают индивидуальностью в том смысле, как это определено в разд. 3.2.8. Однако это не всегда так: бывают ситуации, когда копии электронных артефактов обладают ярко выраженной индивидуальностью. Приведем пару характерных примеров. Первый пример заимствован из книги [2]. Рассмотрим типичный компонент многих систем — блок проверки орфографии. Внутри такого компонента есть словарь, с помощью которого проверяется правописание. Если этот словарь неизменяемый, то экземпляры блока орфографического контроля не обладают индивидуальностью. Но блок проверки орфографии может быть снабжен функцией пополнения и изменения словаря. В таком случае, в процессе эксплуатации и корректировки со стороны пользователя конкретный экземпляр блока орфографического контроля все более и более отличается от своих собратьев, работающих на других компьютерах и приобретает индивидуальность. Второй пример связан с одном из способов защиты от несанкционированного копирования программ (пиратства). Суть этого способа заключается в так называемой привязке кода к компьютеру. При установке

[71] К сожалению, трафарет Visio 2000, с помощью которого подготовлены некоторые диаграммы, приведенные в этой книге, не содержит рекомендуемых значков для стандартных стереотипов компонентов, поэтому мы их не приводим в табл. 4.9.

программы определяются некоторые индивидуальные характеристики компьютера и код устанавливаемой программы изменяется таким образом, чтобы он был работоспособен только на данном компьютере. Очевидно, что привязанные к компьютеру экземпляры компонентов обладают индивидуальностью. Синтаксис UML позволяет четко разграничить в модели эти случаи. Для этого достаточно подчеркнуть имя компонента, обладающего индивидуальностью. Тем самым, в полном соответствии с общими правилами UML (см. разд. 1.7), мы укажем, что речь идет о конкретном экземпляре компонента, обладающим индивидуальностью.

Мы закончим этот раздел примером из информационной системы отдела кадров. Какие компоненты разумно выделить в этой системе? Основное назначение системы — хранить данные о кадрах и выполнять по указанию пользователя некоторые операции с этими данными. Анализирую состав операций, мы видим, что они сводятся к созданию, модификации и удалению хранимых элементов данных. Стандартным решением в таких ситуациях является применение готовой СУБД для управления данными. С точки зрения проектирования информационной системы отдела кадров целесообразно считать используемую СУБД готовым компонентом. Мы можем не заострять внимания на структуре этого компонента — она стандартна и, наверное, достаточно хорошо описана вне нашей модели. Нам достаточно определить интерфейс, с помощью которого осуществляется взаимодействия. Подавляющее большинство СУБД поддерживают интерфейс ODBC, и мы можем предположить, что такова же и используемая нами система управления базам данных.

### Открытый доступ к базам данных

Стандартный метод доступа к базам данных, разработанный корпорацией Microsoft и получивший название open database connectivity (ODBC). В большинстве случаев доступ к данным при помощи ODBC включает три этапа: 1) установка драйвера базы данных, специфического для данной СУБД; 2) создание источника данных ODBC и установление соединения с источником данных; 3) использование инструкций языка SQL для доступа к источнику, с которой установлено соединение. Поскольку драйверы ODBC поставляются практически для всех СУБД, приложение может использовать данные из самых разнообразных источников с помощью одного стандартного интерфейса ODBC.

СУБД возьмет на себя все функции по непосредственному манипулированию данными: создание, удаление и поиск записей в таблицах и т. д. Реализация операций нашей информационной системы отдела кадров сводится к некоторой последовательности элементарных операций с данными. Например, операция перевода сотрудника с одной должности на другую, видимо, потребует изменения в трех элементах данных: в данных о сотруднике и в данных о старой и новой должностях. Однако, целесообразно ли считать, что определение и выполнение самой последовательности элементарных операций с данными также является прерогативой выделенного нами компонента — СУБД. Общепринятая практика отвечает на этот вопрос отрицательно. По многим причинам (см. разд. 3.1.1) лучше

выделить это в отдельный компонент, обычно называемый бизнес-логикой.[72] Наконец, по причинам, указанным выше (см. про 3.1.3), мы должны предположить, что в нашей системе должен быть компонент, ответственный за пользовательский интерфейс. В первом приближении мы приходим к структуре компонентов, приведенной на рис. 3.28, которая называется «трехуровневая архитектура» (см. разд. 3.1.1).

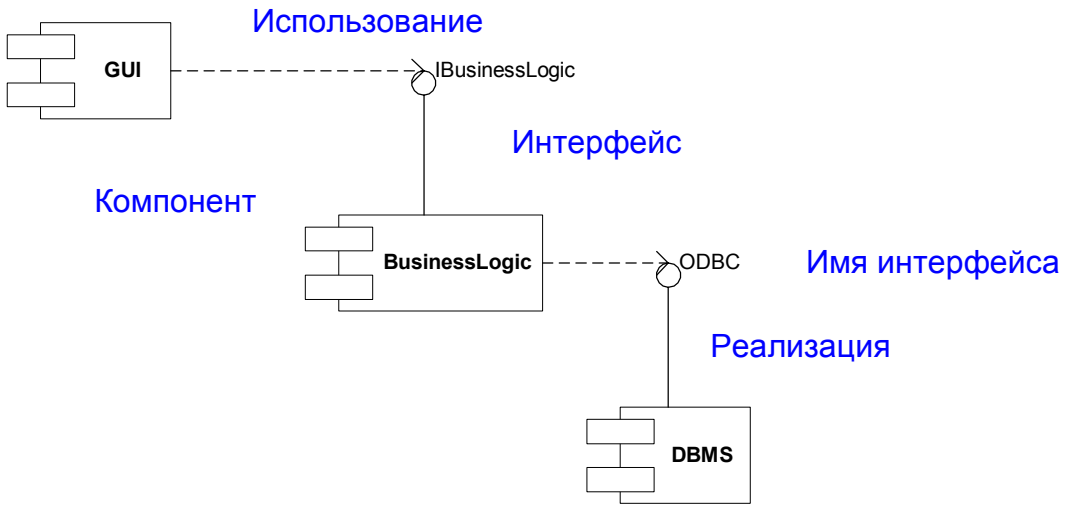



**Рис. 3.28. Диаграмма компонентов**

### *3.3.3. Применение диаграмм компонентов*

Мы переходим к обсуждению применения диаграмм компонентов после того, как привели примеры их применения — причина состоит в том, что здесь мы хотим обсудить такие применения диаграмм компонентов, для которых информационной системы отдела кадров не может послужить примером.

Начнем с элементарного соображения: в случае разработки "монолитного" настольного приложения диаграмма компонентов не нужна — она оказывается тривиальной и никакой полезной информации не содержит. Таким образом, диаграммы компонентов применяются только при моделировании многокомпонентных приложений.

Если приложение поставляется в виде "конструктора" (набора "кубиков" — компонентов) из которого при установке собирается конкретный уникальный экземпляр приложения, то диаграммы компонентов оказываются просто незаменимым средством. Действительно, многие современные приложения, особенно развитые системы автоматизации управления делопроизводством предприятия, поставляются в виде большого (десятки и сотни) набора компонентов, из которых "на месте" собирается нужная пользователю, часто

---

[72] Крайне неудачный, но часто используемый термин, являющийся калькой английского business logic. Бизнес-логика не имеет никакого отношения ни к бизнесу (в российском понимании этого слова), ни к логике. Правильнее было бы использовать сложное словосочетание "правила обработки данных", но мы боимся оказаться непонятыми.

уникальная конфигурация. Некоторые авторы рекомендуют использовать диаграммы компонентов для управления конфигурацией не только на фазе поставки и установки программного обеспечения, но и в процессе разработки: для отслеживания версий компонентов, вариантов сборки и т. п.

При разработке приложений, которые должны взаимодействовать с так называемыми *унаследованными* приложениями и данными, без диаграмм компонентов также трудно обойтись. Дело в том, что фактически единственным средством UML, позволяющим как-то описать и включить в модель унаследованные приложения и данные являются компоненты (и их интерфейсы). Сюда же относится случай моделирования физической структуры данных в базе, работать с которой требуется помимо "родной" СУБД.

Последним (в нашем списке) примером применения диаграмм компонентов является моделирование систем динамической архитектуры, то есть таких систем, которые меняют состав и количество своих компонентов во время выполнения. Например, многие приложения Интернета меняют свою конфигурацию во время выполнения в зависимости от текущей нагрузки.

### *3.3.4. Диаграммы размещения*

Последним структурным аспектом, который необходимо обсудить, является описание размещения компонентов относительно участвующих в работе вычислительных ресурсов. В UML для этой цели предназначены диаграммы размещения. В UML 2.0 эти диаграммы переименованы в *диаграммы развертывания*. Если речь идет о настольном приложении, которое целиком хранится и выполняется на одном компьютере, то отдельная диаграмма размещения не нужна — достаточно диаграммы компонентов (а может быть, и без нее можно обойтись). При моделировании распределенных приложений значение диаграмм размещения резко возрастает: они являются описанием топологии[73] развернутой системы.

На диаграмме размещения, по сравнению с диаграммами компонентов, применяются только один дополнительный тип сущности — узел и два дополнительных отношения: ассоциация между узлами и размещение компонента на узле. В остальном диаграммы размещения наследуют возможности диаграмм компонентов.

*Узел* — это физический вычислительный[74] ресурс, участвующий в работе системы. Компоненты системы во время ее работы размещаются на узлах. В UML узел является классификатором, т. е. мы можем (и должны!) различать описание типа вычислительного ресурса (например, рабочая станция, последовательный порт) и описание экземпляра вычислительного устройства (например, устройство COM1 типа последовательный порт). Данное различие моделируется согласно общему

---

[73] Программисты заимствовали название раздела математики (топология) как термин. Например, часто можно встретить выражение "топология локальной сети". Нельзя сказать, что такое заимствование совершенно неверно, но в то же время оно и не совсем по существу. Речь идет просто об описании структуры связей конечного множества узлов, т. е. о графе (см. разд 1.4).

[74] При использовании UML в других предметных областях, узлом может быть не только компьютер, но и другой объект: человек, механическое устройство и т. д.

механизму UML (разд. 1.7): имя экземпляра узла подчеркивается, а имя типа узла — нет. На диаграмме узел представляется фигурой, изображающей прямоугольный параллелепипед.

Ассоциация между узлами означает то же, что и в других контекстах: возможность обмена сообщениями. Применительно к вычислительным сетям ассоциация означает наличие канала связи. Если нужно указать дополнительную информацию о свойствах канала, то это можно сделать используя общие механизмы: стереотипы, ограничения и именованные значения, приписанные ассоциации.

Размещение компонента на узле, как правило, изображают, помещая фигуру компонента внутрь фигуры узла. Если это по каким-либо причинам неудобно, то отношение размещения можно передать отношением зависимости от узла к компоненту.

**ЗАМЕЧАНИЕ**

> Отношения между компонентами, интерфейсами, классами и объектами, которые показываются на диаграмме компонентов, можно показывать и на диаграмме размещения. При этом, вообще говоря, пересечение линий этих отношений границ фигуры узла ничего не означает, но принято избегать таких пересечений, если это возможно.

Приведем пример из информационной системы отдела кадров. Допустим, что мы приняли архитектуру, приведенную на рис. 3.28. Сколько компьютеров будет использоваться при работе данного приложения? На этот вопрос нужно отвечать также вопросом: а сколько пользователей будет у системы и сколько из них будут работать с приложением одновременно? Если имеется только один пользователь (или, хуже того, нашу систему установят "для галочки", а использовать не будут), то проблем нет — настольное приложение — один компьютер и диаграмма размещения не нужна. Допустим, что у нашей системы должно быть много пользователей и они могут работать одновременно. Тогда ответ очевиден: узлов должно быть не меньше, чем число одновременно работающих пользователей, потому что вдвоем за одним компьютером обычным пользователям работать неудобно.[75] Скорее всего, узлов должно быть на единицу больше чем пользователей, т. к. в большинстве организаций есть специально выделенный компьютер (сервер) для хранения корпоративных данных, за которым никто не работает. Там мы и разместим нашу базу данных, в расчете на то, что нужная СУБД, скорее всего, на сервере уже установлена. Остается вопрос о размещении компонентов, реализующих бизнес-логику. Здесь возможны разные варианты: на компьютере пользователя, на промежуточной машине (сервере приложений), на корпоративном сервере баз данных. Если мы остановимся на последнем варианте (который на жаргоне называется "архитектура клиент/сервер с тонким клиентом"), то получим диаграмму, приведенную на рис. 3.29.

---

[75] Для поклонников экстремального программирования сделаем оговорку: экстремальные программисты не являются подклассом класса пользователей.

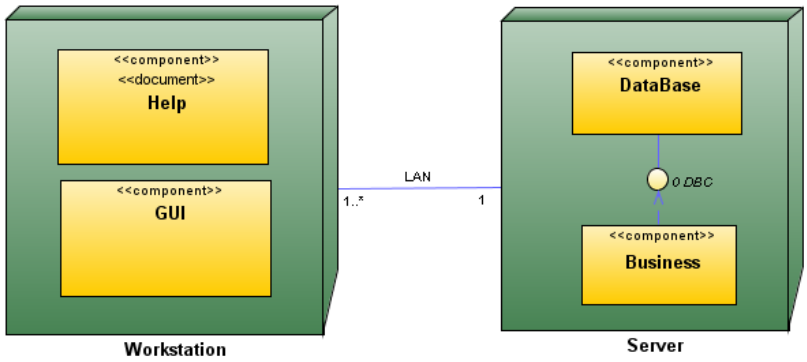



**Рис. 3.29. Диаграмма размещения информационной системы отдела кадров**

## 3.4. Выводы

- Структура сложной системы описывается на уровне дескрипторов.
- Диаграммы классов моделируют структуру объектов и связей между ними.
- Классы выбираются на основе анализа предметной области, взаимного согласования элементов модели и общих теоретических соображений.
- Диаграмма объектов — это пример связей программных объектов в отдельный момент выполнения системы.
- Диаграммы компонентов моделируют структуру компонентов (артефактов) и взаимосвязей между ними.
- Диаграммы размещения моделируют структуру вычислительных ресурсов и размещение компонентов.

# Тема 4. Моделирование поведения

- Что такое моделирование поведения?
- Как может статическая модель описывать динамическое поведение?
- Что такое конечный автомат?
- Какие элементы применяются для моделирования поведения?
- Для чего применяются диаграммы состояний?
- Что общего и в чем различие диаграмм состояний и деятельности?
- Какова область применения диаграмм взаимодействия?
- Как моделируются параллельные процессы?

## 4.1. Объектно-ориентированное моделирование поведение

При создании программной системы недостаточно ответить на вопросы "что делает система?" (глава 2) и "из чего она состоит?" (глава 3) — требуется ответить на вопрос "как работает система?". Ответ на этот вопрос называется (в UML) *моделью поведения*. Поведение реальной программной системы целиком и полностью определяется кодом ее программы — как программа составлена, так она и выполняется (с точностью до сбоев) — "от себя" компьютер ничего не придумывает. Таким образом, модель поведения — это описание алгоритма работы системы.

Существует множество способов описания алгоритмов, каждый из них имеет свои достоинства и недостатки и предназначен для применения в различных ситуациях. Например, при описании алгоритмов, которые предназначены для выполнения компьютером, используются языки программирования, но для описания алгоритмов, выполняемых человеком, языки программирования неудобны и применяются другие способы.

Средства моделирования поведения в UML, ввиду разнообразия областей применения языка, должны удовлетворять набору различных и частично противоречивых требований. Перечислим некоторые из них.

- Модель поведения должна быть достаточно детальной для того, чтобы послужить основой для составления компьютерной программы — компьютер не сможет самостоятельно "додумать" опущенные детали.
- Модель поведения должна быть компактной и обозримой, чтобы служить средством общения между людьми в процессе разработки системы и для обмена идеями.
- Модель поведения не должна зависеть от особенностей реализации конкретных компьютеров, средств программирования и технологий, чтобы не сужать область применения языка UML.
- Средства моделирования поведения в UML должны быть знакомы и привычны большинству пользователей языка и не должны противоречить требованиям наиболее ходовых парадигм программирования.

Удовлетворить сразу всем требованиям в полной мере, видимо, практически невозможно — средства моделирования поведения UML являются результатом многочисленных компромиссов. Многие авторы критикуют UML за то, что он недостаточно хорош с точки зрения того или иного конкретного критерия.[76] Но

[76] Мы тоже по ходу дела стараемся отмечать слабые, по нашему мнению, конструкции UML.

если принять во внимание сразу все противоречивые требования, то, по нашему мнению, следует признать, что на сегодняшний день UML является решением, очень близким к оптимальному. В будущем, по мере развития теории и практики программирования, будет эволюционировать и UML — для этого в языке предусмотрены все необходимые средства.

Описание средств моделирования поведения в UML мы начнем с небольшого отступления на тему одного из разделов дискретной математики — теории автоматов — который послужил теоретической основой средств моделирования поведения в UML.

### *4.1.1. Конечные автоматы*

*Конечным автоматом* называется совокупность пяти объектов:

1. конечного множества $A=\{a_1,\ldots,a_n\}$, называемого *входным алфавитом*; элементы множества $A$ называются входными символами, сигналами или стимулами;
2. конечного множества $Q=\{q_1,\ldots,q_m\}$, называемого *алфавитом состояний*; элементы множества $Q$ называются состояниями;
3. конечного множества $B=\{b_1,\ldots,b_k\}$, называемого *выходным алфавитом*; элементы множества $B$ называются выходными символами;
4. тотальной функции $\delta : A \times Q \to Q$, называемой *функцией переходов*;
5. тотальной функции $\lambda : A \times Q \to B$, называемой *функцией выходов*.

Эта несложная конструкция оказывается применимой для адекватного описания очень многих ситуаций. Пусть задан автомат $S$, некоторое состояние $q$ этого автомата и входное слово $\alpha$ в алфавите $A$. По этой информации однозначно определяется выходное слово $\beta$ в алфавите $B$. А именно, по состоянию $q$ и первому символу слова $\alpha$ с помощью функции выходов $\lambda$ определяется первый символ слова $\beta$ и с помощью функции переходов $\delta$ определяется следующее состояние автомата. Затем по новому состоянию и второму символу входного слова $\alpha$ и текущему состоянию определяется второй символ выходного слова $\beta$ и следующее состояние. И так далее. Поскольку функции $\lambda$ и $\delta$ тотальны, автомат $S$ и состояние $q$ определяют некоторый алгоритм преобразования слов в алфавите $A$ в слова в алфавите $B$. Так вот, оказывается, что класс алгоритмов, которые можно описать автоматом, весьма широк, хотя и не всеобъемлющ.

Обычно при применении конечных автоматов используют некоторые дополнительные соглашения, которые не меняют сути дела и принимаются для удобства. Во-первых, сразу выделяют одно состояние, которое считается начальным (обычно это $q_1$). Автомат с выделенным начальным состоянием называется *инициальным*. Инициальный автомат всегда начинает работу в одном и том же состоянии, поэтому его указывать не нужно. Во-вторых, иногда выделяют некоторое подмножество состояний, которые называются заключительными. Если автомат переходит в заключительное состояние, то работа алгоритма преобразования завершается (хотя во входном слове, может быть, еще остались нерассмотренные символы). В-третьих, можно рассмотреть случай, когда функция

выходов λ имеет только один параметр — состояние — и выходной символ зависит только от состояния и не зависит от входного символа. Такой автомат называется автоматом Мура (а общий случай называется автоматом Мили). Очевидно, что автомат Мура является частным случаем автомата Мили. Более того, нетрудно показать, что введением дополнительных состояний любой автомат Мили может быть преобразован в эквивалентный автомат Мура. Наконец, можно не рассматривать выходной алфавит и функцию выходов: автомат работает ("читает входное слово") до тех пор, пока не окажется в одном их заключительных состояний. Результатом работы алгоритма считается заключительное состояние. Такой автомат называется распознавателем.

Перечисленное в предыдущем абзаце не исчерпывает список возможных модификаций конечных автоматов. Например, можно рассматривать сети взаимодействующих конечных автоматов: выходные символы одного автомата являются входными символами других автоматов. Или же состояния одного автомата являются входными символами других автоматов. Количество рассмотренных (и детально исследованных!) вариаций на тему конечных автоматов весьма велико — их обзор не является предметом данной книги. Нас интересует, почему теория конечных автоматов была использована авторами UML в качестве средства моделирования поведения. По нашему мнению, основных причин три:

- теоретическая;
- историческая;
- практическая.

Теоретическая причина интереса к конечным автоматам и их модификациям заключается в следующем. Для всех универсальных способов задания алгоритмов (моделей вычислимости) справедлива теорема Райса: все нетривиальные свойства вычислимых функций алгоритмически неразрешимы. Поясним формулировку этой теоремы. Вычислимой называется функция, для которой существует алгоритм вычисления. Нетривиальным называется такое свойство функции, для которого известно, что существуют как функции обладающие данным свойством, так и функции им не обладающие. Массовая проблема называется алгоритмически неразрешимой, если не существует единого алгоритма ее решения для любых исходных данных. Например, "быть периодической" — нетривиальное свойство функции. Теорема Райса утверждает, что не существует такого алгоритма, который по записи алгоритма вычисления функции определял бы, периодическая эта функция или нет. В частности, алгоритмически неразрешимыми являются все интересные вопросы о свойствах алгоритмов, например:

- проблема эквивалентности: имеются две записи алгоритмов, вычисляют они одну и ту же функцию или нет?
- проблема остановки: имеется запись алгоритма, завершает этот алгоритм свою работу при любых исходных данных или нет?

Этот список можно продолжать неограниченно: первое слово в теореме Райса — "все". Значит ли это, что вообще все неразрешимо и про алгоритмы ничего нельзя узнать? Разумеется, нет. Если дана конкретная запись алгоритма, то путем ее индивидуального изучения, скорее всего, можно установить наличие или отсутствие нужных свойств. Речь идет о том, что невозможен единый метод, пригодный для всех случаев.

Обратите внимание, что теорема Райса справедлива в случае использования любой, но универсальной модели вычислимости (способа записи алгоритмов). Если же используется не универсальная модель вычислимости, то теорема Райса не имеет места. Другими словами, существуют подклассы алгоритмов, для которых некоторые важные свойства оказываются алгоритмически разрешимыми. Конечные автоматы являются одним из важнейших таких подклассов. С одной стороны, данный формализм оказывается достаточно богатым, чтобы выразить многие практически нужные алгоритмы (примеры см. ниже), с другой стороны, целый ряд свойств автоматов можно проверить автоматически, причем найдены весьма эффективные алгоритмы. В частности, проблемы эквивалентности и остановки для автоматов эффективно разрешимы. В случае использования универсального языка программирования мы не можем автоматически убедиться в правильности программы: нам остается только упорно тестировать ее в смутной надежде повысить свою уверенность в надежности программы. В случае же использования конечных автоматов многое можно сделать автоматически, надежно и математически строго — поэтому теория конечных автоматов столь интересна для разработки программных систем.

Историческая причина популярности конечных автоматов состоит в том, что данная техника развивается уже достаточно давно и теоретические результаты были с успехом использованы при решении многих практических задач. В частности, системы проектирования, спецификации и программирования, основанные на конечных автоматах и их модификациях, активно развиваются и применяются уже едва ли не полвека. Особенно часто конечные автоматы применяются в конкретных предметных областях, где можно обойтись без использования универсальных моделей вычислимости. В результате очень многие пользователи, инженеры и программисты хорошо знакомы с конечными автоматами и применяют их без затруднений. Мы не готовы дать исчерпывающий обзор предметных областей, где применяются конечные автоматы и ограничимся одним достаточно показательным примером. В области телекоммуникаций уже лет 15 активно применяется промышленный стандарт — язык спецификации и описания алгоритмов SDL (Specification and Description Language).

### Язык SDL

Первые варианты языка появились около 15 лет назад, в разработке принимали участие крупнейшие компании, производящие телекоммуникационное оборудование. В настоящее время SDL является промышленным стандартом, который поддерживает ITU (International Telecommunications Union) — международная организация, определяющая стандарты в области телекоммуникаций. В основу SDL положены три основные идеи: структурная декомпозиция, описание поведения систем с помощью конечных автоматов и использование аксиоматически определенных абстрактных типов данных. Конечные автоматы описываются с помощью диаграмм состояний-переходов (см. ниже рис. 4.2), которые обогащены целым рядом дополнительных возможностей. На дуге перехода могут быть представлены различные специальные действия: получение и отправка сигналов, изменение значений переменных, проверка условий и др. На рис. 4.1 приведен

пример диаграммы SDL[77] для описания порядка работы обычного телефонного аппарата при обработке входящего звонка.

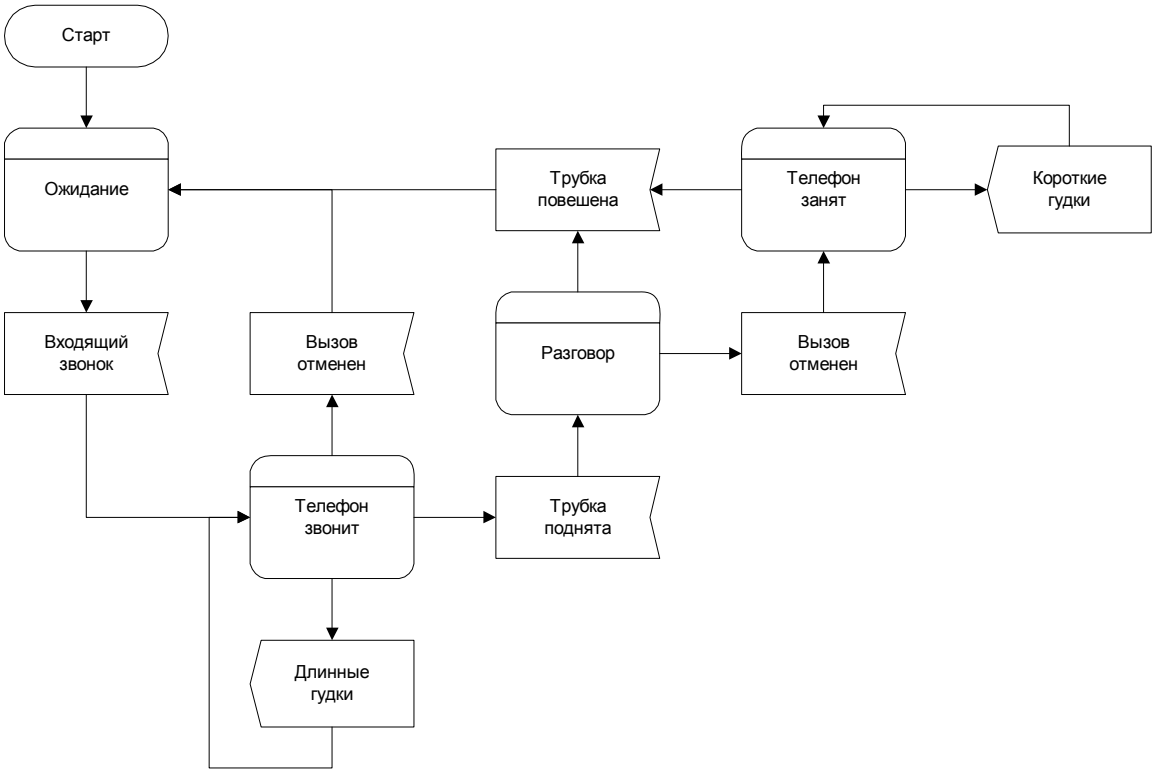



**Рис. 4.1. Диаграмма SDL — обработка входящего телефонного звонка**

SDL с самого начала проектировался и использовался как средство, ориентированное на конкретные практические приложения. Инструменты, поддерживающие SDL, не только позволяют рисовать красивые картинки, но и обеспечивают верификацию моделей, например, выявление взаимных блокировок (тупиков) или недостижимых состояний, имитационное моделирование с целью выявления временных характеристик, автоматическую генерацию кода, в том числе для специальной аппаратуры. Многие промышленные компании используют эти инструменты при проектировании своих изделий.

Нужно сказать, что SDL оказал значительное влияние на средства моделирования поведения UML, что отмечают сами авторы языка.

Еще одна историческая причина популярности конечных автоматов состоит в том, что разработано несколько вариантов нотации для записи конечных автоматов, которые оказались весьма наглядными и удобными. Оставляя в стороне формульную запись и другие математические приемы, приведем два примера способов описания конечных автоматов, которые с первого взгляда понятны буквально любому человеку.

[77] Мы использовали вариант нотации, который поддерживается имеющимся трафаретом Visio. Он несколько не соответствует последней версии стандарта SDL, но отличия не принципиальны.

Первый способ — табличный. Автомат записывается в форме таблицы, столбцы которой помечены символами входного алфавита, строки помечены символами алфавита состояний, а в ячейках таблицы записаны соответствующие значения функций δ и λ. Рассмотрим пример, речь в котором пойдет об автомате, который вычисляет следующее натуральное число по его изображению в позиционной двоичной системе счисления. Входной и выходной алфавиты состоят из символов 0, 1 и s (пробел). Для упрощения примера будем считать, что двоичные цифры записи числа поступают на вход автомата в обратном порядке — справа налево. Автомат имеет три состояния, которые для ясности мы назовем "начальное", "перенос" и "копирование". Сопоставьте табл. 4.1 с определением, приведенным в самом начале раздела — не правда ли, все ясно без слов?

**Таблица 4.1.** Таблица конечного автомата

| | **0** | **1** | **s** |
|---|---|---|---|
| **начальное** | копирование,<br>1 | перенос,<br>0 | начальное,<br>s |
| **перенос** | копирование,<br>1 | перенос,<br>0 | начальное,<br>1 |
| **копирование** | копирование,<br>0 | копирование,<br>1 | начальное,<br>s |

Второй способ — графический. Автомат изображается в виде диаграммы ориентированного графа (см. врезку "Множества, отношения и графы" в разд. 2.1 и врезку "Представление графов в компьютере" в разд. 2.4.1), узлы которого соответствуют состояниям и помечены символами алфавита состояний, а дуги называются переходами и помечены символами входного и выходного алфавита следующим образом. Допустим, $\delta(a_i,q_j)=q_u$, $\lambda(a_i,q_j)=b_v$. Тогда проводится дуга из узла $q_j$ в узел $q_u$ и помечается символами $a_i$ и $b_v$. Такой граф называется *диаграммой состояний-переходов*. На рис. 4.2 приведена диаграмма графа состояний-переходов для предыдущего примера.

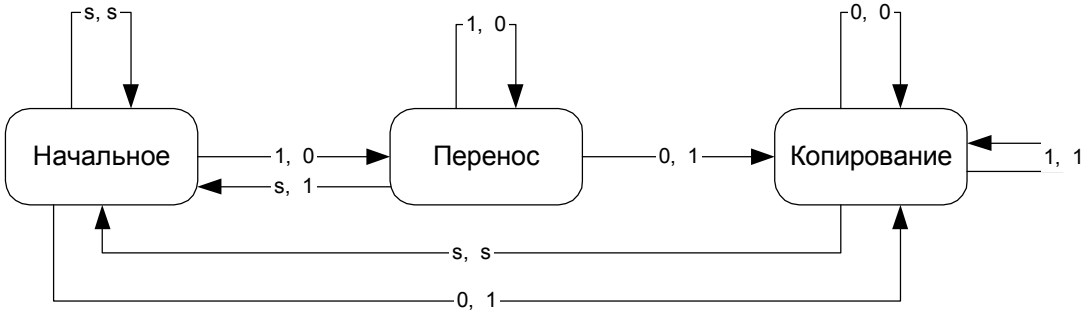



**Рис. 4.2. Диаграмма состояний-переходов**

Третья причина привлекательности конечных автоматов в качестве основного средства моделирования поведения — практическая. В некоторых типичных задачах программирования автоматы уже давно используются как главное

средство. Мы опять уклонимся от исчерпывающего обзора ограничившись одним примером. В задачах синтаксического разбора при трансляции и интерпретации языков программирования автоматы являются основным приемом. Само развитие теории конечных автоматов во многом шло под влиянием потребностей решения задач трансляции. К настоящему времени можно сказать, что эти задачи решены и соответствующие алгоритмы разработаны и исследованы.

В процессе применения в программировании автоматная техника обогатилась рядом приемов, которые не совсем укладываются в исходную математическую модель, но очень удобны на практике. Например, помимо функций переходов и выходов, можно включить в описание автомата еще одну составляющую — процедуру реакции. Смысл добавления состоит в следующем: при выполнении перехода из одного состояния в другое вызывается соответствующая процедура реакции, которая выполняет еще какие-то действия (имеет побочный эффект), помимо вывода выходного символа и смены состояния. Если процедура реакции ничем не ограничена, например, может читать и писать в потенциально бесконечную память, то конечный автомат превращается, фактически, в универсальную модель вычислимости подобную машине Тьюринга. При этом, разумеется, утрачиваются теоретически привлекательные свойства разрешимости, однако программировать с помощью процедур реакции очень удобно. Разработаны и различные промежуточные варианты: например, если побочный эффект процедур реакции ограничен работой со стеком, то получается так называемый магазинный автомат. Магазинные автоматы позволяют запрограммировать существенно более широкий класс алгоритмов и в то же время сохраняют некоторые из важнейших теоретических преимуществ.

Наконец, важным практическим обстоятельством является тот факт, что автоматы очень легко программируются средствами обычных языков программирования. Например, пусть нужно запрограммировать работу автомата с состояниями `1, 2, …`, где `1` — начальное состояние и `k` — заключительное состояние, а входные стимулы: `A, B, ….` В таком случае можно использовать код следующей структуры:[78]

```
state := 1
while state ≠ k
    stimulus := get()
    switch state
        case 1
            switch stimulus
                case A
                . . .
        . . .
```

[78] При записи алгоритмов мы используем некоторый не специфицированный язык, похожий на распространенные языки программирования, применяя такие ключевые слова и операторы, какие хочется и систематически опуская "лишние" разделители — операторы начинаются с новой строки, а структура вложенности определяется отступами.

Мы закончим вводный раздел небольшим примером из информационной системы отдела кадров. Сотрудник в организации, очевидно, может находиться в различных состояниях: вначале он является кандидатом, в результате выполнения операции приема на работу он становится штатным сотрудником. Штатный сотрудник может быть переведен с одной должности на другую, оставаясь штатным сотрудником. Наконец, сотрудник может быть уволен. Жизненный цикл сотрудника естественно описать конечным автоматом, например, в виде табл. 4.2. В этой таблице строки поименованы состояниями, столбцы — стимулами, а в ячейках выписаны процедура реакции и новое состояние.

**Таблица 4.2.** Жизненный цикл сотрудника

| | **Прием** | **Перевод** | **Увольнение** |
|---|---|---|---|
| **Кандидат** | Принять(),<br>В штате | Ошибка(),<br>Кандидат | Ошибка(),<br>Кандидат |
| **В штате** | Ошибка(),<br>В штате | Перевести(),<br>В штате | Уволить(),<br>Уволен |
| **Уволен** | Принять(),<br>В штате | Ошибка(),<br>Уволен | Ошибка(),<br>Уволен |

Если такая таблица кажется недостаточно информативной и наглядной, то эту информацию можно представить в форме диаграммы состояний-переходов. На рис. 4.3 приведена соответствующая данному случаю диаграмма состояний-переходов в нотации UML.

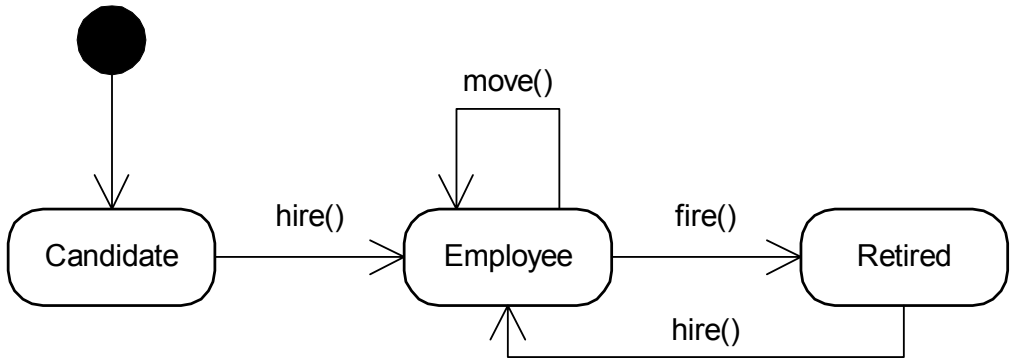



**Рис. 4.3. Жизненный цикл сотрудника в информационной системе отдела кадров**

## *4.1.2. Поведение приложения*

Можно заметить, что приложения различных типов ведут себя (с точки зрения пользователя) совершенно по разному, что не может не сказываться на моделировании поведения.

Для пользователя поведение приложения проявляется прежде всего в интерфейсе. В принципе возможны два архитектурных решения интерфейса с пользователем:

- инициатива принадлежит программе, т. е. программа, будучи запущенной, выполняет свою функцию, следуя заложенному в нее алгоритму решения

задачи, запрашивая по мере необходимости информацию у пользователя (так называемый активный интерфейс);

- инициатива принадлежит пользователю, т. е. программа, будучи запущенной, находится в состоянии ожидания команд пользователя, которые пользователь подает, следуя имеющемуся у него в голове алгоритму решения задачи, а программа выполняет подаваемые команды (так называемый пассивный интерфейс).

В настоящее время более распространенным, особенно в наиболее массовых приложениях для бизнеса, является второе решение (т. е. пассивный интерфейс). Это обусловлено целым рядом причин, в том числе:

- увеличением компактности многофункционального приложения при использовании пассивного интерфейса, т. к. программа должна обеспечить только выполнимость сравнительно небольшого числа сравнительно простых команд, а подбор последовательности выполнения команд, нужных для решения задачи, возлагается на пользователя;
- наличием в современных системах программирования и операционных системах для данной архитектуры развитого механизма поддержки, называемого событийным управлением[79], который описан ниже;
- устоявшейся за последние несколько лет привычкой массового пользователя к пассивному интерфейсу.

### Интуитивно ясный интерфейс

Следует иметь в виду, что при всех своих достоинствах пассивный интерфейс обладает и определенными недостатками, а именно: предполагается, что у пользователя в голове имеется алгоритм решения задачи и что пользователь в состоянии транслировать этот алгоритм в последовательность команд приложения. Другими словами, в приложениях с пассивным интерфейсом предполагается, что пользователь знает, что нужно делать, чтобы решить задачу. Пафос, с которым фирмы, производящие массовые приложения, восхваляют "прозрачность", "интуитивность", "простоту" и т. п. интерфейса своих приложений свидетельствует о том, что задача трансляции не является тривиальной.

*Событийное управление* — это способ структуризации программного кода, основанный на следующей идее. Имеется некоторое предопределенное множество поименованных *событий*. События могут быть явным образом связаны с объектами, а могут быть связаны неявным образом или быть связаны с неявными объектами, в таком случае события обычно называют *системными*. События могут *возникать*. Возникновение события подразумевает, что состояние системы изменилось определенным образом. С событием может быть связана процедура, которая называется *реакцией* на событие. При возникновении события автоматически вызывается процедура реакции. В современных системах программирования, поддерживающих событийное управление, предусматривается

[79] К сожалению, более благозвучное словосочетание "управление событиями" оказывается двусмысленным в русском языке. Правильнее всего было бы говорить "приложение, управляемое событиями", но это слишком длинно и сложно.

большое число самых разнообразных событий, реакции на которые могут быть определены в программе, например: нажатие клавиши на клавиатуре, попадание указателя мыши в определенную область экрана, достижение внутренним таймером заданного значения, открытие заданного файла и т. д. В программе, целиком управляемой событиями, нет основного потока управления, он находится вне программы (в операционной системе или в административной системе времени выполнения, то есть там, где реализован механизм возникновения событий). Управление в программу попадает только в форме вызова процедуры реакции. Такая организация программы обеспечивает высокую модульность, прозрачность, сбалансированность структуры и другие полезные свойства. Понятно, что если связать события с командами приложения (как обычно и делается), то событийное управление как нельзя лучше подходит для реализации пассивного интерфейса.

Однако приложения для бизнеса с пассивным пользовательским интерфейсом являются хотя и распространенным, но не единственным типом приложений. Нетрудно привести примеры важных типов приложений, в которых реакция на внешние события отсутствует или играет второстепенную роль: компиляторы, программы инженерно-технических и научных расчетов и др. В таких программах для описания поведения традиционно используется понятие потока управления.

Вообще говоря, *поток управления* — это последовательность выполнения операторов (команд) в программе. Если программа представляет собой просто последовательность операторов (так называемая *линейная программа*), то операторы в программе выполняются по очереди в естественном порядке (от начала к концу). В этом случае поток управления просто совпадает с последовательностью операторов в программе. Однако обычно это не так.

Во-первых, на поток управления оказывают влияние различные управляющие конструкции: операторы перехода, условные операторы, операторы цикла и т. д. Во-вторых, в большинстве практических систем программирования используется понятие подпрограммы: при выполнении оператора вызова подпрограммы выполнение операторов программы приостанавливается, управление передается в подпрограмму, т. е. в поток управления попадают операторы подпрограммы, а при выходе из подпрограммы возобновляется выполнение операторов программы. При этом считается, что вызванная подпрограмма *активизирована* остается таковой, пока не возобновится выполнение вызвавшей программы или подпрограммы. Кроме того, на поток управления влияют факторы, находящиеся вне данной программы, например, поток управления меняется при обработке прерываний и при возникновении событий.

Различаются однопоточные (т. е. с одним потоком управления) и многопоточные программы. Характерным признаком однопоточной программы является то, что в каждый момент времени можно указать единственный оператор программы, который выполняется в данный момент — говорят, что этот оператор имеет *фокус управления*. В многопоточной программе имеется несколько потоков управления и сосуществуют несколько фокусов управления, соответственно. При этом совершенно необязательно, чтобы различные потоки управления выполнялись физически одновременно на разных процессорах: достаточно иметь в операционной системе механизм, обеспечивающий переключение процессора на выполнение разных потоков управления.

### *4.1.3. Средства моделирования поведения*

В UML предусмотрено несколько различных средств для описания поведения. Выбор того или иного средства диктуется типом поведения, которое нужно описать.

Для описания жизненного цикла конкретного объекта, поведение которого зависит от истории этого объекта, или же требует обработки асинхронных стимулов, используется конечный автомат в форме диаграммы состояний. При этом состояния конечного автомата соответствуют состояниям объекта, т. е. различным наборам значений атрибутов, а переходы соответствуют выполнению операций. Выполнение конструктора объекта моделируется переходом из начального состояния, а выполнение деструктора — переходом в заключительное состояние.

Диаграммы состояний можно составить не только для программных объектов — экземпляров отдельных классов, но и для более крупных конструкций, в частности, для всей модели приложения в целом или для более мелких — отдельных операций.

Для описания потока управления, т. е. последовательности выполняемых элементарных шагов при выполнении отдельной операции или реализации сложного варианта использования, удобно использовать диаграммы деятельности.

Взаимодействие нескольких программных объектов между собой описывается диаграммами взаимодействия в одной из двух эквивалентных форм (диаграммы кооперации и диаграммы последовательности). Для объектно-ориентированной программы поведение прежде всего определяется взаимодействием объектов, поэтому диаграммы данного типа имеют столь важное значение при моделировании поведения в UML.

Моделирование многопоточности в UML выполняется с помощью понятия активного класса, которое по сути введено в язык как синоним понятия поток управления.

В следующих разделах рассматриваются все указанные средства более детально.

## 4.2. Диаграммы состояний

Диаграммы состояний в UML являются реализацией основной идеи использования конечных автоматов как средства описания алгоритмов и, тем самым, моделирования поведения. Конечные автоматы в UML реализованы довольно своеобразно. С одной стороны, в основу положено классическое представление автомата в форме графа состояний-переходов (разд. 4.1.1). С другой стороны, к классической форме добавлено большое число различных расширений и вспомогательных обозначений, которые, строго говоря, не обязательны — без них в принципе можно было бы обойтись — но весьма удобны и наглядны при составлении диаграмм. Такое решение является обоюдоострым оружием: диаграммы состояний UML более наглядны и выразительные по сравнению с классическими представлениями автоматов, но их применение требует большей подготовленности пользователя и предъявляет более высокие требования к "сообразительности" и "внимательности" инструментов моделирования.[80] При

[80] Если проанализировать те многочисленные неточности, ошибки и отклонения от стандарта, которыми грешат версии инструментов моделирования, попавшиеся автору под руку при

описании конструкций диаграмм состояний мы не только опишем их семантику согласно стандарта, но и покажем возможное сведение дополнительных конструкций к базовым.

Итак, начиная обзор средств моделирования с самого верхнего уровня, можно констатировать, что на диаграммах состояний применяется всего один тип сущностей — состояния, и всего один тип отношений — переходы. Совокупность состояний и переходов между ними образует *машину состояний*.[81] На рис. 4.4 приведена метамодель машины состояний UML в самом общем виде, далее мы ее уточним и детализируем.

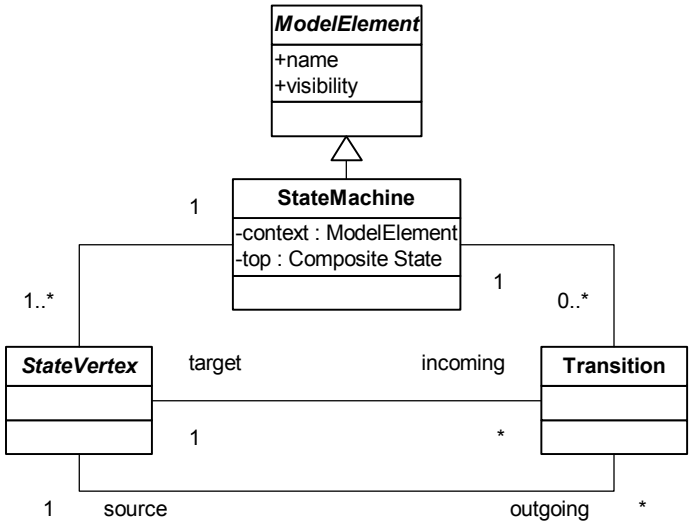



**Рис. 4.4. Метамодель машины состояний**

Мы видим, что автомат (машина состояний) состоит из набора состояний, для каждого из которых определены исходящие (`outgoing`) переходы и входящие (`incoming`) переходы, и набора переходов, для каждого из которых определено исходное (`source`) и целевое (`target`) состояния. Для машины в целом определен контекст (`context`), т. е. тот элемент модели, описанием поведения которого она является. Кроме того, имеется ссылка на одно из составных состояний (атрибут `top`), смысл и назначение которой обсуждается ниже, в разд. 4.2.3.

Таким образом, типов сущностей и отношений предельно мало, но подтипов, вариантов нотации и специальных случаев для них определено много (может быть, даже слишком много).

А именно, состояния бывают:

- простые,
- составные,
- специальные

---

подготовке книги, то бросается в глаза, что хуже всего дело обстоит с диаграммами состояний, и лучше всего — с диаграммами классов.

[81] Машина состояний — термин, принятый в англоязычной литературе, но обозначающий тоже самое, что выше мы называли конечным автоматом. Дело в том, что по-английски, видимо, state machine гораздо удобнее заумного finite automaton, но для нашего уха "автомат" как-то привычнее.

и каждый тип состояний имеет дополнительные подтипы и различные составляющие элементы.
Переходы бывают простые и составные, и каждый переход содержит от двух до пяти составляющих:

- исходное состояние,
- событие перехода,
- сторожевое условие,
- действие на переходе,
- целевое состояние.

Рассмотрим все эти элементы по порядку.

### *4.2.1. Простое состояние*

Простое состояние является в UML простым только по названию — оно имеет следующую структуру:

- имя;
- действие при входе;
- действие при выходе;
- множество внутренних переходов;
- внутренняя активность;
- множество отложенных событий.

*Имя состояния* является обязательным.[82] Все остальные составляющие простого состояния не являются обязательными. Фактически, имя (простого) состояния — это символ алфавита состояний $Q$.
*Действие при входе* (обозначается при помощи ключевого слова `entry`) — это указание атомарного действия (разд. 4.3.1), которое должно выполняться при переходе автомата в данное состояние. Действие при входе выполняется *после* всех других действий, предписанных переходом, переводящим автомат в данное состояние.
*Действие при выходе* (обозначается при помощи ключевого слова `exit`) — это указание атомарного действия, которое должно выполняться при переходе автомата из данного состояние. Действие при выходе выполняется *до* всех других действий, предписанных переходом, выводящим автомат из данного состояния.
Множество внутренних переходов — это множество простых переходов из данного состояния в это же состояние (так называемых *переходов в себя*). *Внутренний переход* отличается от простого перехода в себя тем, что действия при выходе и входе *не* выполняются. Синтаксис внутреннего перехода совпадает с синтаксисом простого перехода.
*Внутренняя активность* (обозначается при помощи ключевого слова `do`) — это указание деятельности, которая начинает выполняться при переходе в данное состояние после выполнения всех действий, предписанных переходом, включая действие на входе. Внутренняя активность либо заканчивается по завершении,

[82] Некоторые инструменты, поддерживающие UML 2.0, анонимные состояния. Уникальное имя все равно присутствует во внутреннем представление модели, но оно генерируется автоматически и не показывается пользователю на диаграмме.

либо прерывается в случае выполнения перехода (в том числе и внутреннего перехода). В классической модели конечный автомат, находясь в некотором состоянии, ничего не делает: он находится в состоянии ожидания перехода. В модели UML считается, что автомат можно нагрузить какой-то полезной фоновой деятельностью, которая будет прерываться при выполнении любого перехода.

Если в то время, когда автомат находится в некотором состоянии, происходит событие, для которого в данном состоянии не определен переход, то согласно семантики UML ничего не происходит и событие безвозвратно теряется.[83] В некоторых случаях этого требуется избежать. Для этого в UML предусмотрено понятие отложенного события. *Отложенное событие* — это событие, для которого не определено перехода в данном состоянии, но которое, тем не менее, не должно быть потеряно, если оно произойдет, пока автомат находится в данном состоянии. Семантика отложенного события такова: если происходит отложенное событие, то оно помещается в конец некоторой системной очереди отложенных событий. После перехода автомата в новое состояние проверяется (начиная с начала) очередь отложенных событий. Если в очереди есть событие, для которого в новом состоянии определен переход, то событие извлекается из очереди и происходит переход.

Рассмотрим пример из информационной системы отдела кадров. Проектируя жизненный цикл сотрудника, ограничимся пока одним простым состоянием — `Working`, соответствующему ситуации, когда сотрудник принят на работу и работает. С помощью действий на входе и выходе мы отмечаем необходимость выполнения соответствующих манипуляций с учетной записью сотрудника. Далее, мы можем отметить, что существует событие — командировка, которое не выводит сотрудника из данного состояния, но требует выполнения каких-то действий. Мы можем различать длительные и местные командировки: скажем, при местной командировке сотрудник только должен временно передать текущие тела кому-то, а при длительной командировке нужно исключить его из исполнителей проектов и заблокировать учетную запись. Различие в деталях реакций на эти события можно передать с помощью перехода в себя и внутреннего перехода. Если ничего не происходит, то по умолчанию сотрудник должен выполнять свои основные обязанности. Соответствующее простое состояние приведено на рис. 4.5.

[83] Некоторые авторы считают это недостатком UML. Действительно, в теории конечных автоматов обычно считается, что функция перехода определена для всех пар состояние–стимул. Такой подход является более строгим. В UML функция переходов может быть не всюду определенной, что иногда позволяет сократить объем описания модели. Фактически, UML чуть-чуть поступается строгостью ради удобства.

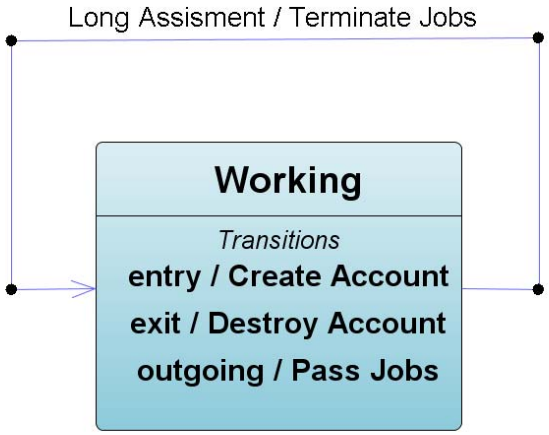



**Рис. 4.5. Простое состояние сотрудника**

Выше мы отметили, что многие элементы диаграмм состояний в UML введены только для удобства, без них можно в принципе обойтись. В подтверждение нашего тезиса приведем один из возможных вариантов исключения дополнительных элементов из модели с сохранением ее семантики. Допустим, что имеется простое состояние `State1`, описанное на рис. 4.6 с использованием таких средств, как действие при входе, действие при выходе и внутренний переход.

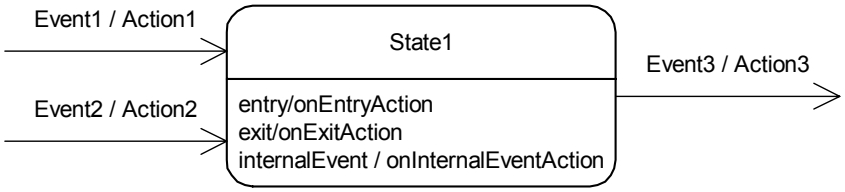



**Рис. 4.6. Фрагмент модели с простым состоянием до преобразования**

В этом случае эквивалентную по поведению модель, не использующую дополнительных средств, а только имена простых состояний и простые переходы (классический вариант конечного автомата), можно получить, перенося действия на входе и выходе в действия входящих и исходящих переходов, соответственно, и моделируя внутренний переход переходом в себя (рис. 4.7).

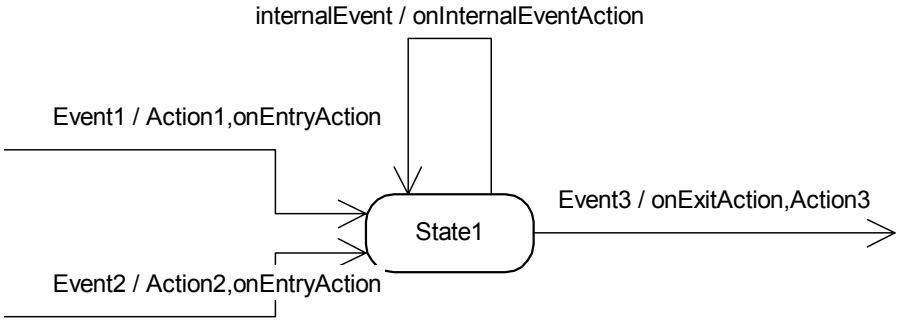



**Рис. 4.7. Фрагмент модели с простым состоянием после преобразования**

Какой стиль предпочесть — дело вкуса.

### 4.2.2. Простой переход

Простой переход всегда ведет из одного состояния в другое состояние.[84] Существует несколько ограничений для специальных состояний (разд. 4.2.3), например, для начального состояния не может быть входящих переходов, а для заключительного — исходящих, а в остальном переходы между состояниями могут быть определены произвольным образом.

Прочие составляющие — событие перехода, сторожевое условие и действия на переходе не являются обязательными. Если они присутствуют, то изображаются в виде текста в определенном синтаксисе рядом со стрелкой, изображающей переход. Синтаксис описания перехода следующий:

```
Событие [ Сторожевое условие ] / Действие
```

Например, в информационной системе отдела кадров переход сотрудника из состояния "кандидат" в состояние "в штате" может быть описано с помощью простого перехода, представленного на рис. 4.8.

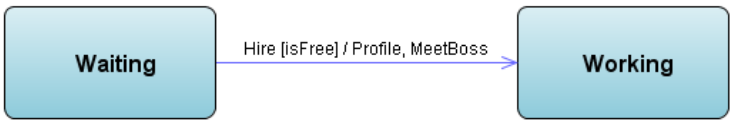



**Рис. 4.8. Простой переход**

Общая семантика перехода такова. Допустим, что автомат находится в состоянии $S_1$, в котором определен исходящий переход $T$ с событием $E$, сторожевым условием $G$ и действием $D$, ведущий в состояние $S_2$. Если возникает событие $E$, то переход $T$ *возбуждается* (в данном состоянии могут быть одновременно возбуждены несколько переходов — это не считается противоречием в модели). Далее проверяется сторожевое условие $G$ (проверяются сторожевые условия всех возбужденных переходов в неопределенном порядке). Если сторожевое условие выполнено, то переход $T$ *срабатывает* — выполняется действие на выходе из состояния $S_1$, выполняется действие на переходе $D$, выполняется действие на входе в состояние $S_2$ и автомат переходит в состояние $S_2$. Даже если у нескольких возбужденных переходов сторожевые условия оказываются истинными, то, тем не менее, срабатывает всегда только один переход из возбужденных. Какой именно переход срабатывает — не определено. Таким образом поведение, описываемое подобным автоматом, является недетерминированным.[85] Если же сторожевое условие $G$ не выполнено, то переход $T$ *не* срабатывает. Если ни один из возбужденных переходов не срабатывает, то событие $T$ *теряется* и автомат остается в состоянии $S_1$.

---

[84] Строго говоря, фрагменты модели на рис. 5.5 и 5.6 синтаксически неправильны. Мы злоупотребили доверчивостью Visio, воспользовавшись тем, что инструмент не проверяет данное синтаксические правило UML.

[85] В некоторых системах моделирования поведения, основанных на конечных автоматах, проблема неоднозначности срабатывания более чем одного перехода решается тем, что статически определяются приоритеты для всех переходов. Срабатывает переход с наивысшим приоритетом.

*Событие перехода* — это тот входной символ (стимул), который вкупе с текущим состоянием автомата определяет следующее состояние.

В UML предусматривается несколько типов событий (разд. 4.2.4). Само слово "событие" невольно вызывает следующую ассоциацию: существует некий внешний по отношению к автомату мир, в котором время от времени происходят события (в общечеловеческом смысле этого слова), автомату становится известно о произошедшем событии и он реагирует на событие путем перехода в определенное состояние. Эта ассоциация вполне правомерна, если речь идет о моделировании жизненного цикла объекта в программе, управляемой событиями. Действительно: в этом случае основной тип событий — это события вызова методов объекта. Объект реагирует на них, выполняя тела методов и меняя значения своих атрибутов (состояние). Однако, данная ассоциация далеко не единственно возможная (хотя и самая распространенная в объектно-ориентированном программировании). Автомату не важен источник событий: важна последовательность, в которой события поступают на вход автомата, вынуждая его реагировать. Например, последовательность символов в слове входного алфавита *A* (см. разд. 4.1.1) также рассматривается как последовательность событий. Для автомата существенно, что первое событие предшествует второму, а почему это происходит: потому ли что первое событие произошло раньше или же потому что расположено левее — это не важно. Таким образом, конечные автоматы, в том числе машины состояний UML, подходят для моделирования поведения не только событийно-управляемых программ, но и любого другого типа управления.[86]

UML допускает наличие переходов без событий — такой переход называется *переходом по завершении*.

*Переход по завершении* – это переход без событий.

Семантика перехода по завершении такова. Имеется одно неявное, а потому безымянное событие, которое наступает при завершении внутренней активности в состоянии (если никакой внутренней активности не предусмотрено, то это событие наступает немедленно после перехода автомата в данное состояние). Поскольку событие завершения является безымянным, оно никак не отображается на диаграмме: в тексте, сопутствующем переходу, просто отсутствует первая часть, относящаяся к событию перехода. В остальном переходы по завершении ничем не отличаются от простых переходов: они могут содержать сторожевые условия, может быть несколько исходящих переходов по завершении для данного состояния (с альтернативными сторожевыми условиями), для переходов по завершении можно определять последовательности действий при переходе и т. д.

Следует, однако, иметь в виду, что при частом использовании переходов по завершении легче допустить и труднее обнаружить ошибку модели, порождающую неопределенное поведение. Простой пример: допустим, что имеется состояние, для которого определено несколько исходящих переходов по завершении (с разными сторожевыми условиями) и не определено других переходов по событиям. Допустим, что ни один из переходов по завершении не сработает, ввиду того, что

[86] Обратите внимание, что в схеме кода конечного автомата, приведенной в конце разд. 5.1.1, используется обычный последовательный поток управления.

все сторожевые условия окажутся ложными. В таком случае, поскольку событие завершения возникает ровно один раз и оно уже утеряно, а все другие события в данном состоянии теряются, так как для них не определены переходы, моделируемая система будет демонстрировать характерное поведение, печально знакомое слишком многим пользователям: "зависание".[87] Таким образом, переход по завершении средство удобное, но опасное. В классические модели конечных автоматов подобные средства обычно не включают, благодаря чему становится возможна автоматическая проверка отсутствия "зависаний" и других неприятностей. Мы настойчиво рекомендуем не использовать переходы по завершении в машинах состояний, кроме случаев крайней необходимости, примеры которых приведены ниже. Другое дело диаграммы деятельности (разд. 4.3) — там переходы по завершении являются основным средством.

*Сторожевое условие* — это логическое выражение, которое должно оказаться истинным для того, чтобы возбужденный переход сработал.

Для каждого возбужденного перехода сторожевое условие проверяется ровно один раз, сразу после того, как переход возбужден и до того, как в системе произойдут какие-либо другие события. Если сторожевое условие ложно, то переход не срабатывает и событие теряется. Даже если впоследствии сторожевое условие станет истинным, переход сможет сработать только если повторно возникнет событие перехода.

Как уже было сказано, возможны несколько исходящих переходов из данного состояния с одним и тем же событием перехода, но с разными сторожевыми условиями. Все такие переходы возбуждаются при наступлении события перехода, но только один срабатывает. Сторожевые условия формулируются относительно значений аргументов события перехода и значений атрибутов объекта, поведение которого моделируется. Для того, чтобы поведение автомата было определено детерминировано, набор сторожевых условий должен образовывать полную дизъюнктную систему предикатов, т. е. при любых значениях переменных, входящих в сторожевое условие, ровно одно из сторожевых условий должно быть истинным. Сторожевые условия могут написаны на любом языке, в частности, на естественном. Поэтому автоматически проверить выполнение данного требования невозможно. Более того, даже если принять соглашение, что сторожевые условия должны быть написаны на каком-либо формальном логическом языке, то, тем не менее, проверить полноту и дизъюнктность системы сторожевых условий не всегда возможно автоматически. Таким образом, сторожевые условия в автоматах — вещь удобная, но требующая от моделирующего определенных умственных усилий.

В UML предусмотрены синтаксические средства, до некоторой степени облегчающие семантически правильное построение сторожевых условий за счет более наглядного их изображения. Таковыми являются:

- сегментированные переходы;
- символы ветвления;
- переходные состояния;

[87] Добиться "зависания" машины состояний UML можно не только переходами по завершении, но и другими средствами — дурное дело не хитрое.

- предикат `else`.

Линия перехода может быть разбита на части, называемые *сегментами*.

| *Сегменты перехода* — части, на которые может быть разбита линия перехода. |
|---|

Разбивающими элементами являются следующие фигуры:

- *переходное состояние* (изображается в виде небольшого кружка);
- *ветвление* (изображается в виде ромба);
- действия посылки и приема сигнала.

Сегментирование перехода и переходные состояния применяются в UML в нескольких ситуациях. Здесь мы рассматриваем их в связи со сторожевыми условиями, а прочие случаи описаны в соответствующем контексте.

Несколько переходов, исходящих из данного состояния и имеющих общее событие перехода, можно объединить в дерево сегментированных переходов следующим образом. Имеется один сегмент перехода, который начинается в исходном состоянии и заканчивается в переходном состоянии или ветвлении. Далее из этого переходного состояния или ветвления начинаются другие сегменты, которые заканчиваются в целевых состояниях или новых переходных состояниях или ветвлениях. Сегмент перехода, начинающийся в исходном состоянии, называется корневым, сегменты, заканчивающиеся в целевых состояниях называются листовыми. Событие перехода может быть указано только для корневого сегмента, действия на переходе могут быть указаны только для листовых сегментов, а сторожевые условия могут быть указаны для любых сегментов. Такое дерево сегментированных переходов семантически эквивалентно множеству простых переходов, которое получится, если рассмотреть все пути из исходного состояния в целевые состояния, считая встречающиеся на пути сторожевые условия соединенными конъюнкцией.

Замысловатое определение предыдущего абзаца не таит в себе ничего необычного или нового. Покажем это на примере из информационной системы отдела кадров. Допустим, что требуется проводить более дифференцированную кадровую политику и различать три различных состояния уволенных с предприятия:

- `non grata` — скандалист, бездельник и нарушитель трудовой дисциплины, уволенный по инициативе администрации, которого ни при каких обстоятельствах нельзя нанимать на работу;
- `welcome` — хороший работник, с котором администрации пришлось расстаться ввиду временных трудностей, переживаемых предприятием, и которого при первой возможности следует пригласить обратно;
- `retired` — работник уволившийся по собственному желанию, повторный прием которого должен проходить на общих основаниях.

Это, разумеется, очень грубая классификация, но для демонстрационных целей вполне достаточная. На рис. 4.9 представлен соответствующий фрагмент диаграммы состояний, в котором использованы дерево сегментированных переходов и переходные состояния.

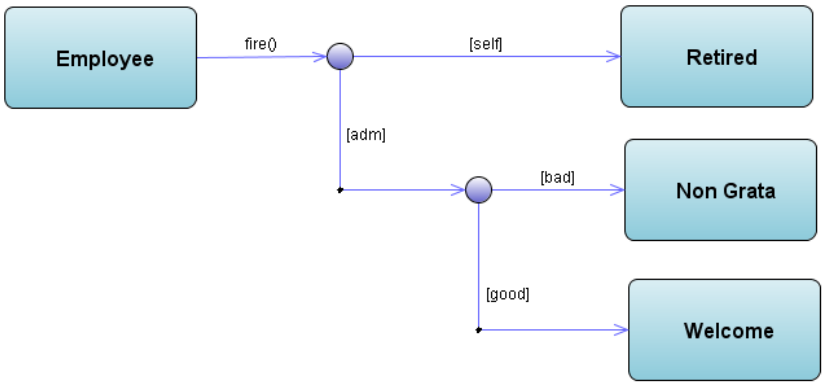



**Рис. 4.9. Дерево сегментированных переходов**

Переходные состояния в данном контексте имеют тот же смысл и могут быть использованы вместо фигур ветвления с сохранением семантики диаграммы (см. ниже рис 4.10).

Продолжим рассмотрение данного примера. *Мы* знаем, что условия увольнения `adm` и `self` взаимно исключают друг друга и одно из них при увольнении обязательно имеет место. Но это знаем только мы — инструмент моделирования этого не знает и проверить полноту и дизъюнктность системы условий не сможет. Но он может помочь обозначить наше желание наделить систему условий нужными свойствами. Для этого используется ключевое слово `else`, которое обозначает условие, считающееся истинным во всех случаях, когда ложны все другие условия, приписанные к сегментам, исходящим из данного ветвления. Далее, поскольку такой предикат единственен, его можно не писать, подразумевая по умолчанию.[88]

В результате описание сложной системы условий становится нагляднее и надежное. Кстати, в эту систему обозначений элегантно вписывается семантика отсутствия сторожевого условия для простого перехода: если ничего не написано, то опущенное условие истинно, когда альтернативные ложны, то есть в случае простого безальтернативного перехода всегда, поскольку альтернативные условия отсутствуют. Таким образом, фрагмент диаграммы состояний на рис. 4.10 семантически эквивалентен фрагменту на рис. 4.9.

[88] Строго говоря, данное утверждение не является частью спецификации UML. Инструмент имеет право этого не знать. Однако данное соглашение является распространенной языковой практикой и большинство людей такую систему обозначений знают и понимают. Поскольку наша книга предназначена для чтения людьми, мы используем соглашение, что по умолчанию подразумевается `else`, хотя это не является частью стандарта.

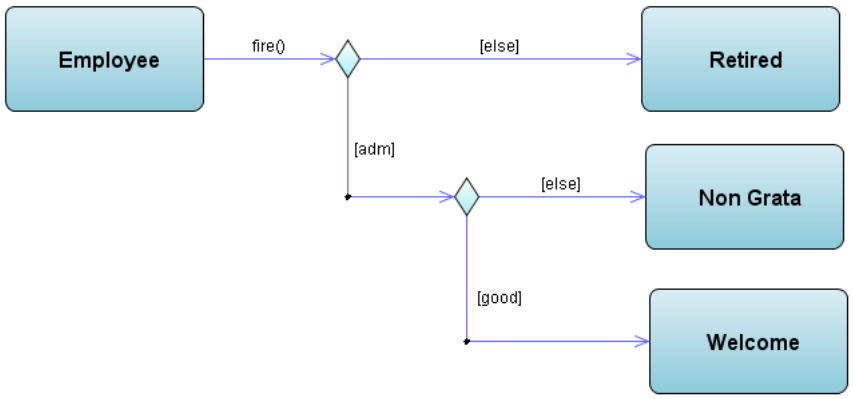



**Рис. 4.10. Использование предиката else**

Подводя итог обсуждению сторожевых условий, еще раз подчеркнем, что сегментированные переходы и ветвления ничего не добавляют (и не убавляют) в семантике модели: это просто синтаксические обозначения, введенные для удобства и наглядности. Ту же самую семантику, которую имеют фрагменты диаграмм состояний на рис. 4.9 и 4.10 можно передать с помощью фрагмента, приведенного на рис. 4.11.

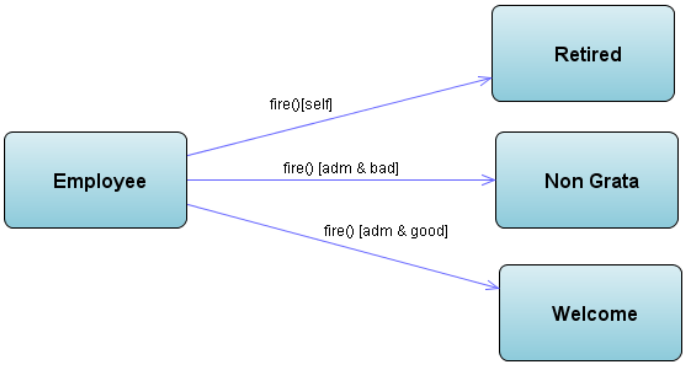



**Рис. 4.11. Множество простых переходов с одним событием перехода и различными сторожевыми условиями**

Последней составляющей простого перехода является действие.

| |
|---|
| *Действие* — это непрерываемое извне атомарное вычисление, чье время выполнения пренебрежимо мало. |

Авторы языка подразумевали, что инструменты моделирования будут связывать с понятием действия в модели UML понятие действия (или аналогичное) в целевом языке программирования. Например, для обычных языков программирования действиями являются вычисление значения выражения и присваивание его переменной, вызов процедуры, посылка сигнала и т. д. В UML предусмотрено несколько типов действий (см. разд. 4.3.1), похожих по семантике на действия в наиболее распространенных языках программирования. Однако UML не является языком программирования и, тем самым, не претендует на то, чтобы быть универсальным языком описания действий. Поэтому понятие действия в UML

сознательно недоопределено — оставлена свобода, необходимая инструментам для непротиворечивого расширения семантики действий UML до семантики действий конкретного языка программирования. Более детально эти вопросы рассмотрены в разд. 4.3.1. Здесь, в контексте обсуждения машины состояний UML, стоит подчеркнуть три обстоятельства.

- Действие является атомарным и непрерываемым. При выполнении действия на переходе или в состоянии не могут происходить события, прерывающие выполнение действия. Точнее говоря, событие может произойти, но система обязана задержать его обработку до окончания выполнения действия.
- Действие является безальтернативным и завершаемым. Раз начавшись, действие выполняется до конца. Оно не может "раздумать" выполняться или выполняться неопределенно долго.
- Последовательность действий также является действием. (Синтаксически, действия в последовательности разделяются запятыми).

Действия являются важнейший частью описания поведения с помощью конечных автоматов. В сущности, действия в UML это в точности то, что выше (в разд. 4.1.1) мы назвали процедурой реакции автомата. В UML действия, составляющие процедуру реакции, фактически ничем не ограничены: в так называемых не интерпретируемых действиях (см. разд. 4.3.1) могут быть скрыты любые программистские трюки. Поэтому формальные свойства машины состояний UML трудно проверить автоматически. С другой стороны, машины состояний UML выразительны и наглядны — многочисленные синтаксические добавления позволяют моделировать сложное поведение компактно и красиво.

Мы закончим раздел, посвященный переходам, соответствующим фрагментом метамодели (рис. 4.12). Метамодель получилась очень простой, это объясняется тем, что детали событий и действий мы рассматриваем в более подходящем, по нашему мнению, контексте (разд. 4.2.4 и 4.3.1), а также тем, что метамодель отражает только абстрактный синтаксис UML, необходимый для описания семантики. Многочисленные детали способов изображения простых переходов нет нужды отражать в метамодели.

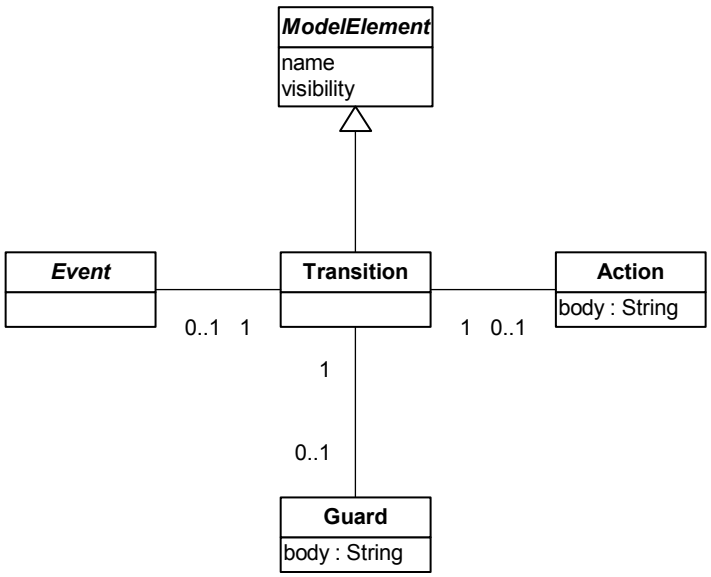



**Рис. 4.12. Метамодель простого перехода**

### *4.2.3. Составные и специальные состояния*

В разд. 4.2.1 мы рассмотрели простые состояния, соответствующие состояниям обычной модели конечного автомата. Пришла пора рассмотреть другие понятия, близкие к понятию состояния, но специфические для UML. Таковых две основные группы:

- составные состояния и
- специальные состояния.
- Составное состояние может быть
- последовательным или
- параллельным (ортогональным).

Специальные состояния, в свою очередь, бывают следующих типов:

- начальное состояние,
- заключительное состояние,
- переходное состояние,
- историческое состояние (в двух вариантах),
- синхронизирующее состояние,
- ссылочное состояние,
- состояние "заглушка".

Переходное состояние рассмотрено в разд. 4.2.2, параллельное составное состояние и состояние синхронизации рассмотрены в разд. 4.4.1, остальные перечисленные понятия рассматриваются в данном разделе.

Начнем с составных состояний.

*Составное состояние* — это состояние, в которое вложена машина состояний. Если вложена только одна машина, то состояние называется последовательным, если несколько — параллельным.[89] Глубина вложенности в UML неограниченна, т. е. состояния вложенной машины состояний также могут быть составными.

Мы начнем с простого примера, чтобы сразу пояснить прагматику составного состояния, т. е. зачем это понятие введено в UML, а затем опишем тонкости семантики и связь с другими понятиями машины состояний UML.

Рассмотрим все известный прибор: светофор. Он может находится в двух основных состояниях:

- `Off` — вообще не работает — выключен или сломался, как слишком часто бывает;
- `On` — работает.

Но работать светофор может по-разному:

- `Blinking` — мигающий желтый, дорожное движение не регулируется;
- `Working` — работает по-настоящему и регулирует движение.

В последнем случае у светофора есть 4 видимых состояния, являющихся предписывающими сигналами для участников дорожного движения:

`Green` — зеленый свет, движение разрешено;

`GreenYellow` — состояние перехода из режима разрешения в режим запрещения движения (это настоящее состояние, светофор находится в нем заметное время);

`Red` — красный свет, движение запрещено;

`RedYellow` — состояние перехода из режима запрещения в режим разрешения движения (это состояние отличное от `GreenYellow`, светофор подает несколько иные световые сигналы и участники движения обязаны по другому на них реагировать).

На рис. 4.13 приведена соответствующая диаграмма состояний (несколько забегая вперед, мы использовали здесь событие таймера, выделяемое ключевым словом `after` и описанное в следующем разделе).

[89] В UML 2.0 параллельные состояния переименованы в ортогональные. Мы используем оба термина как синонимы.

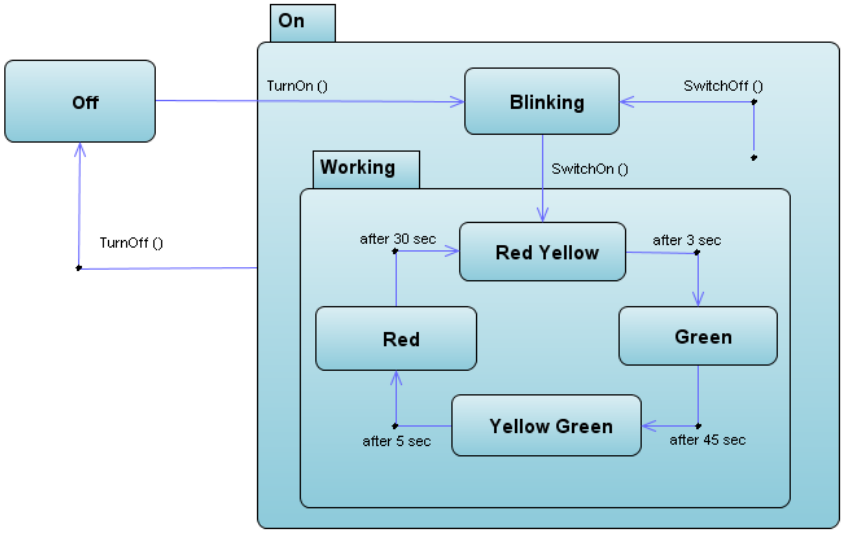



**Рис. 4.13. Составные состояния**

Как мы уже упоминали, в принципе всегда можно обойтись без использования составных состояний. Например, на рис. 4.14 приведена эквивалентная машина состояний (т. е. описывающая то же самое поведение), не содержащая составных состояний. Сравнение диаграмм на рис. 4.13 и 4.14 является, по нашему мнению, достаточным объяснением того, *зачем* в UML введены составные состояния.

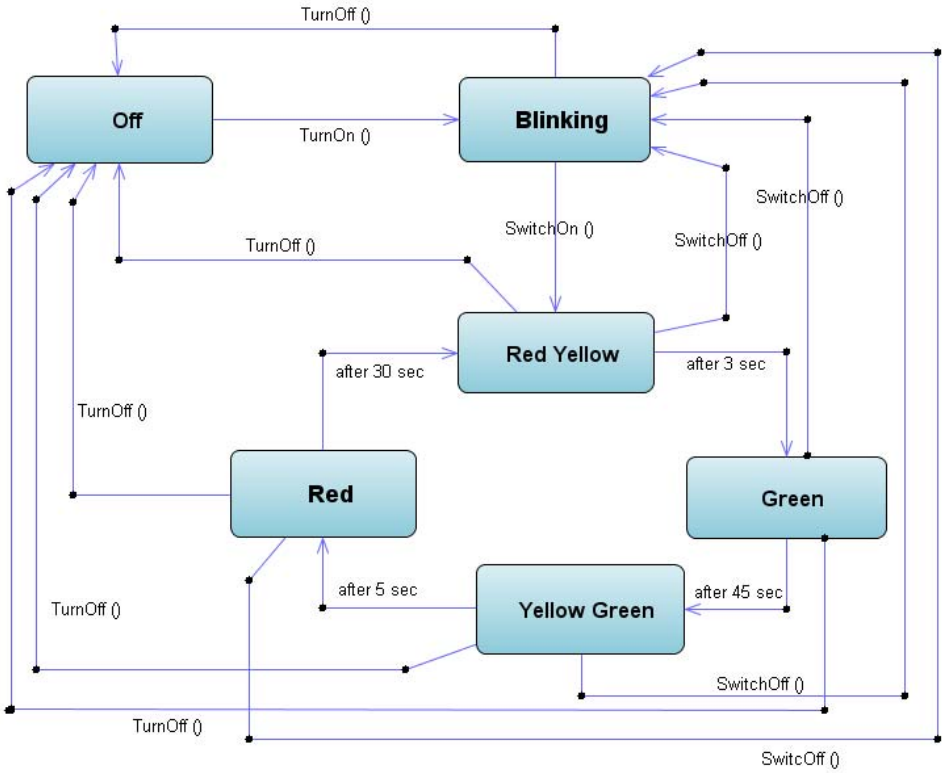



**Рис. 4.14. Эквивалентная диаграмма, не содержащая составных состояний**

Общая идея составного состояния ясна. Теперь нужно внимательно разобраться с деталями составных состояний и связанных с ними переходов. Мы сделаем это постепенно, несколько раз возвратившись к описанию семантики составных состояний и переходов в подходящем контексте.

Первое правило можно сформулировать прямо здесь: переход из составного состояния наследуется всеми вложенными состояниями. При этом, если для вложенного состояния определен переход с тем же событием перехода и сторожевым условием, что и у наследуемого перехода, то преимущество имеет локально определенный переход.

Мы не случайно употребили характерный термин объектно-ориентированного программирования "наследование" в данном контексте. По нашему мнению, назначение составных состояний аналогично назначению суперклассов:[90] выявить общее в нескольких элементах и описать это общее только один раз. Тем самым сокращается описание модели и она становится более удобной для восприятия человеком (сравните рис. 4.13 и 4.14 еще раз).

Перейдем к рассмотрению специальных состояний. Прежде всего, специальное состояние состоянием не является,[91] в том смысле, что автомат не может в нем "пребывать" и оно не может быть текущим активным состоянием (см. разд. 4.4.1).

*Начальное состояние* — это специальное состояние, соответствующее ситуации, когда машина состояний *еще* не работает.

На диаграмме начальное состояние изображается в виде закрашенного кружка. Начальное состояние не имеет таких составляющих, как действия на входе, выходе и внутренняя активность, но оно обязано иметь исходящий переход,[92] ведущий в то состояние, которое будет являться по настоящему первым состоянием при работе машины состояний. Исходящий переход из начального состояния не может иметь события перехода, но может иметь сторожевое условие. В последнем случае должны быть определены несколько переходов из начального состояния, причем один из них обязательно должен срабатывать. В программистских терминах начальное состояние — это метка точки входа в программу. Управление не может задержаться на этой метке. Даже графически типичный случай начального состояния с одним непомеченным переходом очень похож на бытовую пиктограмму "начинать здесь". Начальное состояние может иметь действие на переходе — это действие выполняется до начала работы машины состояний.

Насколько обязательным является использование начального состояния в диаграмме состояний? Этот вопрос не имеет однозначного ответа — все зависит от ситуации. Например, в диаграммах на рис. 4.13 и 4.14 мы обошлись без начального состояния, и поведение светофора не стало от этого менее понятным. Однако в других случаях наличие начального состояния может быть желательно или даже

---

[90] Но состояния не являются классификаторами, поэтому прямо использовать отношение обобщения было бы синтаксически неправильным.

[91] В оригинале специальное состояние называется pseudostate, но слово псевдосостояние нам кажется слишком искусственным.

[92] Разумеется, начальное состояние не может иметь входящих переходов — машина состояний не может вернуться в ситуацию до начала своей работы.

необходимо. Прежде всего, если имеется переход в составное состояние, то внутри этого составного состояния обязано присутствовать начальное состояние — в противном случае неясно, куда же ведет данный переход. Далее, если машина состояний описывает поведение программного объекта, создаваемого и уничтожаемого в программе, то присутствие начального состояния на диаграмме является весьма желательным: начальное состояние показывает, в каком состоянии находится объект при создании его конструктором (а в действия на переходе из начального состояния удобно поместить инициализацию атрибутов). С другой стороны, если начало и окончание жизненного цикла объекта, поведение которого моделируется, выходят за пределы моделируемого периода (например, нас не интересует ни процесс изготовления новых светофоров, ни процесс утилизации отслуживших свое), то начальное состояние на диаграмме состояний является излишним и может даже мешать восприятию.

*Заключительное состояние* — это специальное состояние, соответствующее ситуации, когда машина состояний *уже* не работает.

На диаграмме заключительное состояние изображается в виде закрашенного кружка, который обведен дополнительной окружностью. Подобно начальному состоянию, заключительное состояние не имеет таких составляющих, как действия на входе, выходе и внутренняя активность, но имеет входящий переход,[93] ведущий из того состояния, которое является последним состоянием в данном сеансе работы машины состояний.

Вообще говоря, работа машины состояний может завершаться несколькими различными способами. Это соответствует общепринятой программистской практике: программа может иметь вариант нормального завершения и несколько вариантов завершения по исключительной ситуации или ошибке. Отражая данную особенность поведения на диаграмме состояний, можно указать несколько переходов в одно и то же заключительное состояние. Синтаксически это допустимо. Однако мы настойчиво рекомендуем так не делать и помещать на диаграмму столько заключительных состояний, сколько в действительности существует семантически различных вариантов завершения работы данной машины состояний.

### Вредные советы

Авторы UML приложили огромные усилия, чтобы разработать систему обозначений, которая была бы понятна всем без исключения. Однако все равно находятся люди, рисующие диаграммы, понять которые крайне затруднительно. Специально для них мы помещаем несколько простых советов, следование которым надежно обеспечит абсолютную непонятность диаграмм, в частности, диаграмм состояний.

- Используйте имена состояний, которые предлагает инструмент: `State1`, `State2` и т. д.
- Никогда не рисуйте более одного заключительного состояния.

[93] Разумеется, заключительное состояние не может иметь исходящих переходов — чтобы машина состояний заново заработала, ее нужно снова запустить.

- Нарисуйте столько вложенных состояний, сколько поместится на листе, но ни в коем случае не используйте переходы между составными состояниями.
- Не допускайте ситуации, когда переход из начального состояния ведет в простое состояние той же машины состояний: такой переход должен пересекать несколько границ составных состояний.

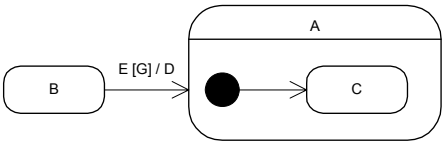
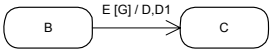
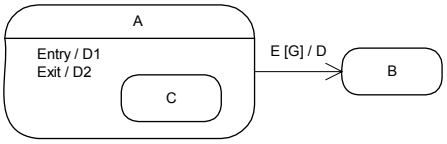
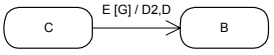
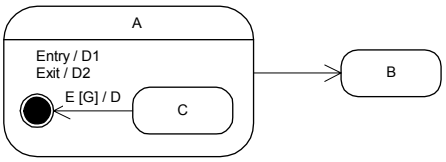
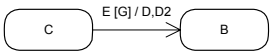
Попробуйте применить наши советы на практике — результат превзойдет ваши ожидания!

Прежде чем переходить к описанию других специальных состояний, еще раз уточним связь между составными состояниями, переходами между ними, начальным и заключительным состоянием. Напомним, что:

- если имеется входящий переход в составное состояние, то машина состояний, вложенная в данное составное состояние, обязана иметь начальное состояние;
- если машина состояний, вложенная в составное состояние, имеет заключительное состояние, то данное составное состояние обязано иметь исходящий переход по завершении;
- машина состояний верхнего уровня считается вложенной в составное состояние (с именем `top`, см. метамодель на рис. 4.4), которое не имеет ни исходящих, ни входящих переходов.

Семантику составных состояний и переходов мы хотим пояснить с помощью указания эквивалентных конструкций, не использующих составных состояний. В табл. 4.3 состояние A — составное, а состояния B и C — любые, т. е. простые или составные. В последнем случае правила табл. 4.3 рекурсивно применяются к вложенным состояниям на всю глубину вложенности.

**Таблица 4.3.** Эквивалентные выражения для переходов между составными состояниями

| Вид перехода | Диаграмма перехода | Эквивалентная диаграмма |
|---|---|---|
| Переход в составное состояние |  |  |
| Переход из составного состояния |  |  |
| Переход по завершении |  |  |

*Историческое состояние* — это специальное состояние, подобное начальному состоянию, но обладающее дополнительной семантикой.

Историческое состояние может использоваться во вложенной машине состояний внутри составного состояния. При первом запуске машины состояний историческое состояние означает в точности тоже, что и начальное: оно указывает на состояние, в котором находится машина в начале работы. Если в данной машине состояний используется историческое состояние, то при выходе из объемлющего составного состояния запоминается то состояние, в котором находилась вложенная машина при выходе. При повторном входе в данное составное состояние в качестве текущего состояния восстанавливается то состояние, в котором машина находилась при выходе. Проще говоря, историческое состояние заставляет машину помнить, в каком состоянии ее прервали прошлый раз и "продолжать начатое".
Историческое состояние имеет две разновидности.

*Поверхностное историческое состояние* запоминает, какое состояние было активным на том же уровне вложенности, на каком находится само историческое состояние.

*Глубинное историческое состояние* помнит не только активное состояние на данном уровне, но и на вложенных уровнях.

Рассмотрим пример из информационной системы отдела кадров. Работающий сотрудник, если все идет хорошо, пребывает в одном из двух взаимоисключающих состояний: либо он на работе (`inOffice`), либо в отпуске (`onVacations`). Но может случиться такая неприятность, как болезнь (`Ill`). Понятно, что заболевший сотрудник не на работе, но он и не в отпуске, болезнь — это еще одно состояние. Но в какое состояние переходит сотрудник по выздоровлении? Допустим, что в нашей информационной системе отдела кадров действует положение старого Комплекса законов о труде — если сотрудник заболел, находясь в отпуске, то отпуск прерывается, а по выздоровлении возобновляется. Для того, чтобы построить модель такого поведения, нужно воспользоваться историческим состоянием. В данном случае достаточно поверхностного исторического состояния, поскольку на данном уровне вложенности все состояния уже простые. На рис. 4.15 приведен соответствующий фрагмент машины состояний.

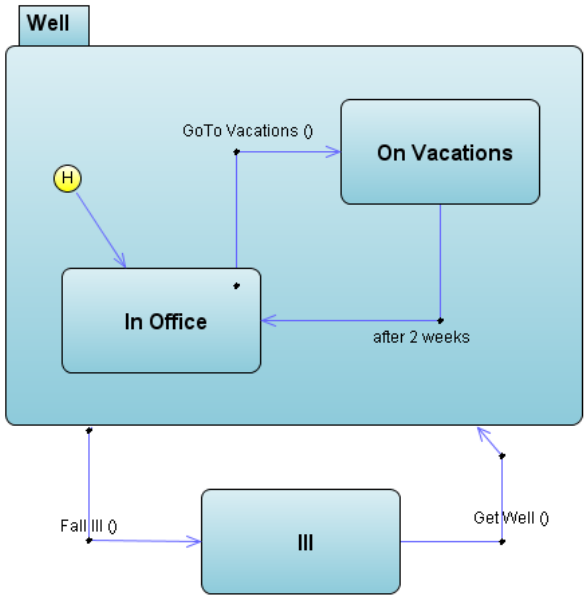



**Рис. 4.15. Историческое состояние**

Прежде чем погрузиться в описание следующей порции деталей машины состояний UML, давайте на минуту приподнимем голову и оглядим горизонт. Конечные автоматы — хорошее, теоретически проработанное и широко известное средство описания поведения.[94] Машины состояний — реализация в UML идеи конечных автоматов с различными добавлениями. В частности, составные состояния и вложенные машины состояний позволяют структурировать модель поведения. Казалось бы, все отлично — чего еще желать? Для тех примеров, которые мы пока что рассматривали, действительно, никаких других средств не нужно. Но что будет при описании действительно сложного поведения? Если в диаграмме состояний сотня состояний и десяток уровней вложенности? Они просто не поместятся на диаграмме нормальных размеров, а если поместятся, то их будет невозможно прочитать и понять. Если мы хотим описать на UML действительно сложное поведение, то мы должны иметь возможность сделать это по частям, используя принцип "разделяй и властвуй". И такая возможность в UML предусмотрена — это ссылочные состояния и состояния заглушки.

Выше мы дважды упомянули, что всякая машина состояний UML вложена в некоторое составное состояние (машина верхнего уровня вложена в искусственное составное состояние и именем `top`). Теперь можно объяснить, для чего это сделано — чтобы можно было сослаться на любую машину состояний по имени (конкретно, по имени составного состояния, в которое вложена машина).

| |
|---|
| *Ссылочное состояние* — состояние, которое обозначает вложенную в него машину состояний. |

[94] Некоторые ученые считают, что конечные автоматы суть наилучшее и едва ли не единственно возможное средство описания поведения, см. например, сайт http://is.ifmo.ru.

На диаграмме ссылочное состояние изображается в виде фигуры простого состояния с именем, которому предшествует ключевое слово `include`. Семантика ссылочного состояния заключается в следующем. Если на диаграмме присутствует ссылочное состояние, то это означает, что в модели вместо ссылочного состояния присутствует составное состояние (и, соответственно, вложенная машина состояний), на которое делается ссылка. Таким образом, на одной диаграмме мы можем представить поведение, нарисовав его "крупными мазками", т. е. в терминах составных состояний верхнего уровня, а на других диаграммах раскрыть содержание составных состояний, нарисовав соответствующие машины состояний и т. д. Критерий Дейкстры хорошего стиля программирования — "правильно написанный модуль должен помещаться на одной странице" — соблюдается.

Переходы между составными состояниями в UML можно подразделить на два типа: переходы внутрь и изнутри составных состояний (т. е. переходы, пересекающие границы составных состояний), и переходы, начинающиеся и заканчивающиеся на границах состояний. Вообще говоря, без переходов, пересекающих границы состояний, можно обойтись (равно как и без многого другого), но жалко — это удобная возможность. Если переход начинается и заканчивается на границе состояний, то вся работа вложенной машины состояний инкасулирована в составном состоянии, семантика переходов определена в табл. 4.3 и никаких проблем нет — ссылочного состояния достаточно для включения одной машины внутрь другой. Однако, если такие переходы есть (а мы договорились, что не хотим от них отказываться), то возникает проблема: грубо говоря, нам нужно провести стрелку с одной диаграммы на другую. Решением этой проблемы в UML являются состояния заглушки.

*Состояние заглушка* — это специальное состояние, которое обозначает в ссылочном состоянии некоторое вложенное состояние того составного состояния, на которое делается ссылка.

Звучит замысловато, но все очень просто: мы разрешаем себе показать в ссылочном состоянии необходимый нам минимум деталей той машины состояний, которая скрыта в ссылочном состоянии. А нужны нам только имена вложенных состояний, в которые или из которых делается переход. На диаграмме состояние заглушка изображается в виде короткой вертикальной черты внутри фигуры ссылочного состояния и с именем соответствующего вложенного состояния. У этой черты начинается или заканчивается стрелка перехода, пересекающего границу ссылочного состояния.[95]

Приведем пример из информационной системы отдела кадров, чтобы проиллюстрировать приведенные сухие определения. В этом примере мы опять рассматриваем жизненный цикл сотрудника на предприятии (см. рис. 4.3), но раскрываем делали поведения объекта `Person`, находящегося в самом важном для предприятия состоянии — `Employee`. В предыдущих примерах (см. рис. 4.5, 4.8–4.11, 4.15) мы раскрывали разные детали машины состояний сотрудника. Здесь мы покажем, как с помощью ссылочного состояния можно собрать все заготовки в цельную и полную модель поведения сотрудника. На рис. 4.16 приведена

[95] Если бы такого перехода не было, то и состояние заглушка было бы не нужным.

диаграмма верхнего уровня, описывающая поведение в целом, без деталей. Состояние `Employee` — ссылочное, что подразумевает наличие другой диаграммы, на которой раскрыта внутренняя структура состояния `Employee`. Состояние заглушка `inOffice` указывает, что внутри составного состояния `Employee` есть вложенное состояние `inOffice`. Данное состояние выявлено на диаграмме верхнего уровня, чтобы подчеркнуть, что переходы по событию `fire` ведут именно в это вложенное состояние, а не в другое (при приеме сотрудник сначала обязательно должен попасть в состояние "на работе" и только потом может уйти в отпуск, заболеть и т. д.).

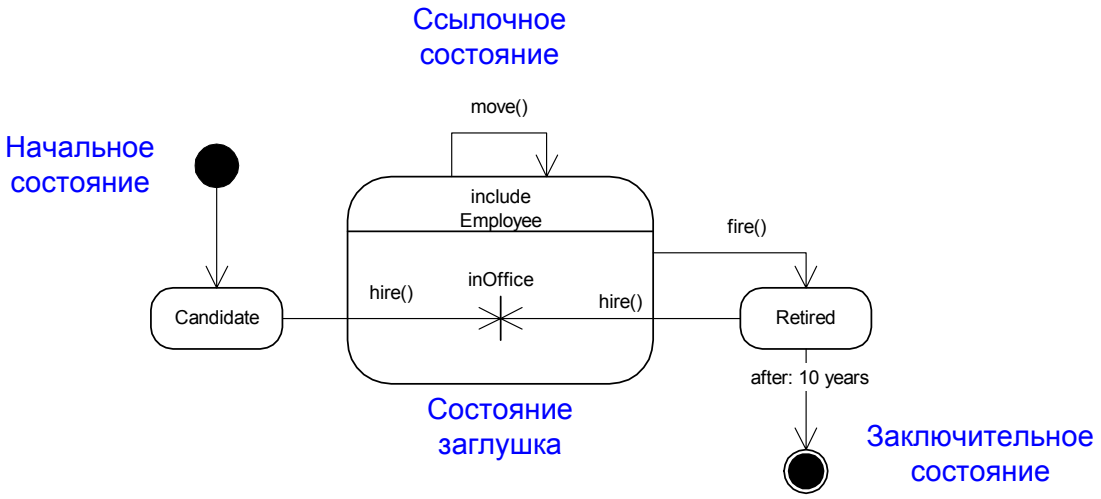



**Рис. 4.16. Ссылочное состояние и состояние заглушка**

На другой диаграмме (рис. 4.17) раскрывается внутренняя структура состояния Employee.

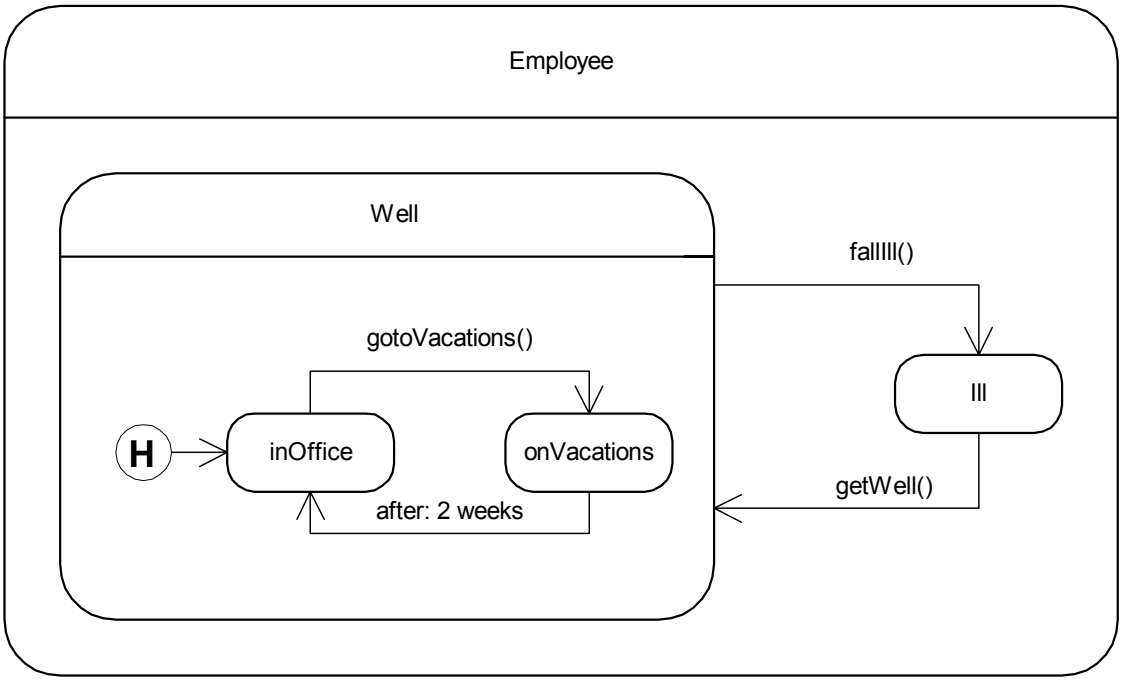



**Рис. 4.17. Составное состояние, раскрывающее ссылочное состояние**

Здесь мы сразу показали все вложенные состояния `Employee`, как простые, так и составные, поскольку диаграмма получается достаточно компактной и обозримой. Но если бы нам нужно было учесть больше деталей поведения, например, командировки, особые режимы работы (скажем, работу в праздничные дни), работу по графику (сутки работы — трое выходных), различные типы отпусков (очередной, за свой счет, по уходу за ребенком, учебный), короче говоря, если бы все нужные детали перестали помещаться на диаграмму, то нужно было бы ввести еще несколько ссылочных состояний и раскрыть их детали на отдельных диаграммах.[96] Фактически, при описании поведения средствами UML, вполне применим метод пошагового уточнения, описанный в разд. 3.3.1.

Для возможности практического использования метода пошагового уточнения при описании сложного поведения с помощью машин состояний в UML 2.0 явным образом введено понятие вложенной машины состояний вместо понятия ссылочного состояния и заглушки.

На диаграмме верхнего уровня указывается вложенная машина состояний как простое состояние, а на диаграмме нижнего уровня вложенная машина состояний раскрывается как составное состояние. Вместо понятия заглушки указываюися точки входа и выхода на границе вложенной машины состояний. Уточненное в UML 2.0 понятие вложенной машины состяний представляется нам настолько простым и естественным, что вместо детальных пояснений мы просто сошлемся на рис. 18 и 19, где представлен еще один пример.

---

[96] Детальная модель поведения объекта "сотрудник" в промышленной информационной системе отдела кадров, по нашему опыту, содержит примерно десяток диаграмм и использует три-четыре уровня вложенности ссылочных состояний.

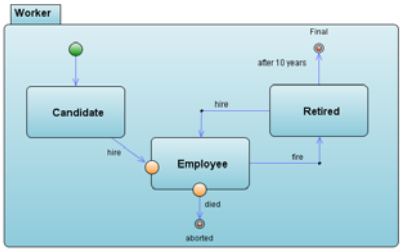



**Рис. 4.18. Составное состояние, использующее вложенную машину состояний**

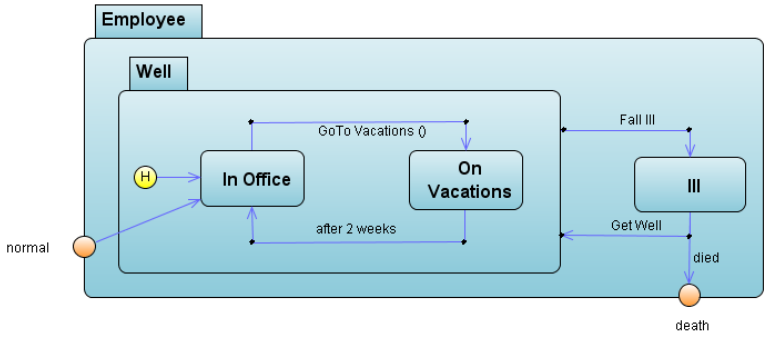



**Рис. 4.19. Составное состояние, раскрывающее вложенную машину состояний**

В заключение данного раздела, мы приводим метамодель состояний UML (рис. 4.20), на которой показаны (с некоторыми упрощениями), рассмотренные в этом разделе понятия.

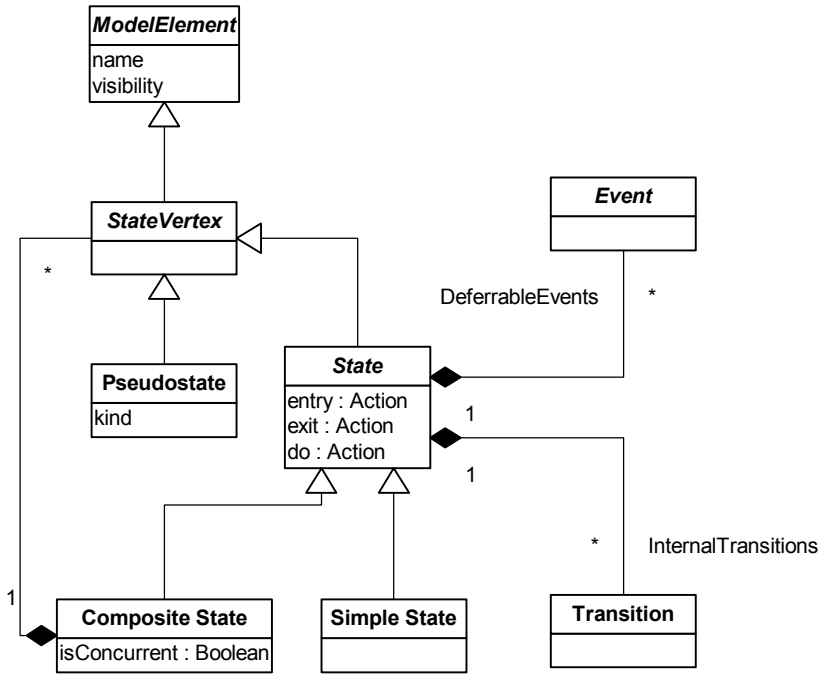



**Рис. 4.20. Метамодель состояния**

### *4.2.4. События*

Различные типы состояний, рассмотренные в предыдущем разделе, позволяют задавать структуру машины состояний. Но основная семантическая нагрузка при описании поведения падает на переходы различных типов. Помимо структурных составляющих — исходного и целевого состояний — переход может быть нагружен событием перехода, сторожевым условием и действиями на переходе. Детали описания сторожевых условий рассмотрены в разд. 4.2.2, детали описания действий рассмотрены в разд. 4.3.1, а в этом разделе рассматриваются детали описания событий.

Вообще говоря, можно различать события внешние — которые возникают вне системы, в мире действующих лиц и обрабатываются в системе и события внутренние — которые возникают внутри системы и там же обрабатываются. В UML не специальных средств для различения событий этих типов, что, но нашему мнению, является недостатком.

В UML используются четыре типа событий:

- событие вызова,
- событие сигнала,
- событие таймера,
- событие изменения.

| *Событие вызова* — это событие, возникающее при вызове операции класса. |
|---|

Если событие вызова используется как событие перехода в машине состояний, описывающей поведение класса, то класс должен иметь соответствующую операцию. Событие вызова — наиболее часто используемый тип событий перехода. В большей части рассмотренных примеров в качестве событий перехода использовались именно события вызова. Поскольку событие вызова — это вызов операции, то оно может иметь аргументы, как всякая операция. Значения аргументов могут использоваться в действиях перехода. Если операция возвращает значение, то этот факт отмечается с помощью действия возврата (см. разд. 4.3.1) в последовательности действий данного перехода.

Допустим, что некоторая операция класса реализована (то есть соответствующее данной операции поведение описано) с помощью события вызова машины состояний данного класса. Пусть данная операция вызвана. Тогда возможны два варианта: либо соответствующий переход срабатывает (т. е. в активном состоянии определен переход с данным событием вызова и сторожевое условие выполнено), либо не срабатывает. В каждом случае поведение зависит от того, возвращает или не возвращает значение вызываемая операция. Если операция не возвращает значения, то ошибки нет в любом случае: если переход срабатывает, то выполняются действия на переходе, автомат переходит в новое состояние и в этом заключается выполнение вызванной операции; если же переход не срабатывает, то ничего не происходит и управление немедленно возвращается в место вызова, причем это не считается ошибкой. Если же данная операция возвращает значение, то если переход не срабатывает или если переход срабатывает, но на переходе не возвращается значение или возвращается значение неподходящего типа, то это считается ошибкой, поскольку поведение, описываемое машиной состояний, не соответствует спецификации операции.

Рассмотрим пример из информационной системы отдела кадров. Допустим, что в классе `Person` определена операция `move(newPos:Position):Boolean`, которая переводит работающего сотрудника на новую должность `newPos` и возвращает значение `true`, если перевод успешно произведен. Пусть в классе `Position` определены операции `isFree():Boolean`, которая проверяет, свободна ли должность и `occupy(person:Person)` — назначает сотрудника на должность. Тогда типичное поведение операции `move` можно описать диаграммой состояний, приведенной на рис. 4.21.

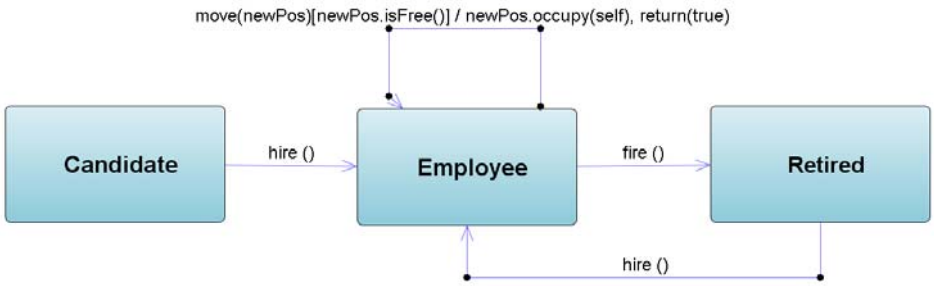



**Рис. 4.21. Событие вызова**

Однако такое описание поведения, хотя и правильно определяет, что делает операция `move` если "все хорошо", но не является достаточно надежным. Действительно, если операция `move` будет вызвана в тот момент, когда сотрудник находится в состоянии `Candidate` или `Retired`, или же если должность `newPos` окажется занятой, то возникнет ошибка, поскольку требуемое по спецификации операции `move` логическое значение не будет возвращено. Диаграмма состояний на рис. 4.22 описывает поведение операции `move` более надежно, поскольку обеспечивает возврат значения и в том случае, когда "не все хорошо".

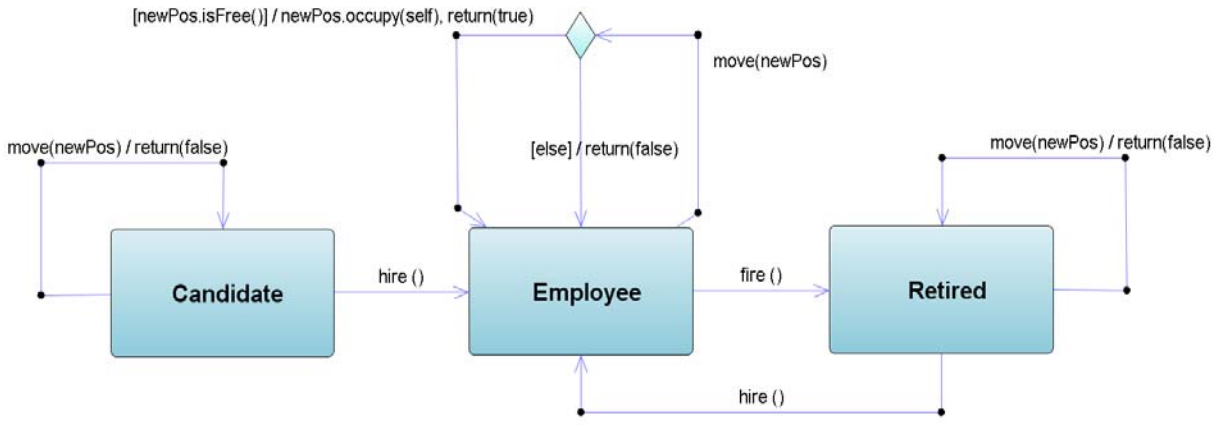



**Рис. 4.22. Событие вызова с учетом всех вариантов поведения**

Помимо иллюстрации понятия события вызова в UML последний пример может послужить, по нашему мнению, поводом для обсуждения понятий предусловия и постусловия операций в UML.

*Предусловие* — это условие, налагаемое на входные данные операции, которое должно выполняться до вызова операции.

Если предусловие операции нарушено, то корректное выполнение операции не гарантируется.

*Постусловие* — это условие, связывающее входные и выходные данные операции, которое должно выполняться после вызова операции.

Если постусловие операции оказывается нарушенным, то значит реализация операции содержит ошибку. В UML предусловия и постусловия операций предлагается описывать при моделировании структуры, в виде ограничений со стереотипами «`precondition`» и «`postcondition`», соответственно. По сути, однако, предусловия и постусловия операций следует отнести к моделированию поведения. Впрочем, как мы уже не раз отмечали, само деление моделирования на структурное и поведенческое носит условный характер: моделирование — это единый итеративный процесс.

Определение предусловий и постусловий операций — чрезвычайно сильное средство моделирования, но рекомендации по его применению неоднозначны. Например, в только что разобранном случае операции `move`: условие `newPos.isFree()` является частью предусловия операции `move` или нет? Можно ответить "да" и тогда кто-то должен позаботиться о выполнении этого условия, можно ответить "нет" и тогда есть риск, что проверка не будет выполнена и в результате перевода сотрудника на занятую должность будет нарушена целостность базы данных. Чем слабее предусловие, тем шире область применения операции, она более универсальна. Но универсальность не всегда благо: иногда это никому не нужная трата ресурсов. Другой вопрос: как следует использовать предусловие и постусловие операции, если они заданы? Весьма авторитетные ученые рекомендуют доказывать (в математическом смысле) правильность реализации операции. Спору нет, доказательное программирование — отличная вещь, но где же набрать необходимое количество программистов, умеющих доказывать свои программы?[97] Сугубые практики советуют вручную вставлять в начало программы проверку предусловия, а в конец — проверку постусловия. Это, конечно, заметно повышает надежность программы, для библиотек общего пользования фактически так всегда и делается, но в приложении типа информационной системы отдела кадров в 9 случаев из 10 такие проверки будет излишними и заметно ухудшат эффективность приложения. Короче говоря, по данному вопросу сколько авторов, столько и взаимоисключающих мнений. И все правы.

У нас тоже есть совет — не универсальный и не масштабируемый. Он применим в случае разработки простых приложений для бизнеса, в которых надежность важна, но не критична, на доказательства нет времени и денег, а операции просты. Совет состоит из двух частей. Во-первых, какие-то предусловия и постусловия нужно выписывать всегда, а вот тратить силы на их анализ и совершенствование не обязательно. Во-вторых, предусловие и постусловие нужно поместить в код

[97] И сколько будут стоить такие программисты?

программы до и после вызова операции, соответственно, в форме утверждений (см. ниже врезку "Доказательное программирование"). Разумеется, это не повысит надежность программы — если ошибка в коде есть, она там останется и приложение завершится аварийно, но не так болезненно и ошибку легче будет локализовать.

### Доказательное программирование

Идея автоматического *доказательства правильности* (или *верификации*) программ высказана давно. К сожалению, известно, что в общем случае эта задача не разрешима и не проще, чем задача автоматического синтеза императивной программы по декларативной спецификации (см. врезку "Автоматический синтез программ" в разд. 1.2.3). Существует целый ряд частных случаев,[98] в которых задача верификации оказывается разрешимой, но эти частные случаи таковы, что в практических ситуациях (недостаточно формальные спецификации, недостаточно формализованная семантика используемого императивного языка программирования и пр.) автоматическая верификация оказывается фактически неприменимой.

Апологеты доказательного программирования ратуют за *ручное* доказательство правильности программ, как альтернативы тестированию и отладке. При этом приводятся весьма поучительные и ценные примеры и приемы таких доказательств (см., например, блестящую книгу Э. Дейкстры *Дисциплина программирования*). К сожалению, ручная верификация практически возможна только для очень маленьких программ, просто в силу ограниченности способностей программистов к построению доказательств.

Ручное доказательство правильности целесообразно использовать для верификации часто используемых небольших *образцов проектирования*. Образец, в отличие от конкретной программы, практически невозможно надежно отладить тестированием, потому что невозможно заранее определить все возможные варианты и контексты будущего использования образца, а значит невозможно построить представительный набор тестов.

Тем не менее, один из приемов, связанных с идеей верификации, целесообразно использовать, даже если полная верификация программы и не проводится. А именно, при верификации в текст программы вставляются промежуточные утверждения (assert), которые должны удовлетворяться при выполнении *правильной* программы. Вставка утверждений в программу иногда называется *аннотированием* программы. В простейшем случае утверждения представляют собой эффективно вычислимые предикаты над значениями переменных программы. Например, предусловия и постусловия операций, инварианты цикла, ограничители числа итераций плохо сходящихся процедур и т. п. Если программист *знает*[99] какое-либо (достаточно сильное) условие на значения переменных в данной точке программы,

[98] Например, если известны инварианты всех циклов.

[99] Это знание может быть почерпнуто из спецификации, из опыта или из частичной ручной верификации.

то настоятельно рекомендуется перенести это знание в код программы. Существуют три основных формы использования утверждений.

- Запись утверждения в виде комментария. Так или иначе эту форму используют практически все программисты. Например, комментарий к определяющему вхождению параметра процедуры содержит, как правило, неформальное описание области допустимых значений аргумента. Систематическое выписывание в комментариях нетривиальных утверждений о текущих значениях переменных резко увеличивает читабельность программы и рекомендуется к использованию наряду с другими формами внесения утверждений в текст.
- Запись утверждения в формальном синтаксисе. Эта форму настоятельно рекомендуется использовать, если система программирования поддерживает работу с утверждениями. Например, многие современные системы программирования поддерживают следующий механизм работы с утверждениями. Вставленное в текст программы формальное утверждение проверяется и, в случае невыполнения, возбуждается указанный сигнал или исключительная ситуация.
- Запись утверждения в виде ловушки.[100] Ловушка — это фрагмент кода, предназначенный для перехвата и обработки ошибок (исключительных ситуаций). Например, это может быть условный оператор, который проверяет невыполнение утверждения и немедленно обрабатывает исключительную ситуацию или посылает сообщение системе обработки исключительных ситуаций.

Во всех приведенных формах наличие утверждений не превращает, конечно, неправильную программу в правильную. Однако использование утверждений может сильно увеличить потребительскую ценность даже неправильной программы. Одно дело, когда программа аварийно завершается или, хуже того, молча выдает неправильный результат и совсем другое дело, когда программа хотя и не работает как должно, но вежливо информирует об этом пользователя.

*Событие сигнала* — это событие, возникающее, когда послан сигнал.

Чтобы разъяснить событие сигнала в UML, нужно рассмотреть прежде всего саму концепцию сигнала. Здесь опять имеет место пересечение моделирования структуры и поведения: определяются сигналы в структурной части модели, а используются в поведенческой.

Синтаксически *сигнал* (в UML) — это объект класса со стереотипом «`signal`». Семантически *сигнал* — это именованный объект, который создается другим объектом (отправителем) и обрабатывается третьим объектом (получателем).

[100] Фактически, это ручная реализация идеи обработки утверждений в тех системах программирования, которые не поддерживают автоматическую обработку утверждений.

Сигнал, как объект, может иметь атрибуты, т. е. параметры сигнала. Сигнал может иметь операции. Одна из них считается определенной по умолчанию. Она имеет стандартное имя `send` и параметр, являющейся множеством объектов, которым отправляется сигнал. Объявлять эту операцию не нужно. Можно объявлять другие операции, которые служат для доступа к значениям атрибутов (аргументам) сигнала, но это не обязательно.

В сущности, концепция сигнала в UML принадлежит к числу наиболее фундаментальных и имеет ясную и строго определенную семантику. Объект, являющийся отправителем, обращается к классификатору сигнала (вызывает операцию `send`), указывая аргументы сигнала (значения атрибутов) и целевое множество объектов, которым должен быть отправлен сигнал. После этого объект-отправитель продолжает свою работу — дальнейшее его не касается. Как правило, целевое множество содержит один элемент, но может не содержать элементов (это не считается ошибкой) или может содержать много элементов (это называется *широковещательным сигналом*). Объекты, которым отправляется сигнал, обязаны иметь операцию для получения и обработки сигнала. Такая операция имеет стереотип «`signal`», ее имя совпадает с именем классификатора сигнала, а имена и типы параметров совпадают с именами и типами атрибутов сигнала. Операция получения сигнала не может возвращать результат. Классификатор сигнала создает новый объект — экземпляр сигнала и отправляет его копии всем объектам целевого множества, т. е. вызывает в объектах целевого множества операции приема сигнала с указанными значениями аргументов.

Поскольку сигналы в UML являются классификаторами, они могут участвовать в отношении обобщения. Сигнал, являющийся подклассом, наследует все атрибуты (параметры) суперкласса и, может быть, имеет свои собственные. Суперклассом сигнала не может быть не сигнал. Принцип подстановочности при обобщении сигналов имеет следующий смысл: если на переходе указано событие сигнала, то переход возбуждается

Сигналы подходят для описания всех видов взаимодействия объектов — как синхронных, так и асинхронных, как в монолитных приложениях, так и в распределенных. Авторы UML отмечают, что обычный вызов операции можно рассматривать как частный случай посылки сигнала: вызывающий объект отправляет сигнал вызова (с аргументами) и дожидается ответного сигнала о завершении выполнения (с возвращаемым значением). Такой подход возможен, но, на наш взгляд, понятие обычного вызова операции и так уже устоялось и можно обойтись без его сведения к более общей концепции.

В UML имеется один важный частный случай сигналов — исключения, которые определяются с помощью стереотипа «`exception`». *Исключения* во всем подобны сигналам — их назначение состоит в том, чтобы подчеркнуть в модели то обстоятельство, что для реализации должен использоваться встроенный механизм обработки исключений, который имеется в большинстве современных объектно-ориентированных систем программирования. В частности, для исключений (равно как и для сигналов) действует правило "перехвата": исключение, посланное объекту, обрабатывается операцией обработки данного исключения, определенной в ближайшем по иерархии обобщения суперклассе для класса данного объекта.

Перейдем к рассмотрению нотации, связанной с сигналами, исключениями и событиями сигнала. На диаграмме классов сигнал определяется в форме класса со стереотипом «signal» или «exception». Этот класс связывается с отправителем зависимостью со стереотипом «send» и связывается с получателем объявлением в получателе операции со стереотипом «signal».

Рассмотрим пример из информационной системы отдела кадров. Допустим, что мы решили случай занятости целевой должности при выполнении операции move перевода сотрудника обрабатывать как исключительную ситуацию. Ответственность за обработку исключительной ситуации мы решили возложить на класс Company.[101] Такое решение на диаграмме классов можно изобразить следующим образом (рис. 4.23).

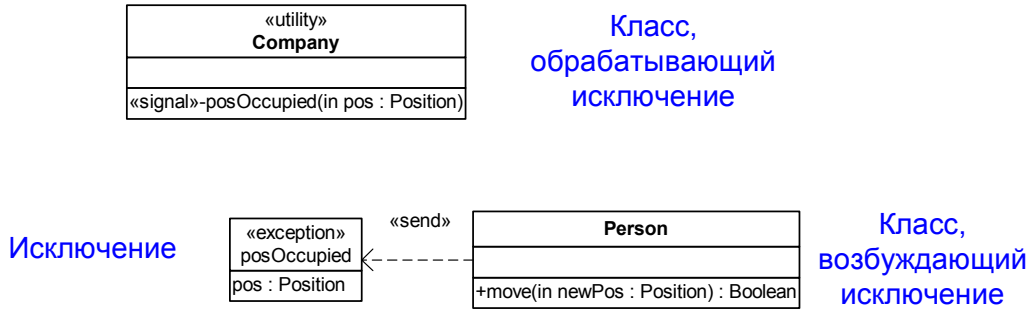



**Рис. 4.23. Описание сигнала на диаграмме классов**

Вернемся теперь в мир машин состояний. Принятое решение о введении исключения на диаграмме состояний класса Person может быть отражено, как показано на рис. 4.24. Здесь в случае занятости целевой позиции выполняется действие по посылке сигнала, которому мы для ясности предпослали ключевое слово send, хотя это и не обязательно.

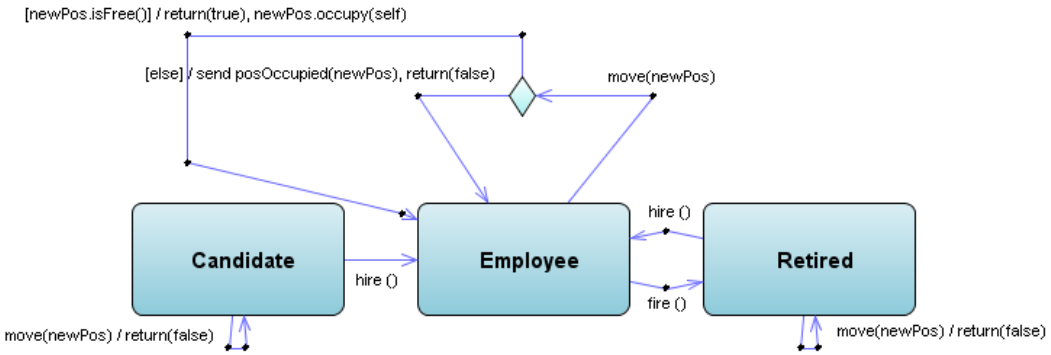



**Рис. 4.24. Посылка сигнала в машине состояний**

---

[101] Решение спорное, оно отражает личные авторские предпочтения: концентрировать обработку исключительных ситуаций в одном месте, желательно в глобальной службе общего назначения. Альтернативный вариант — помещать обработку исключительных ситуаций как можно ближе к месту их потенциального возникновения.

В машине состояний объекта класса, ответственного за обработку исключительных ситуаций, исключение выступает как событие сигнала, возбуждающее переход. Синтаксис события сигнала ничем не отличается от события вызова (потому что обработка события сигнала это тоже выполнение операции). Мы не стали детализировать машину состояний класса `Company`, ограничившись одним интересным нам сейчас одним переходом с событием сигнала (рис. 4.25). На этом переходе выполняется не интерпретируемое действие (см. разд. 4.3.1) `putLog` — мы полагаем необходимым в любом случае по крайней мере записать в протокол выполнения приложения факт срабатывания перехода по исключению.

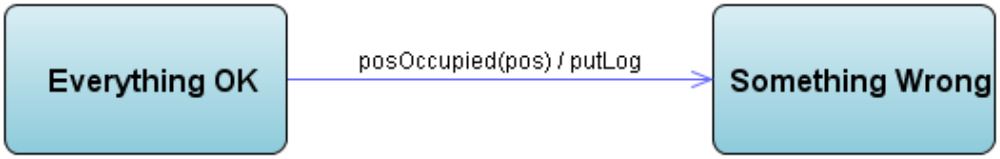



**Рис. 4.25. Переход по событию сигнала**

Полагаем, что в первом приближении события сигнала описаны достаточно подробно. Мы еще несколько раз вернемся к обсуждению сигналов в разд. 4.3.

| |
|---|
| *Событие таймера* — это событие, которое возникает, когда истек заданный интервал времени с момента попадания автомата в данное состояние. |

Синтаксически событие таймера записывается с помощью ключевого слова `after`, за которым указывается выражение, доставляющее длину интервала времени. Семантически событие таймера означает следующее. Подразумевается, что у состояния имеется таймер, который сбрасывается в 0, когда автомат переходит в данное состояние (напомним, что автомат считается перешедшим в состояние, когда закончено выполнение всех действий, предписанных переходом). Таймер ведет отсчет времени. Если до истечения указанного интервала времени сработает другой переход, то событие таймера не возникает. Когда указанный интервал времени истекает, наступает событие таймера и возбуждается соответствующий переход. Если переход срабатывает, то автомат переходит в новое состояние. Заметим, что если переход по событию таймера является переходом в себя, то таймер опять сбрасывается в 0, а если переход по событию таймера является внутренним переходом, то таймер продолжает отсчет. Если переход по событию таймера не срабатывает (из-за ложности сторожевого условия), то событие таймера теряется и таймер продолжает отсчет времени, так что позже может сработать другой переход по событию таймера с большим интервалом времени. Событие таймера не может быть отложено.[102]

Мы уже привели несколько примеров использования события таймера на переходах в машинах состояний (см. рис. 4.13–4.17). Приведем еще один (рис. 4.26). В этом примере мы определяем, что по истечении 10 лет объект, представляющий уволенного сотрудника (запись в базе данных) уничтожается,

[102] Забегая вперед заметим, что таймеры параллельных вложенных состояний независимы.

если только этот объект не помечен специальным образом (сторожевое условие `keepForever`).

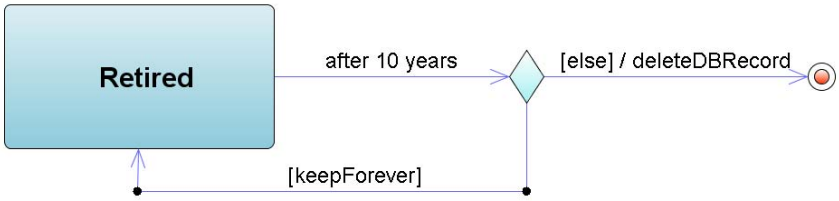



**Рис. 4.26. Переход по событию таймера со сторожевым условием**

| |
|---|
| *Событие изменения* — это событие, которое возникает, когда некоторое логическое условие становится истинным, будучи до этого ложным. |

Синтаксически событие изменения записывается с помощью ключевого слова `when`, за которым указывается логическое выражение (условие). Семантически событие изменения означает следующее. Подразумевается, что в системе имеется механизм, работающий как демон, который возбуждает событие, если в результате изменения состояния системы изменяется значение логического выражения. Если выражение, являющееся аргументом события изменения, принимает значение `true` (имея до этого значение `false`), то переход возбуждается. Если выражение имеет значение `true` в тот момент, когда автомат переходит в данное состояние, то переход сразу возбуждается. Если переход срабатывает, то автомат, как обычно, переходит в новое состояние. Если переход не срабатывает, то событие изменения теряется. При этом, если условие продолжает оставаться истинным, то нового события изменения не возникает. Для того, чтобы снова возникло событие изменения, нужно, чтобы условие стало сначала ложным, а потом истинным.

### Демоны в программировании

Обычно демоном называется программа, которая выполняет свою функцию без явного вызова со стороны пользователя или той программы, которая пользуется услугами демона. Название обусловлено тем, что демон срабатывает как бы сам, по собственной инициативе. Разумеется, это только видимый для пользователя эффект. На самом деле демоны запускаются с помощью механизма, скрытого в операционной системе. Чаще всего такой механизм основан либо на периодической проверке состояния ("не пора ли вызвать демона?"), либо на встраивании вызова демона в процедуры реакции на определенные события. Исторически это понятие впервые было использовано, видимо, в операционной системе UNIX.

На первый взгляд кажется, что событие изменения — это нечто крайне неэффективное. Действительно, как узнать, что значение логического выражения изменилось? Кажется, что нужно непрерывно его перевычислять, чтобы не прозевать момент. На самом деле, это совершенно не обязательно. Вполне возможен эффективный демон, и вот один из способов реализации. Рассмотрим, от чего зависит значение логического выражения? От значений входящих в него переменных. Таким образом, значение логического выражения может измениться

только по причине (и одновременно!) изменения значения одной или нескольких входящих в него переменных. В результате чего изменяется значение переменной? В результате присваивания. Большинство современных систем программирования поддерживают указатели от использующих вхождений переменной к определяющему. Все что нужно сделать — это добавить обратные указатели от определяющего вхождения к использующим, причем достаточно не ко всем, а только к тем, которые входят в интересующие нас условия (помеченные ключевым словом `when`). Далее нужно, чтобы при выполнении присваивания переменной система проверяла, нет ли указателя от этой переменной к контролируемому выражению. Если есть, нужно вычислить значения выражения. Если оно окажется равным `true`, то нужно возбудить соответствующее событие изменения. Все это вполне может сделать разумный компилятор за счет небольших добавлений к коду операторов присваивания. На самом деле и без специального компилятора можно обойтись. Схема реализации, которою мы только что описали, является частным случаем применения образца проектирования `subscribe publish`, так что если нужно, события изменения нетрудно запрограммировать и вручную.

**ЗАМЕЧАНИЕ**

> События, которые должны быть привязаны к физическому времени, а не к относительному времени пребывания в состоянии, задаются как события изменения, а не как события таймера.

Рассмотрим элементарный пример из информационной системы отдела кадров: по достижении определенного возраста сотрудник увольняется на пенсию (рис. 4.27).

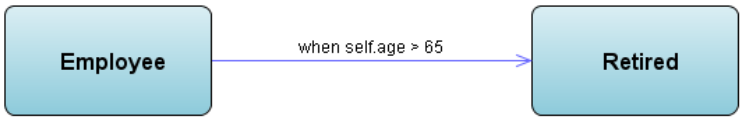



**Рис. 4.27. Переход по событию изменения**

В конце раздела мы приводим наш вариант метамодели событий UML (рис. 4.28), которая получилась не слишком сложной.

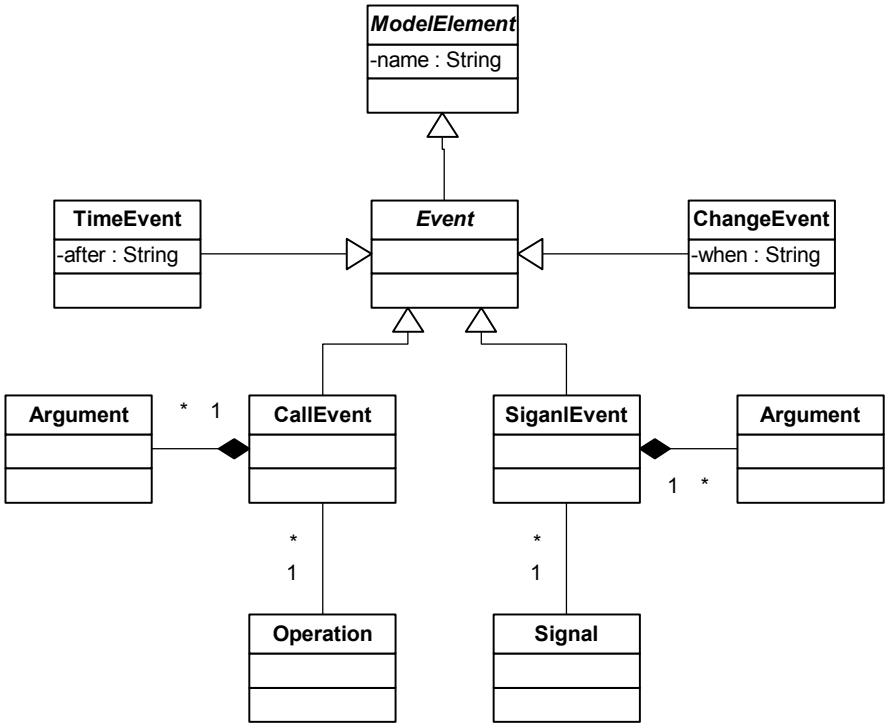



**Рис. 4.28. Метамодель событий**

## *4.2.5. Протокольный автомат*

В UML 2.0 появилось еще одно важное применение аппарата машин состояний.[103] Эта так называемая протокольная машина состояний, или протокольный автомат.

Протокольный автомат задается также как и любая другая машина состояний, но на переходах нет действий. Для указания того, что машина состояний является протокольным автоматом используется ключевое слово `protocol`.

Смысл использования протокольного автомата состоит в следующим. Допустим, имеется объект (или компонент), который предоставляет для использования некоторый интерфейс. С точки зрения объектно-ориентированного программирования здесь должна стоять точка: мы описали набор операций, предоставляемых для использования, что еще нужно? На самом деле такого объектно-ориентированного описания интерфейса «изнутри» бывает недостаточно. Операции, предоставляемые интерфейсом, часто не являются независимыми: некоторые последовательности операций являются допустимыми, и объект должен их выполнять, но некоторые последовательности не являются допустимыми и объект не может и не должен их выполнять. Таким образом, встает задача описания допустимых и недопустимых последовательностей вызовов операций объектов, то есть описания интерфейса «снаружи». Если рассматривать вызовы операций как символы, то каждый объект определяет некоторый *язык*, состоящий из допустимых последовательностей символов. Конечный автомат — одно из лучших известных средств задания языков. Именно оно используется в UML под названием протокольный автомат.

[103] В UML это средство действительно появилось только в версии 2.0. Но вне UML конечные автоматы уже давно использовались для верификации протоколов.

Протокольный автомат ничего не делает,[104] он только проверяет допустимость последовательности операций, поэтому действия на переходах и при входе/выходе ему не нужны.

*Протокольный автомат* — это машина состояний, предназначенная для задания допустимых последовательностей вызовов операций и сигналов, получаемых объектом.

Рассмотрим пример. Допустим, что информационная система отдела кадров ведет учет сессий своих пользователей, например, с целью разграничения прав доступа к информации. Тогда в ней, наряду с другими операциями, должны быть предусмотрены операции `login` и `logout`, выполняемые в начале и в конце каждой сессии работы с системой (рис. 4.29). При этом все остальные операции (на рис. 4.29 опи условно обозначены `anyOp`) пользователь может выполнять только после того, как выполнена операция `login` и до того, как выполнена операция `logout`.

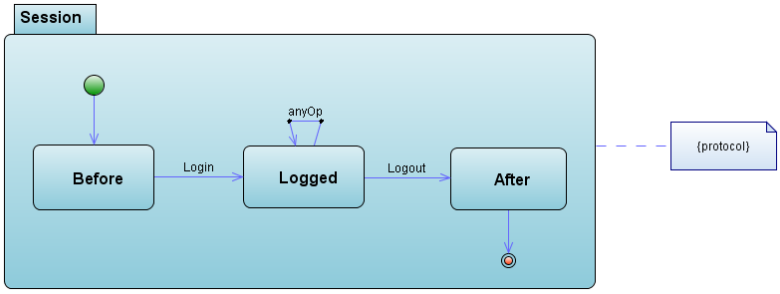



**Рис. 4.29. Протокольный автомат**

Состав и названия состояний могут быть общими как для протокольного автомата, так и для соответствующего ему поведенческого автомата. Но в общем случае это совершенного не обязательно — у этих машин состояний разное назначение!

## 4.3. Диаграммы деятельности

*Диаграммы деятельности* являются средством описания поведения в UML, причем их место в языке допускает некоторые разночтения. С одной стороны, семантика диаграммы деятельности определена через семантику машины состояний и в этом смысле диаграммы деятельности вторичны по отношению к диаграммам состояний.[105] С другой стороны, диаграммы деятельности UML очень близки по своей сути к блок-схемам, являющимся едва ли не старейшим средством описания поведения, и в этом смысле самодостаточны — считать их незначительным частным случаем чего-то было бы неестественно. С третьей стороны, диаграммы деятельности в UML снабжены дополнительными синтаксическими средствами, резко повышающими их выразительность и область применимости, даже по сравнению с машинами состояний UML. С четвертой стороны, диаграммы деятельности в наименьшей степени объектно-ориентированны и содержат наибольшее количество сознательно допущенных недоговоренностей и

[104] Иногда, чтобы подчеркнуть отличие от протокольного автомата, обычные автоматы, которые выполняют действия, называют поведенческими машинами состояний.

[105] Строго говоря, это утверждение верно для версий языка 1.x. В стандарте 2.0 появилось независимое определение семантики действий, но сути дела это обстоятельство не изменило.

произвольных элементов (это сделано для гибкости и расширения области применения). Короче говоря, наша позиция по отношению к этому средству не совсем определена. По поводу других средств UML мы позволяем себе давать хотя и субъективные (возможно, ошибочные), но достаточно однозначные рекомендации: "диаграммы классов следует применять для моделирования структуры…", "диаграммы состояний предназначены для описания поведения объектов, если оно зависит от истории…" и т. д. По поводу же диаграмм деятельности наши рекомендации более расплывчаты: диаграммы деятельности можно применять так-то и так-то, а можно не применять вовсе, и ничего страшного не случится. В этом разделе мы детально описываем все элементы диаграмм деятельности и приводим несколько разноплановых примеров, оставляя читателю возможность самому выработать свое личное отношение к диаграммам деятельности UML.

Основных сущностей и отношений, применяемых на диаграмме деятельности, в некотором смысле еще меньше, чем на диаграмме состояний (хотя, казалось бы, меньше уже некуда — на диаграмме состояний только состояния и переходы). Дело в том, что основная сущность на диаграмме деятельности является частным случаем простого состояния (состояние деятельности), а основное отношение — частным случаем простого перехода (переход по завершении). В тоже время всевозможных дополнительных обозначений и вариантов нотации на диаграмме деятельности еще больше, чем на диаграмме состояний. Поэтому, чтобы не затеряться в деталях, в следующем разделе мы обсудим содержание основных понятий, а уже в последующих разделах перейдем к примерам и картинкам.

### *4.3.1. Действие и деятельность*

В разделе 4.2.2 (и в других местах) мы использовали понятие действия, постулировав, что действие является атомарным, непрерываемым извне, безусловным и завершаемым. Действия используются на переходах и в состояниях машины состояний UML и играют там ключевую роль. Здесь мы более детально рассмотрим различные типы действий в UML и на основе противопоставления определим понятие деятельности, состояния действия и состояния деятельности, которые являются основными элементами в диаграммах деятельности. Это позволит нам выявить как сходство, так и различия диаграмм состояний и деятельности.

*Действие* в UML (версий 1.1–1.4) может быть одного из следующих типов:

- присваивание значения;
- вызов операции;
- создание объекта;
- уничтожение объекта;
- возврат значения;
- посылка сигнала;
- останов;
- не интерпретируемое действие.

Приведенный список (за исключением последнего пункта) очень похож на список основных выполняемых операторов в обычном языке программирования. Именно

на этот "эффект узнавания" и рассчитывали авторы UML. Действительно, подразумевается, что действие — это примитивная исполняемая конструкция языка программирования. Но UML *не* является языком программирования, поэтому семантика действий до конца в UML *не* определяется. Можно было бы взять один из распространенных языков программирования (или придумать еще один) и задать семантику выполнения действий UML через примитивы выбранного языка. UML стал бы визуальным[106] языком программирования. Но именно этого и хотели избежать авторы — оказать предпочтение одному языку в ущерб остальным.

### Универсальный язык программирования

К настоящему времени придумано и живет (т. е. имеет пользователей) очень большое число различных языков программирования. Несколько сотен по меньшей мере. Тенденции к унификации не наблюдается — процесс языкотворчества продолжается. Конечно, постоянно предпринимаются попытки "ввести" единый для всех язык программирования, опираясь либо на достоинства продвигаемого языка, либо на финансово-экономическую мощь заинтересованной в нем корпорации. Пока что такие попытки успеха не имели и, по нашему мнению, в обозримом будущем иметь не будут. Дело в том, что языки программирования существенно различны по той причине, что постоянно меняются области приложения программирования, возникают принципиально новые программные технологии, меняется и совершенствуется архитектура компьютеров, стремительно растут их количественные возможности. Фактор, который еще вчера был решающим при выборе языковой конструкции, завтра может ничего не значить. Отсюда не следует, что всеобщий язык программирования невозможен в принципе (например, у математиков практически есть общий язык). Но, по нашему мнению, сегодня это настолько маловероятно, что может не приниматься во внимание (математикам для выработки своего языка потребовалось несколько тысяч лет).

Главной целью UML является стать lingua franca для обмена идеями в сообществе архитекторов и разработчиков программных средств. Разработчики и архитекторы используют разные языки программирования и не собираются от них отказываться — это объективная реальность. Оказывать слишком явное предпочтение только одному языку недопустимо. Раз нельзя выбрать один язык, значит приходится отказываться от конкретного языка элементарных действий вообще. Поэтому семантика действий в UML определена несколько расплывчато и синтаксис не слишком строг. Расчет делается на то, что на абстрактном уровне всем все понятно, а инструмент, ориентированный на конкретный язык программирования, доопределит семантику действий так, как это принято в данном языке программирования. В табл. 4.4. приведены основные сведения о действиях в UML.

---

[106] Приставка visual, которую использует корпорация Microsoft в названиях своих средств программирования, строго говоря, не оправдана. В системе программирования Visual Studio действительно используется графический интерфейс пользователя, но и только. Код программы как был линейным текстом, так и остался. Мы склонны считать правомерным употребление термина "визуальный" только применительно к таким языкам, где код "рисуется" в виде схем, диаграмм и т. п.

**Таблица 4.3.** Действия

| Тип действия | Ключевое слово | Описание |
|---|---|---|
| присваивание значения | `:=` | Присваивание значения атрибуту объекта. После выполнения присваивания значение атрибута равно значению выражения в правой части |
| вызов операции | `call` | Вызов операции заданного объекта с заданными аргументами. Можно вызывать операцию сразу у нескольких объектов. В этом случае все они выполняются параллельно, а операция не должна возвращать значение. Операции вызываются синхронно: вызывающий объект ждет, когда выполнение операции закончится и ему вернут управление и, возможно, значение. |
| создание объекта | `create` | Создание и инициализация нового объекта заданного класса. Аргументы используются для инициализации атрибутов. |
| уничтожение объекта | `destroy` | Уничтожение заданного объекта и всех его составляющих, включенных отношением композиции. |
| возврат значения | `return` | Возврат значения в точку вызова операции. Это действие может применяться только внутри вызванной операции, например, на переходе, возбужденным событием вызова. Возвращаемых значений может быть несколько. |
| посылка сигнала | `send` | Создание нового экземпляра сигнала и отправка созданного сигнала заданному целевому множеству объектов. Целевое множество может не содержать элементов, содержать один элемент или несколько элементов. Сигналы посылаются асинхронно: посылающий объект не ждет, когда объекты целевого множества получат сигналы, а сохраняет свой поток управления и продолжает работу. |
| останов | `terminate` | Прекращение работы (принудительная остановка) машины состояний объекта и уничтожение этого объекта. |
| не интерпретируемое действие | любой текст | Любое действие, не определенное в UML. Например, исполняемая конструкция (оператор), целевого языка программирования, поддерживаемого инструментом или же текст на естественном языке. |

Еще одно средство, относящееся к действиям UML, называется повторитель.

*Повторитель* — это выражение, предписывающее выполнить действие несколько раз (может быть, ноль раз).

Синтаксически повторитель записывается подобно сторожевому условию, в квадратных скобках, перед которыми ставится звездочка:

```
* [ повторитель ] действие
```

Синтаксис выражения, определяющего число или условие повторений действия, в UML не предписывается, подразумевается, что это нечто вроде заголовка цикла целевого языка программирования.

Напомним еще раз, что последовательность действий, в том числе определенная повторителем, рассматривается как единое действие и обладает основными признаками действия: атомарностью и завершимостью. Прежде чем переходить к

обсуждению деятельности и других родственных понятий, завершим разговор о действиях нашим вариантом метамодели действий (рис. 4.30).

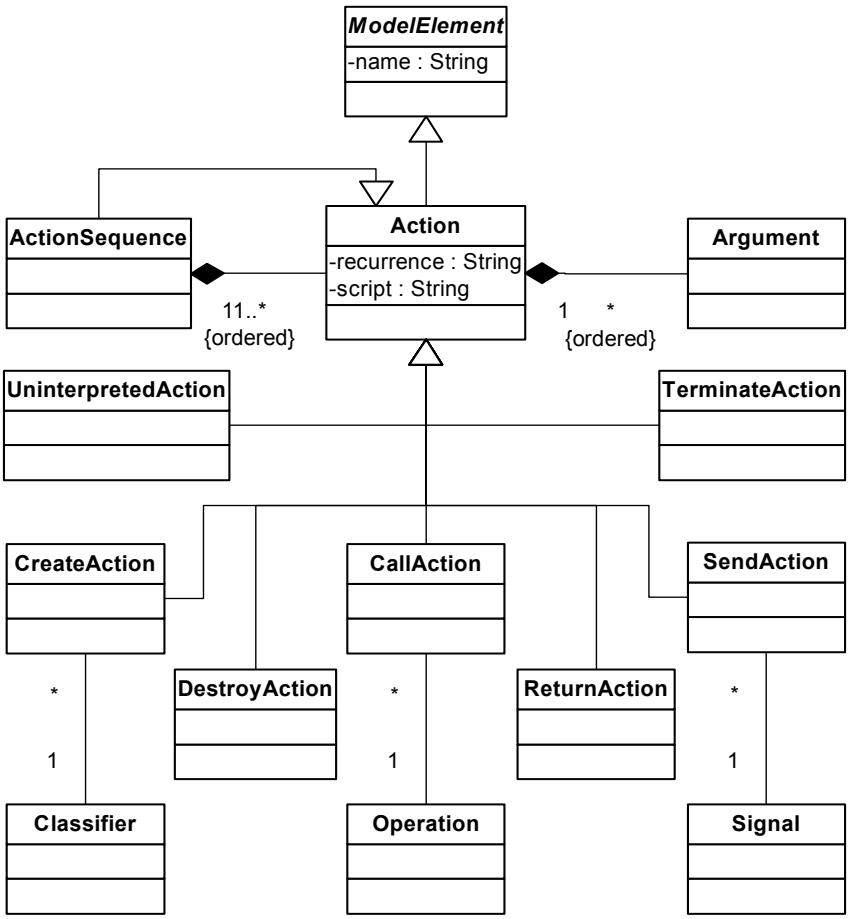



**Рис. 4.30. Метамодель действий**

*Деятельность* в UML — это средство структурной композиции действий.

Деятельность в UML моделирует то же, что и действие, т. е. какую-то содержательную активность во время работы системы; в этом смысле деятельность подобна действию, но деятельность противопоставляется действию по всем характеристическим признакам. В табл. 4.4. проведено сопоставление понятий действие и деятельность в UML.

**Таблица 4.4.** Сопоставление действия и деятельности

| Характеристика | Действие | Деятельность |
|---|---|---|
| Внешнее событие | Не прерывает выполнения | Может прервать и завершить выполнение |
| Завершаемость | Всегда завершается самостоятельно | Может продолжаться неограниченно долго |
| Внутренняя структура | Не моделируется в UML | Может быть раскрыта в отдельной диаграмме |
| Время выполнения | Пренебрежимо мало | Продолжительное |

Теперь наконец мы может точно[107] и кратко определить основные сущности на диаграмме деятельности UML. Таковых две:

- *состояние действия* — это состояние, внутренняя активность (разд. 4.2.1) которого является действием;
- *состояние деятельности* — это состояние, внутренняя активность которого является деятельностью.

*Состояние действия* не имеет специальной нотации и изображается на диаграмме как простое состояние с действием на входе. Вообще говоря, понятие состояния действия не имеет глубокого смысла: без него вполне можно обойтись. Действительно, при переходе в состояние действия выполняется действие на входе (которое атомарно и мгновенно) и немедленно после этого выполняется переход по завершении (см. первое примечание в разд. 4.3.2). Такую ситуацию можно промоделировать простым переходом, как показано на рис. 4.31. На этом рисунке на фрагменте диаграммы слева использовано состояние действия, а справа приведен эквивалентный фрагмент без состояния действия.

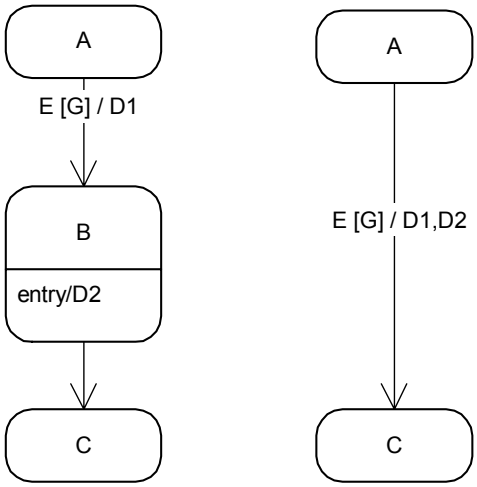



**Рис. 4.31. Элиминация состояния действия**

Пожалуй, единственной причиной, по которой состояние действия оставлено в UML, является стремление угодить вкусам тех, кто привык на блок-схемах операторы присваивания писать внутри прямоугольников, а не над стрелками. Видимо поэтому состояние действия *разрешается* обозначать в виде фигуры состояния деятельности, внутрь которой вписано действие, а не деятельность.

*Состояние деятельности*,[108] напротив, является существенным элементом диаграммы деятельности и имеет специальное обозначение — прямоугольник со скругленными сторонами. Это обозначение очень похоже на обозначение простого

---

[107] Что нам кажется очень полезным в данном случае. Данные понятия систематически путают все, кроме авторов языка, включая переводчиков книг и разработчиков инструментов.

[108] Мы иногда для краткости речи вместо словосочетания "состояние деятельности" употреблять просто слово "деятельность", если из контекста ясно, что подразумевается именно состояние деятельности.

состояния (прямоугольник со скругленными углами), но в тоже время отлично от него.[109] Тем самым подчеркивается родственная связь диаграмм состояний и действий. Внутри фигуры состояния деятельности пишется так называемое *выражение деятельности*. Никакого специального синтаксиса для данного выражения UML не определяет: это может быть текст на псевдокоде, фрагмент кода на языке программирования и просто название деятельности на естественном языке. Никакой видимой внутренней структуры состояние деятельности не имеет, в противоположность простому состоянию, которое может иметь действия на входе, выходе, внутренние и отложенные переходы. Собственно говоря, то, что написано внутри фигуры состояния деятельности — это и есть его внутренняя активность. Но это отнюдь не означает, что состояние деятельности атомарно и не имеет внутренней структуры. Напротив, оно по определению *не* атомарно и имеет внутреннюю структуру. Предполагается, что либо эта структура не важна на выбранном уровне абстракции, либо она раскрывается в другом месте[110] с помощью машины состояний, другой диаграммы деятельности или иным способом.

На диаграмме деятельности применяется еще один тип сущностей — объекты — и целый ряд фигур, похожих на сущности, но таковыми не являющимися. Мы обсудим их в подходящем контексте.

### *4.3.2. Переходы по завершении и ветвления*

На диаграмме деятельности применяется один основной тип отношений — простые переходы по завершении (а также поток объектов, см. разд. 4.3.4). Переход по завершении не имеет возбуждающего события — событием является завершение внутренней активности (деятельности) в состоянии. Как правило, исходящий переход по завершении один; если их несколько, они должны быть снабжены сторожевыми условиями, образующими полную дизъюнктную систему предикатов (см. разд. 4.2.2). Кроме того, можно использовать и переходы, возбуждаемые событиями. Срабатывание такого перехода означает прерывание выполнения деятельности в состоянии и переход в другое состояние.

**ЗАМЕЧАНИЕ**

> Граница между диаграммой состояний и диаграммой деятельности весьма условна и зыбка. Мы определим ее так: диаграмма деятельности – это такая диаграмма состояний, в которой все или почти все состояния являются состояниями действия или деятельности, и все или почти все переходы являются переходами по завершении. Осталось только договориться, что значит "почти все".

Таким образом, в рамках объектно-ориентированной семантики машины состояний авторы UML сумели описать семантику обычных блок-схем (в которых нет

---

[109] При достаточно мелком масштабе отличие заметно разве что под микроскопом. Честно говоря, мы затрудняемся дать однозначную оценку тому факту, что обозначения путаются. С одной стороны, это нехорошо, а с другой стороны, как еще побудить людей, привыкших думать в терминах блок-схем, побудить начать думать в терминах переходов-состояний? Только подав новое средство в привычной упаковке. Но в UML 2.0 от этих проблем решили просто отказаться — все рисуется одинаково.

[110] Почти все инструменты поддерживают эффективную навигацию в модели, позволяющую раскрыть структуру состояния деятельности, если она определена.

никаких событий, а есть последовательная передача управления следующей деятельности по завершении предыдущей деятельности).

Оставим пока в стороне особые добавления к диаграммам деятельности, имеющиеся в UML, и рассмотрим пример самой обычной блок-схемы (в нотации UML), описывающую последовательность действий при выполнении некоторого сценария. В нашей информационной системе отдела кадров предусмотрен вариант использования `DeleteDeparment` для удаления подразделения. Удаление подразделения не такая простая операция — нельзя просто взять и удалить подразделение — оно может быть связано отношением композиции с другими подразделениями и должностями, а те, в свою очередь, связаны с другими сущностями и т. д. Мы должны решить, какие элементарные операции и в какой последовательности нужно выполнить, чтобы достичь требуемого не нарушив целостность наших данных. Допустим, что в нашем распоряжении имеются классы и операции, представленные на рис. 4.32 (детали, ненужные в данном примере, опущены).

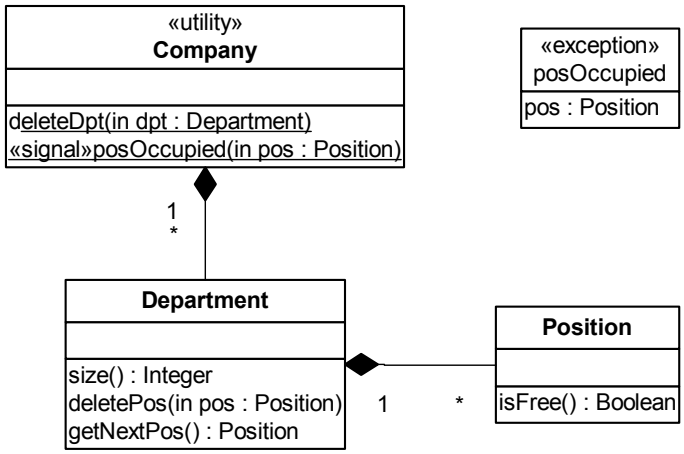



**Рис. 4.32. Некоторые классы в информационной системе отдела кадров**

Операция `size` класса `Department` доставляет количество должностей в подразделении, `deletePos` — удаляет указанную должность, `getNextPos` — итератор, позволяющий перебирать должности в подразделении. Функция `isFree` класса `Position` проверяет вакантность должности. Операция обработки исключения `posOccupied` и операция `deleteDpt` определены в службе общего пользования `Company`. Разумеется, нам должно быть известно, какое именно подразделение следует удалить — пусть этот объект называется `dpt`. В этой ситуации можно предложить различные варианты поведения — все зависит от кадровой политики организации, которую нужно реализовать. Допустим, что в нашей информационной системе отдела кадров действует такое правило: если в подразделении есть "живые люди", то его нельзя просто так удалить — это ошибка, а если нет никого или только "мертвые души", то можно удалять все не задумываясь. Тогда алгоритм удаления подразделения на псевдокоде[111] можно описать, например, следующим образом.

[111] См. сноску 3 в разд. 5.1.1.

```
if dpt.size() > 0
  repeat
    pos := dpt.getNextPos()
    if not pos.isFree() then
      send posOccupied(pos)
      stop
    dpt.deletePos(pos)
  until dpt.size() = 0
deleteDpt(dpt)
```

Это же поведение в нотации диаграммы деятельности UML представлено на рис. 4.33.

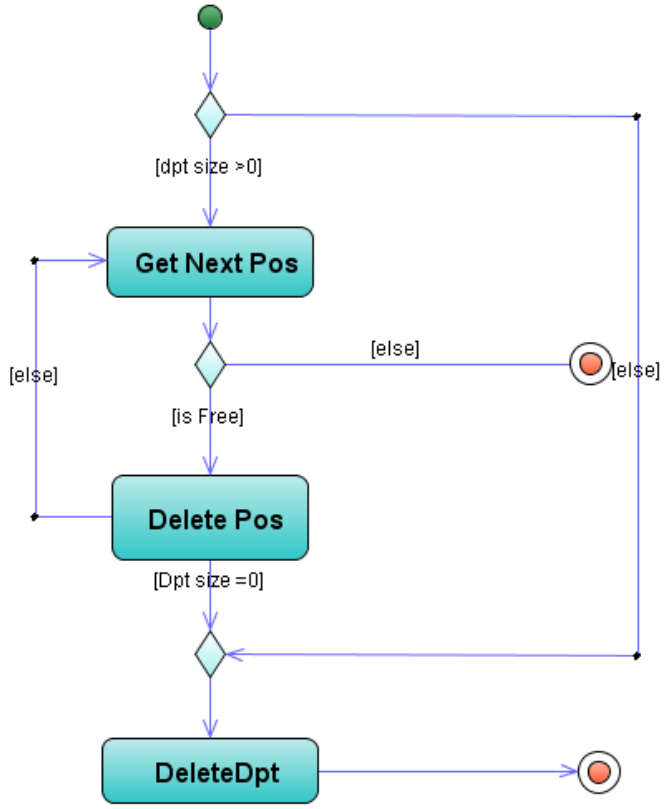



**Рис. 4.33. Диаграмма деятельности по удалению подразделения в информационной системе отдела кадров**

Здесь мы считаем нужным отступить от основного русла разбора примера и сделать существенное замечание относительно нотации. Ромбик, который используется на диаграмме деятельности, хотя и является значком, но не является сущностью. Это не более чем связующий символ, позволяющий более экономно изображать сегментированные переходы (см. разд. 4.2.2, рис. 4.10). В примере на рис. 4.30 два ромбика в верхней части диаграммы использованы, чтобы показать

место *ветвления* потока управления на альтернативные[112] потоки управления. Но сегментировать в виде дерева можно не только множество исходящих переходов данного состояния, но и множество входящих переходов. В таком случае можно также использовать ромбик, чтобы показать место *слияния* альтернативных потоков управления (нижний ромбик на рис. 4.30). При ветвлении ромбик соединяет один входящий и несколько исходящих сегментов перехода, а при слиянии ромбик соединяет несколько входящих и один исходящий сегмент.
На этом можно было бы успокоиться, поскольку в обычных блок-схемах больше ничего и не бывает, а можно рассмотреть рис. 4.34 и задаться вопросом, что может означать фрагмент диаграммы на рис. 4.34 справа.

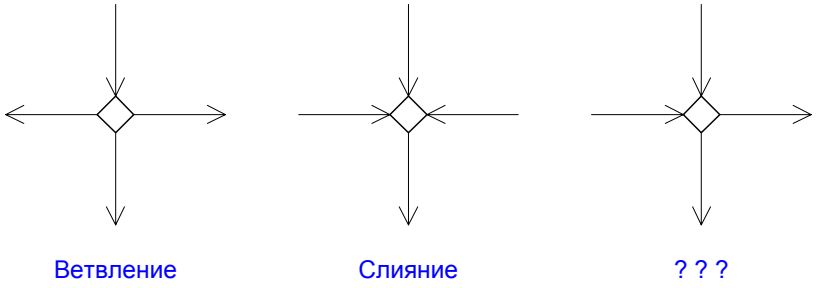



**Рис. 4.34. Варианты использования ромбика на диаграммах деятельности**
Стандарт и авторы языка на этот счет хранят молчание и в своих примерах таких конструкций не используют. Авторы же популярных руководств по UML и разработчики инструментов либо также обходят этот вопрос молчанием, либо считают данную конструкцию синтаксической ошибкой. Мы же полагаем, что такую конструкцию вполне осмысленной: на рис. 4.35 смысл фрагмента диаграммы слева раскрыт на фрагменте диаграммы справа. Другими словами, мы считаем разумным и допустимым произвольный ациклический ориентированный граф из переходных состояний, обозначенных ромбиками, и сегментов перехода, считая семантикой такого графа сегментированных переходов множество переходов, прочитанных, как все возможные пути в этом графе.

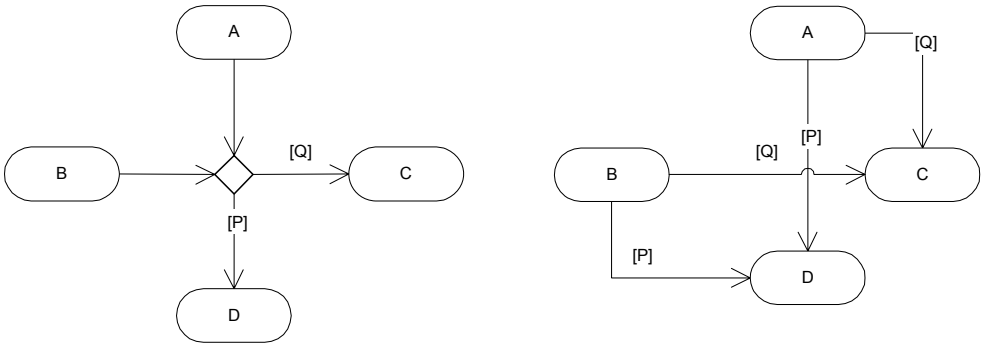



**Рис. 4.35. Семантически эквивалентные фрагменты диаграммы деятельности**
Возвращаясь к нашему примеру, мы считаем синтаксически возможным описать деятельность по удалению подразделения диаграммой на рис. 4.36.

[112] Не следует путать альтернативные потоки управления с параллельными.

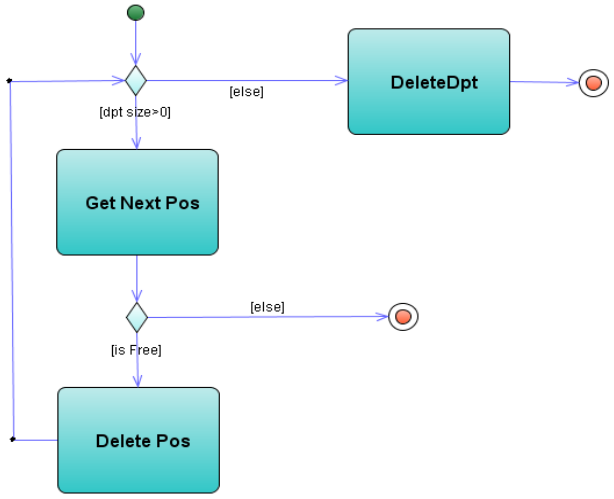



**Рис. 4.36.** Диаграмма деятельности по удалению подразделения

По нашему мнению, диаграмма на рис. 4.36 соответствует следующему псевдокоду:

```
while dpt.size() > 0
  pos := dpt.getNextPos()
  if not pos.isFree() then
    send posOccupied(pos)
    stop
  dpt.deletePos(pos)
deleteDpt(dpt)
```

Вообще говоря, можно поспорить, что более наглядно: диаграммы деятельности на рис. 4.33 и рис. 4.36 или приведенные программы на псевдокоде. Если положить перед человеком два листа: с диаграммой рис. 4.33 (или 4.36) и с текстом на псевдокоде, и предложить выбрать, то реакция, как показывают наблюдения автора, далеко не однозначна и сильно зависит от личностных качеств испытуемого. Чтобы читатель мог оценить, какой стиль ему предпочтителен, мы приведем еще несколько вариантов графической записи данного элементарного алгоритма, уже не в стиле структурной блок-схемы, как на рис. 4.36, а стиле машины состояний UML. Начнем с очевидного: традиционные ромбики не нужны, они только занимают место на диаграмме (рис. 4.37).

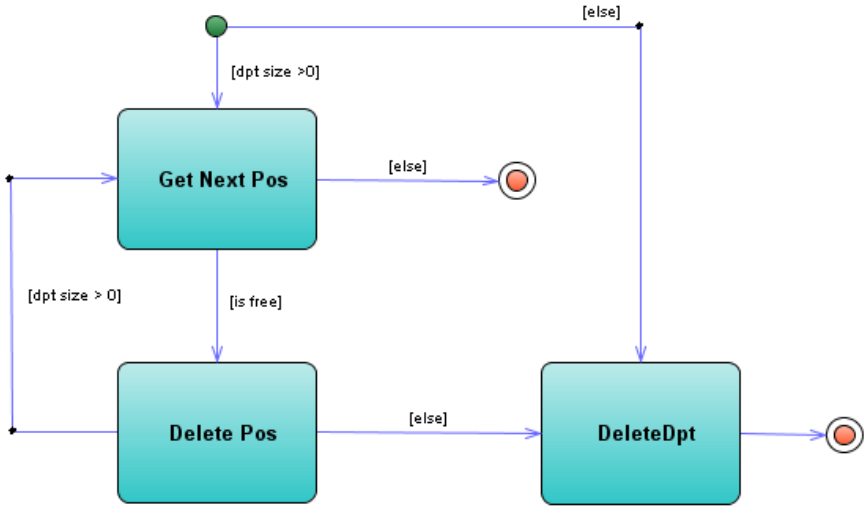



**Рис. 4.37. Диаграмма деятельности без использования переходных состояний**

Если вам еще не надоел данный пример, попробуйте прочитать диаграмму на рис. 4.38. Здесь мы попытались отразить основную структуру разбираемого алгоритма: если подразделение пусто, то его можно удалять без всяких сомнений, а если нет, то нужно анализировать состояние должностей. Этот анализ выполняется в простом состоянии `in Doubt`. В данном состоянии имеется внутренняя активность `do`, состоящая в "бесконечном" циклическом переборе должностей в подразделении. Но никакой бесконечности, разумеется, нет: на каждом шаге цикла выполняется внутренний переход, в результате которого либо мы натыкаемся на занятую должность и работа машины состояний грубо обрывается действием `terminate`, либо уменьшаем размер перебираемого множества, так что процесс неизбежно закончится.

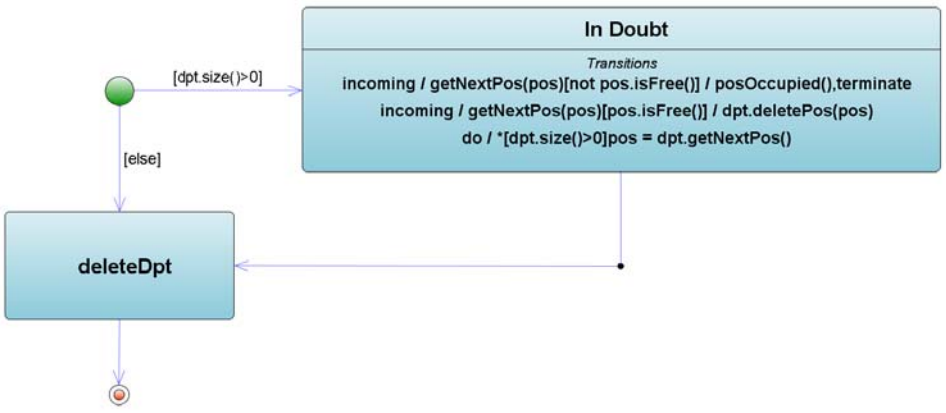



**Рис. 4.38. Использование простого состояния с внутренней активностью и переходами**

Состояние `deleteDpt` является, на самом деле, состоянием действия, а не состоянием деятельности и без него, как мы отмечали, можно обойтись. Вглядимся теперь внимательно в простое состояние `in Doubt`. В сущности, там в цикле выполняется пара действий: взять следующую должность, обработать ее. Таким

образом, имеются две точки, два мгновения в вычислительном процессе: когда мы от взятия переходим к обработке и когда от обработки переходим к взятию. Мы можем считать эти мгновения состояниями (но это не состояния объекта или системы, это состояния нашего вычислительного процесса!). Сами по себе эти состояния не важны — алгоритм определяется тем, в каком порядке посещаются эти состояния и какие содержательные действия выполняются на переходах. В результате мы получаем конечный автомат, приведенный на рис. 4.39. Все действия выполняются на переходах, а простые состояния служат только для определения смежности переходов. Их можно было бы никак не назвать, но синтаксис UML требует наличия имен простых состояний, поэтому мы их просто перенумеровали.

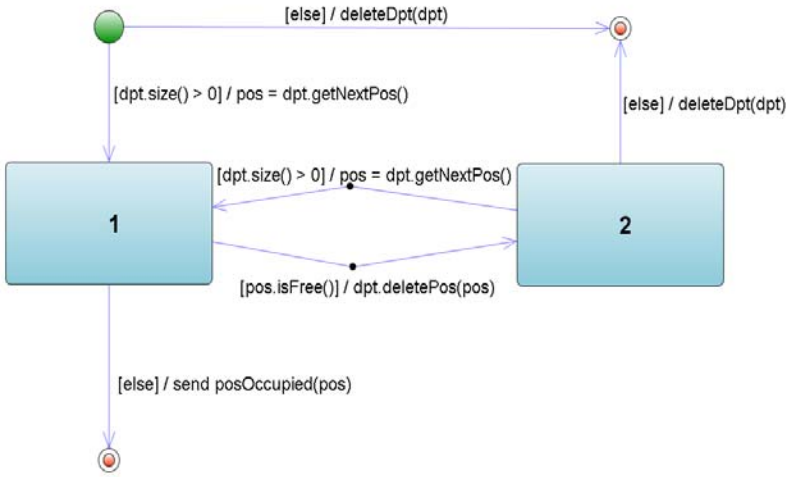



**Рис. 4.39. Эквивалентный конечный автомат**

Мы надеемся, что повторный просмотр рис. 4.33–4.39 даст читателю достаточно пищи для размышлений и выработки собственного[113] отношения к машинам состояний и диаграммам деятельности в UML.

## *4.3.3. Дорожки*

В UML имеется своеобразное графическое средство, которое называется дорожкой.

*Дорожка* — это графический комментарий, позволяющий классифицировать по некоторому признаку сущности на диаграмме деятельности.

Дорожка не является элементом модели — в метамодели UML нет такого понятия.[114] Это именно графический комментарий, подобный границам системы на диаграмме использования (разд. 2.2.2). Поэтому никаких формальных правил применения дорожек нет.

---

[113] Будь на то наша воля, мы бы предпочли видеть спецификацию поведения операции удаления подразделения `dpt` примерно такой: $\forall pos \in dpt(isFree(pos)) \rightarrow deleteDpt(dpt)$.

[114] В последних версиях стандаратных метамоделей появилось.

Дорожки (или подобные им конструкции) часто применяются при моделировании бизнес-процессов в организациях, откуда они и были заимствованы в UML. Рассмотрим бизнес-процесс приема сотрудника на работу в нашей информационной системе отдела кадров (см. разд. 3.3.2–3.3.3, рис. 3.11–3.14). Обсуждая сценарий этого варианта использования, в главе 3 мы сосредоточили свое внимание не на самом бизнес-процессе приема, а на участии моделируемой системы в этом процессе. Давайте допустим, что нас интересует прохождение всего процесса в целом, включая как те шаги, которые выполняются системой, так и те шаги, которые (пока) предполагается выполнять вручную. Польза такого описания очевидна: если мы будем знать необходимые в бизнес-процессе шаги, в том числе выполняемые вручную, мы сможем составить более надежный план его поэтапной автоматизации. Даже в самом простом случае приема на работу этапу оформления документов предшествуют другие этапы: сбор информации о кандидате, проверка и обработка этой информации, принятие решения и наконец собственно прием или отказ в приеме. На рис. 4.40 представлена диаграмма деятельности, отражающая простейший бизнес-процесс найма на работу. Мы считаем, что наш процесс включает четыре деятельности:

- `Interview` — сбор информации;
- `Approve` — анализ собранной информации и принятие решения;
- `Fill Forms` — заполнение документов;
- `Refuse` — отказ в найме.

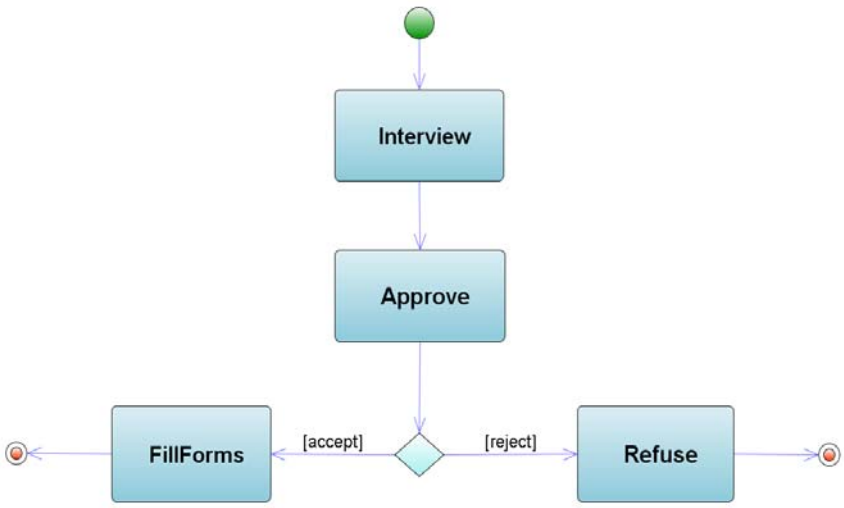



**Рис. 4.40. Диаграмма деятельности процесса найма на работу**

На диаграмме рис. 4.40 нет никаких дорожек — все деятельности равноправны и однородны. Допустим теперь, что деятельность, в которую нанимаемый вовлечен непосредственно (`Interview, Fill Forms, Refuse`) происходит в одном месте и, так сказать, у него на глазах, а важная деятельность (о которой нанимаемый может не догадываться) по анализу информации и принятию решения осуществляется в другом месте и, может быть, другими действующими лицами (техническими специалистами, руководителями подразделений и т. д.). Эта важная информация не является частью модели, т. к. не имеет отношения к поведению системы, но мы

можем отразить ее на диаграмме с помощью дорожек (рис. 4.42). В данном случае мы подразумеваем, что дорожка с названием `HRDpt` содержит деятельности, выполняемые в приемной отдела кадров, а дорожка с названием `SomeDpt` содержит деятельности, выполняемые в том подразделении, куда предполагается принять кандидата.

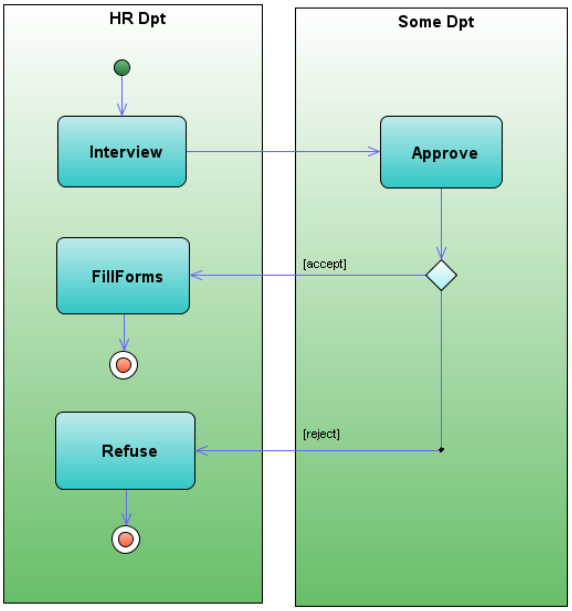



**Рис. 4.42. Дорожки**

Графически дорожки изображаются в виде прямоугольников с названиями. Как правило, их изображают со смыкающимися боковыми сторонами, хотя это и не обязательно. Авторы UML утверждают, что изображение, подобное приведенному на рис. 4.42, напоминает плавательные дорожки в бассейне, откуда и произошло название данной графической конструкции. Возложим ответственность за правомерность такой ассоциации на авторов UML и завершим этот несложный раздел.

### *4.3.4. Траектория объекта*

Диаграммы деятельности UML позволяют моделировать поведение не только определяя поток управления, как в приведенных выше примерах, но и (до некоторой степени) поток данных.

#### Поток управления и поток данных

В программе, написанной на обычном процедурном языке программирования, порядок выполнения операторов (шагов алгоритма) определяется порядком расположения операторов в тексте программы и структурами управления. Фактически, обычная программа задает поток управления (разд. 4.1.2). Но при выполнении шагов алгоритма выполняются преобразования данных: переменные приобретают и меняют своих значения. Последовательность изменения данных называется потоком данных. Поток управления (в детерминированной программе) определяет и поток данных. Действительно, если,

например, в программе есть несколько оператор присваивания для одной переменной, то история изменения значения переменной (т. е. поток данных) определяется тем, в каком порядке выполняются присваивания (т. е. потоком управления). Но это зависимость по данным между операторами непосредственно в тексте программы отражается далеко не так явно, как зависимость по управлению.[115] Довольно давно была предложен двойственный способ определения программы: явно указывается зависимость по данным, т. е. поток данных, а поток управления определяется потоком данных. Например, самый простой способ определить поток управления по потоку данных такой: для всех шагов алгоритма известно, какие данные являются входными (какие переменные используются), а какие данные являются выходными (какие переменные вычисляются). Поток управления определяется так: шаг алгоритма выполняется, как только все его входные данные определены (см. для сравнения врезку "Диаграммы потоков данных" в разд. 2.2.6).

При объектно-ориентированном подходе к моделированию поток данных — это изменение состояний объектов во времени, и описание такого изменения существенным образом характеризует поведение. Для описания данной характеристики поведения в UML используются понятия траектория объекта и объект в состоянии.

*Объект в состоянии* — это объект некоторого класса, про который известно, что он находится в определенном состоянии в данной точке вычислительного процесса.

Синтаксически объект в состоянии изображается, как обычно, в виде прямоугольника и его имя подчеркивается, но дополнительно после имени объекта в квадратных скобках пишется имя состояния, в котором в данной точке вычислительного процесса находится объект. В некоторых случаях состояние объекта не важно, например, если достаточно указать, что в данной точке вычислительного процесса создается новый объект данного класса, и в этом случае применяется обычная нотация для изображения объектов. Важно подчеркнуть, что объект в состоянии на диаграммах деятельности "по определению" считается состоянием, т. е. вершиной графа модели, которая может быть инцидентна переходам, правда переходам особого рода

*Траектория объекта*[116] — это переход особого рода, исходным и/или целевым состоянием которого является объект в состоянии.

---

[115] Это обстоятельство является источником болезненных ошибок в программах. Например, типичная ошибка: использование неинициализированной переменной, т. е. использование в вычислениях значения переменной до того, как это значение было определено.

[116] Некоторые авторы предпочитают использовать слово "поток", поскольку словосочетания "поток данных" и "поток управления" являются давно и хорошо устоявшимися терминами. Мы все-таки остановились на термине "траектория объекта", поскольку "поток объекта" по-русски звучит уж очень нескладно, хотя и соответствует устоявшейся традиции.

Траектория объекта изображается в виде пунктирной стрелки (в отличии от сплошной стрелки обычного перехода). Семантически траектория объекта, проведенная от состояния деятельности к объекту в состоянии означает, что результатом деятельности является переход указанного объекта в данное состояние (или может быть, создание нового объекта в указанном состоянии, что является частным случаем изменения состояния). Траектория объекта, проведенная от объекта в состоянии к состоянию деятельности к означает, что объект в данном состоянии является необходимым входным данным указанной деятельности.

Таким образом объекты в состоянии и траектории позволяют показать на диаграмме деятельности не только зависимость по управлению, но и зависимость по данным между состояниями деятельности на диаграмме. А именно, чтобы показать, что одна деятельность использует результаты другой деятельности, достаточно показать траекторию передаваемого объекта.

Рассмотрим это на примере диаграммы деятельности, описывающей процесс найма сотрудника в информационной системы отдела кадров. Рис. 4.43 в основном повторяет рис. 4.21 и 4.42, но здесь представлена траектория объекта класса `Person`, хранящего данные о принимаемом сотруднике. На диаграмме хорошо видно, что именно является входными и выходными данными каждого из состояний деятельности: в результате деятельности `Interview` создается новый объект, который далее обрабатывается, меняя свою состояние.

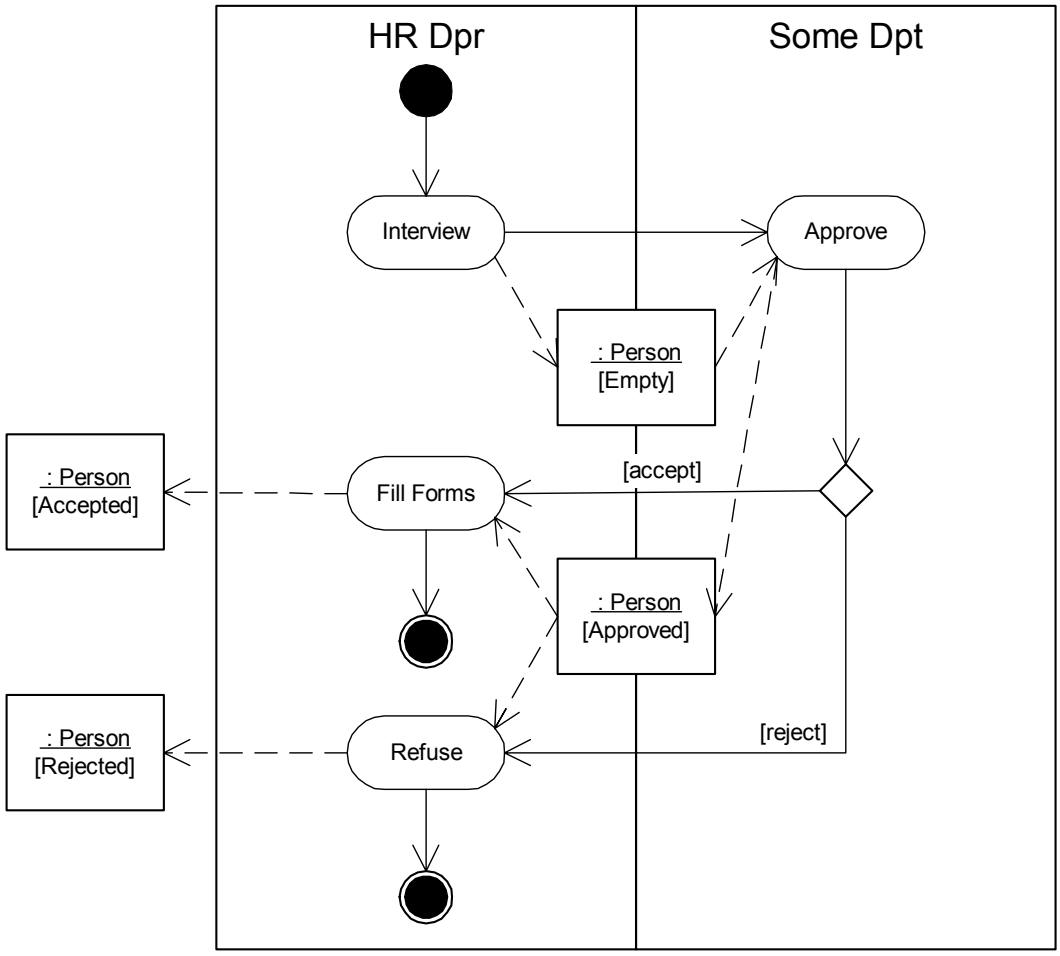



**Рис. 4.43. Траектория объекта**

Нетрудно заметить, что в данном случае мы фактически повторяемся, описывая поведение системы. Из диаграммы на рис. 4.43 следует, что деятельность `Approve` выполняется после деятельность `Interview` причем это указано дважды: один раз с помощью перехода по завершении из `Interview` в `Approve` и второй раз с помощью траектории объекта, показывающей, что для выполнения деятельности `Approve` необходим объект, создаваемый деятельностью `Interview`. Разумеется, UML позволяет не говорить лишнего: диаграмма на рис. 4.44 описывает то же самое поведение, что и диаграмма на рис. 4.43.

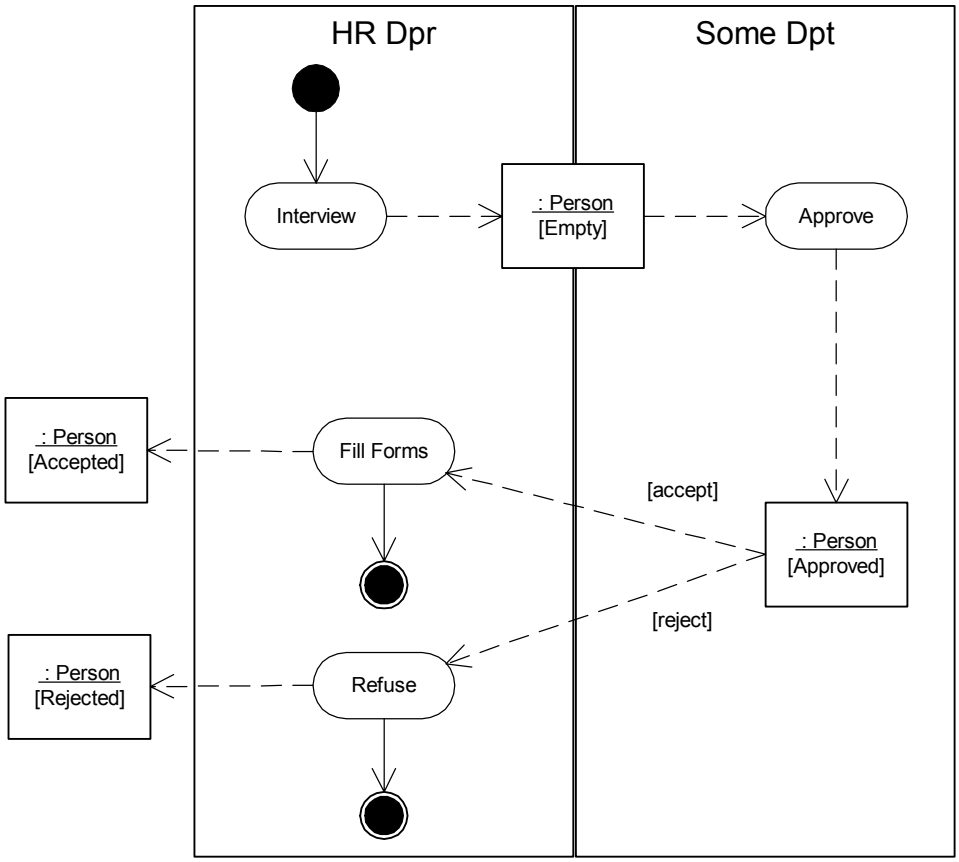



**Рис. 4.44. Использование траекторий объектов вместо переходов по завершении**

## *4.3.5. Отправка и прием сигналов*

В UML имеется довольно много обозначений, которые не несут самостоятельной семантической нагрузки, а являются "синтаксическим сахаром", применяемым для повешения наглядности, сокращения записи и просто для украшения диаграмм. Особенно много таких украшений предусмотрено для диаграмм деятельности, что не удивительно, учитывая их близкое родство с блок-схемами. Часть таких вспомогательных обозначений мы уже рассмотрели в предыдущих разделах, например обозначение для ветвления сегментированных переходов (ромбик, см. разд. 4.3.2), другие рассматриваются ниже, в более подходящем контексте (например, развилки и слияния, разд. 4.4.4). Здесь мы рассмотрим специальную нотацию, применяемую для обозначения действий по отправке и получению сигналов на переходах.

Вернемся еще раз к примеру с наймом на работу и допустим, что мы хотим отразить в модели несколько иной вариант поведения. В диаграммах на рис. 4.41–4.44 процесс происходящий в приемной отдела кадров приостанавливается на то время, пока не будут завершена деятельность по оценке кандидата и принятию решения, которая фактически происходит в другом месте. Такое вынужденное ожидание может быть психологически неприятно кандидату (равно как и менеджеру по персоналу).[117] Допустим, что в проектируемой информационной системы отдела кадров требуется обеспечить асинхронное проведение процесса приема: после сбора сведений о кандидате менеджер по персоналу отправляет сигнал в соответствующие инстанции и в ожидании ответного сигнала с решением переводит себя и кандидата в состояние ожидания с внутренней активностью — угощает чаем, рассказывает о миссии организации и т. п. В рамках уже рассмотренных обозначений такая ситуация может быть описана диаграммой деятельности (с использованием простого состояния), как показано на рис. 4.45. Здесь мы предполагаем, что в не отображаемых на диаграмме "инстанциях" принимается сигнал `Request` с аргументом `person` и в ответ отправляется сигнал `Answer` с аргументом `decision`.


**Рис. 4.45. Асинхронный процесс принятия решения при найме**

[117] Распространенный выход из этой ситуации — "зайдите за ответом завтра после обеда" — не соответствует современному стилю общения.

Приведенная на рис. 4.45 диаграмма точно описывает желаемое поведение, но может показаться не слишком наглядной: читатель должен знать синтаксические детали обозначений UML, чтобы понять описание процесса с первого взгляда. Между тем имеется хорошо знакомая очень многим наглядная система обозначений для передачи и приема сигналов (см. рис. 4.1). Эта система обозначений также включена в UML. Суть состоит в том, что действие по отправке и приему сигнала изображаются в виде фигур, сегментирующих соответствующие переходы. Применение данных обозначений приводит нас к диаграмме на рис. 4.46.


**Рис. 4.46. Специальные обозначения для отправки и приема сигналов**

### *4.3.6. Применение диаграмм деятельности*

Подводя итоги описания диаграмм деятельности в UML, мы хотим вернуться к обсуждению области применения этих диаграмм.

С одной стороны, диаграммы деятельности — общее и мощное средство описания алгоритмов. Причем не обязательно программно реализованных алгоритмов: диаграммы деятельности с успехом можно применять для моделирования поведения людей, устройств и организаций при выполнении бизнес-процессов. Обратите внимание, что в наших примерах про процесс приема на работу (см. рис. 4.41–4.46) никакой программной реализации, вообще говоря, не подразумевается. Скорее всего, деятельности, обозначенные на этих диаграммах, должны иметь некоторую программную поддержку, но они не могут выполняться

полностью автоматически — в них вовлечены люди. Если мы рассматриваем диаграммы деятельности как средство описания бизнесс-процессов, то их естественное место в рамках UML — послужить первым шагом при реализации вариантов использования.

С другой стороны, диаграммы деятельности, равно как и блок-схемы, можно применять практически как средство визуального структурного программирования. Посмотрите еще раз рис. 4.33, 4.36-4.37 и особенно рис. 4.39 — это готовый к выполнению код.[118] В качестве средства программирования диаграммы деятельности целесообразно применять для реализации операций (что, фактически, и сделано нами в упомянутых примерах — реализована операция удаления подразделения).

Таким образом, диаграммы деятельности применяются для описания поведения на самом высоком уровне абстракции, наиболее удаленном от программной реализации и на самом низком уровне, практически на уровне программного кода. Данное наблюдение дает нам повод дать следующую рекомендацию. Если при проектировании выявляется операция, выполнение которой непосредственно реализует вариант использования (как в случае операции удаления подразделения), то составление диаграммы деятельности является наилучшим способом моделирования поведения.

На рис. 4.47 приведен наш вариант метамоделей элементов диаграммы деятельности в UML. Поскольку диаграмма деятельности является частным случаем машины состояний, метаклассы, соответствующие элементам диаграммы деятельности определяются как подклассы метаклассов машины состояний.

---

[118] К сожалению, анализируя возможности инструментов, доступных автору в момент написания книги, приходится констатировать, что диаграммы деятельности для генерации кода не используются (или используются не в полной мере), хотя даже авторы языка в [1] указали на возможность и желательность такого применения.

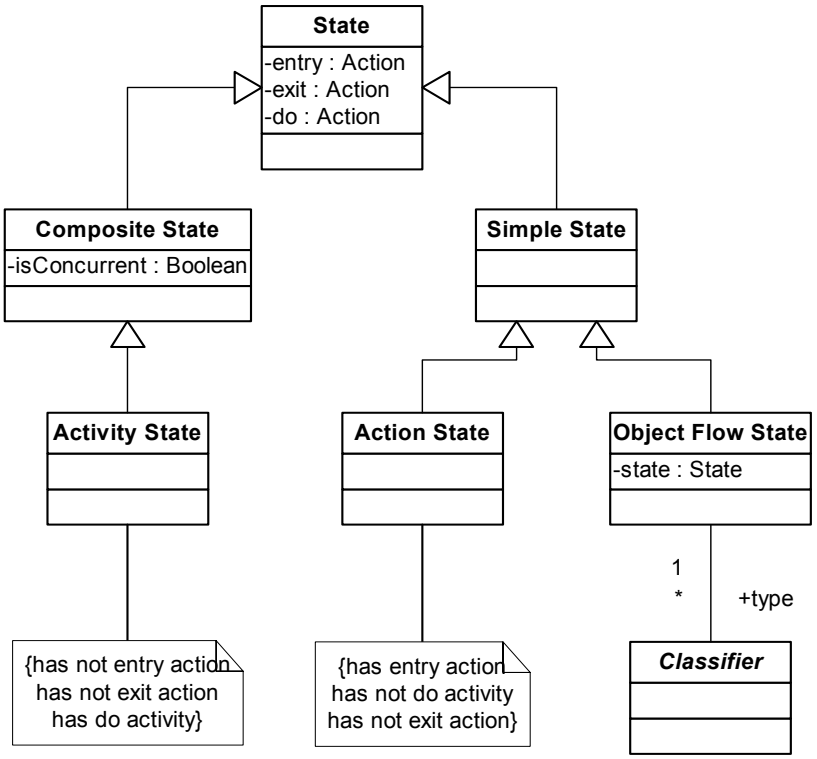



**Рис. 4.47. Метамодель элементов диаграммы деятельности**

## 4.4. Диаграммы взаимодействия

Диаграммы взаимодействия предназначены для моделирования поведения путем описания взаимодействия объектов для выполнения некоторой задачи или достижения определенной цели. Взаимодействие происходит путем обмена сообщениями. Диаграммы взаимодействия применяются на разных уровнях моделирования: как для описания поведения отдельных операций, так и целых вариантов использования. Данный тип диаграмм позволяет описывать не только взаимодействие программных объектов (экземпляров классов), но и взаимодействие экземпляров иных классификаторов: действующих лиц, вариантов использования, подсистем, компонентов, узлов. Диаграммы взаимодействия графически изображаются в двух формах: диаграммы последовательности и диаграммы кооперации.

Мы уже отмечали, что диаграммы кооперации и диаграммы последовательности семантически эквиваленты, хотя графически выглядят совсем по-разному. Семантически эти диаграммы эквиваленты потому, что описывают одно и то же: последовательность передачи сообщений между объектами в процессе взаимодействия объектов. А выглядят по-разному они потому, что в диаграмме последовательности графически подчеркивается упорядоченность во времени передаваемых сообщений, в то время как в диаграмме кооперации на передний графический план выдвигается структура связей между объектами, по которым передаются сообщения.

Сразу подчеркнем главное: оба типа диаграмм моделируют поведение "по индукции", от частного к общему, путем описания конкретного протокола передачи сообщений (т. е. сценария). В этом и сила и слабость данного способа

описания поведения. Сильная сторона состоит в том, что в объектно-ориентированной парадигме обмен сообщениями — это и есть само выполнение программы, поэтому протокол передачи сообщений является наиболее точной моделью поведения. Диаграммы взаимодействия находятся "ближе" к реальному выполнению программы, чем другие средства описания поведения. Слабость диаграмм взаимодействия состоит в том, что это диаграммы описывают поведение на уровне объектов, а не классов, на уровне протоколов выполнения алгоритма, а не самого алгоритма. Диаграммы взаимодействия менее "алгоритмичны", чем машины состояний и диаграммы деятельности.

На диаграммах обоих типов основными сущностями являются объекты: экземпляры классификаторов — классов и действующих лиц. Отношениями же являются связи, т. е. экземпляры ассоциаций, по которым передаются сообщения.

Наряду с основными сущностями и отношениями на диаграммах последовательности и кооперации применяется множество дополнительных элементов нотации, которые рассматриваются в разд. 4.4.2 и 4.4.3, соответственно. Но прежде чем переходить к особенностям нотации, необходимо рассмотреть основные элементы, используемые на этих диаграммах — сообщения.

### *4.4.1. Сообщения*

*Сообщение* — это передача управления и информации от одного объекта (*отправителя*) к другому (*получателю*). Отправка сообщения является действием (см. разд. 4.3.1), а получение сообщения — событием (см. разд. 4.2.4).

Не все действия связаны с передачей информации и отправкой сообщений. В UML таковыми считаются:

- вызов операции;
- создание объекта;
- уничтожение объекта;
- возврат значения;
- посылка сигнала.

Действие записывается в виде текста над (или рядом со) стрелкой, символизирующей сообщение. Если действие имеет параметры (вызов операции, создание объекта, посылка сигнала), то аргументы, соответствующие параметрам по числу и типам, записываются справа от имени действия в круглых скобках. Если действием является вызов операции, возвращающей значения, то слева от имени записывается список переменных для возвращаемых значений (их может быть несколько в случае широковещательного вызова) знак присваивания `:=`. Таком образом, та часть нотации сообщений, которая относится у выполняемому действию, имеет следующий синтаксис.

```
переменные := ИМЯ ( аргументы )
```

Сообщение имеет отправителя и получателя. Получателей может быть несколько, такое сообщение называется *широковещательным*. Все получатели широковещательного сообщения получают одно и то же сообщение.

Поскольку получение сообщения является событием, то получатель сообщения вместе с информацией получает и управление (для того, чтобы иметь возможность выполнить действия, инициируемые полученным сообщением). В UML

различается несколько типов тип передачи управления с помощью сообщения. Чтобы отличить тип передачи сообщения, в UML применяется специальная графическая нотация, а именно, различаются виды стрелок, которыми обозначаются сообщения. Хотя на диаграммах кооперации и последовательности сообщения обозначаются различным образом, принципы изображения одинаковы и перечислены в табл. 4.4.

**Таблица 4.4.** Типы передачи сообщений

| Вид стрелки | Тип передачи сообщения |
|---|---|
|  | *Вложенный поток управления*. Данный тип передачи сообщения подразумевает, что отправитель может отправить следующее сообщение только после того, как завершиться выполнение всех действий, инициированных данным сообщением. Обычно применяется при вызове операций. |
|  | *Простой поток управления*. Данный тип передачи подразумевает, что управление передается от отправителя сообщения получателю (возможно, безвозвратно). Обычно применяется при моделировании поведения на уровне действующих лиц и вариантов использования. |
|  | *Асинхронный поток уп*равления. Данный тип передачи подразумевает, что сообщение асинхронно передается от отправителя получателю, при этом у отправителя сохраняется свой поток управления, независящий от потока управления получателя. Обычно применяется при отправке сигналов. |
|  | *Возврат управления*. Данный тип передачи подразумевает возврат управления после выполнения всех действий, инициированных передачей сообщения с вложенным поток управления. При этом могут быть указаны возвращаемые значения. Данный тип передачи сообщения можно не отображать на диаграмме, поскольку он подразумевается по умолчанию при вызове операций. |
| не определяется | Допускается использование при моделировании других, не определяемых в UML, типов передачи управления, например, передача управления по истечении времени. |

Для того, чтобы сообщение могло быть передано от отправителя к получателю, отправитель должен "знать" получателя. Другими словами, должна существовать ассоциация между классами отправителя и получателя, экземпляр которой (связь) и служит тем путем, по которому передается сообщение. На диаграмме кооперации эта связь всегда изображается в явном виде, на диаграмме последовательности она подразумевается.

Однако поведение определяется не только и не столько тем, какие объекты посылают какие сообщения, но прежде всего тем, в каком порядке это происходит. UML позволяет определить относительный порядок сообщений во взаимодействии, причем это делается несколькими различными способами.

- На диаграмме последовательности порядок сообщений определяется временем их отправки, а время считается текущим на диаграмме сверху вниз. Таким

образом, сообщения, изображенные выше, предшествуют сообщениям, изображенным ниже.[119]

- Порядок можно задать с помощью последовательного *номера сообщения*, правила формирования которого описаны в разд. 4.4.3. Данные номера уникальны и обладают тем свойством, что сообщения с меньшими номерами предшествуют сообщениям с большими.
- Наконец, порядок можно указать, перечислив (через запятую) номера сообщений, *предшествующих* данному.

Чуть выше мы утверждали, что диаграммы *почти* не позволяют выйти за пределы описания протоколов выполнения алгоритмов (т. е. линейных программ). Здесь необходимо уточнить, что же все-таки *можно* сделать. UML позволяет задать *повторность* сообщения, т. е. либо задать сторожевое условие (см. разд. 4.2.2), либо повторитель (см. разд. 4.3.1). Семантика этих конструкций очевидна: сообщение посылается только при условии истинности сторожевого условия, а повторитель определяет, сколько раз нужно послать данное сообщение (возможно, разным объектам). Синтаксис такой же, как и для диаграмм состояний.

Таким образом, вообще говоря, сообщение может быть довольно сложной синтаксической конструкцией. Сразу отметим, что абсолютно все возможные части описания сообщения, как правило, нет нужды использовать — обязательным является только имя. Общий синтаксис текста описания сообщения следующий.

```
предшественники / повторность номер : переменные := ИМЯ ( аргументы )
```

На рис. 4.48 приведена метамодель сообщений.

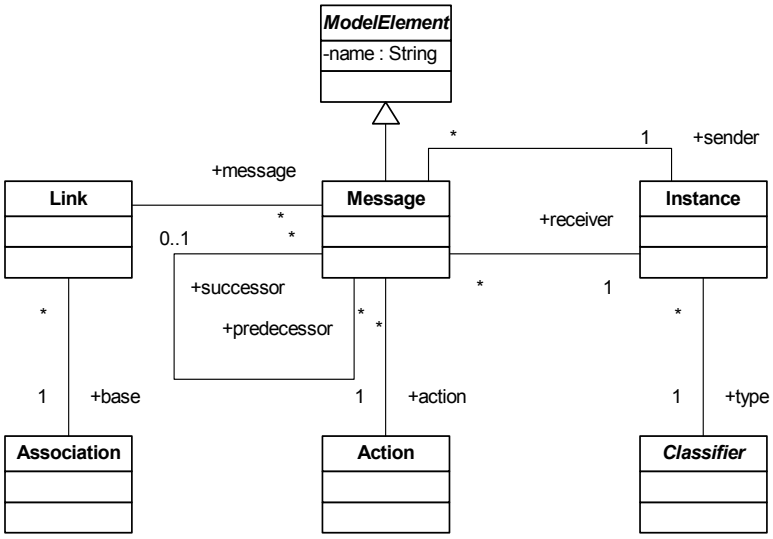



**Рис. 4.48. Метамодель сообщений**

[119] Это единственный случай в UML, когда взаимное положение элементов на диаграмме имеет значение. Во всех остальных случаях взаимное положение элементов не важно: все определяется только инцидентностью элементов.

### *4.4.2. Диаграммы последовательности*

*Диаграмма последовательности* предназначена для моделирования поведения в форме описания протокола сеанса обмена сообщениями между взаимодействующими объектами во время выполнения одного из возможных сценариев. Из этого определения вытекают несколько следствий, определяющих состав сущностей и набор отношений, используемых на диаграмме последовательности (равно как и на диаграмме кооперации).

- На диаграмме присутствуют только те объекты, которые задействованы в данном сеансе. Прочие объекты не показываются, хотя возможно и присутствуют в системе.
- Отображаются только те связи (экземпляры ассоциаций), которые нужны для передачи данной последовательности сообщений, прочие ассоциации не показываются.
- Состав сообщений (а тем самым операций и сигналов) определяется назначением данного взаимодействия; в других взаимодействиях эти же объекты могут обмениваться другими сообщениями.

Перейдем к описанию особенностей нотации диаграммы последовательности. Напомним, что в UML семантика первична, а нотация вторична: возможны различные вариации в способах рисования диаграмм, особенно это характерно для диаграмм последовательности. В наших примерах мы придерживаемся самого "скромного" стиля отображения диаграмм, с минимумом украшений. Однако все допустимые возможности

На диаграмме последовательности считается выделенным одно направление, соответствующее течению времени. По умолчанию считается, что время течет сверху вниз, но это не обязательно, например, можно считать, что время течет слева направо, оговорив это специальным примечанием. В наших примерах используется исключительно нотация по умолчанию: время всегда течет сверху вниз. Саму ось времени не отображают.

Сообщения изображаются прямыми стрелками разного вида (табл. 4.5). Если передача сообщения считается мгновенной (т. е. время передачи пренебрежимо мало), то стрелка горизонтальна (т. е. перпендикулярна оси времени). Если же нужно отобразить *задержанную доставку сообщения*, то стрелку немного наклоняют, так чтобы конец стрелки был ниже начала.

Среди сообщений есть первое, которое кладет начало данному взаимодействию. Стрелка этого сообщения расположена выше всех других стрелок сообщений. Все объекты, которые находятся выше первого сообщения, существуют *до* начала данного взаимодействия; все объекты, которые расположены ниже, возникают *в процессе* данного взаимодействия. Обычно объект возникает в результате выполнения конструктора класса данного объекта. Стрелку сообщения, соответствующую вызову операции конструктора, принято направлять к фигуре (прямоугольнику), обозначающей созданный объект.

**ЗАМЕЧАНИЕ**

> Иногда на диаграмме не отображают объект, отправляющий первое сообщение (например потому, что этот объект не участвует в дальнейшем взаимодействии). Стрелка первого сообщения приходит "ниоткуда". Вообще говоря, синтаксически это допустимо, но мы не рекомендуем использовать такой стиль, поскольку он

расходится с общепринятым способом отображения графов (вместе с отношениями всегда отображать инцидентные им сущности).

В направлении оси времени от всех участвующих во взаимодействии объектов отходит прямая пунктирная линия, которая называется *линией жизни*. Линия жизни представляет объект во взаимодействии: если стрелка отходит от линии жизни объекта, то это означает, что данный объект отправляет сообщение, а если стрелка сообщения входит в линию жизни, то это означает, что данный объект получает сообщение. Если же стрелка *пересекает* линию жизни объекта, то это ничего не значит — сообщение пролетело мимо.[120] Если в процессе взаимодействия объект заканчивает свое существование, то линия жизни обрывается и в этом месте ставится жирный косой крест.[121]

Над стрелкой сообщения указывается текстовая часть описания сообщения. Синтаксис этой части приведен в разд. 4.4.1. Заметим, что номер сообщения, равно как и номера предшествующих сообщений (см. разд. 4.4.1), на диаграммах последовательности обычно не указывают, поскольку в этом нет нужны: относительный порядок сообщений и так хорошо определяется осью времени.

Мы еще не закончили описание особенностей нотации диаграммы сообщений, но чувствуем, что пора привести пример. Рассмотрим взаимодействие, возникающее при одном из простых сценариев в нашей информационной системе отдела кадров, а именно, создание подразделения. Данное взаимодействие инициируется внешним действующим лицом — менеджером штатного расписания, который открывает соответствующую форму и запускает выполнение операции создания подразделения, после чего закрывает более не нужную ему форму (рис. 4.49).

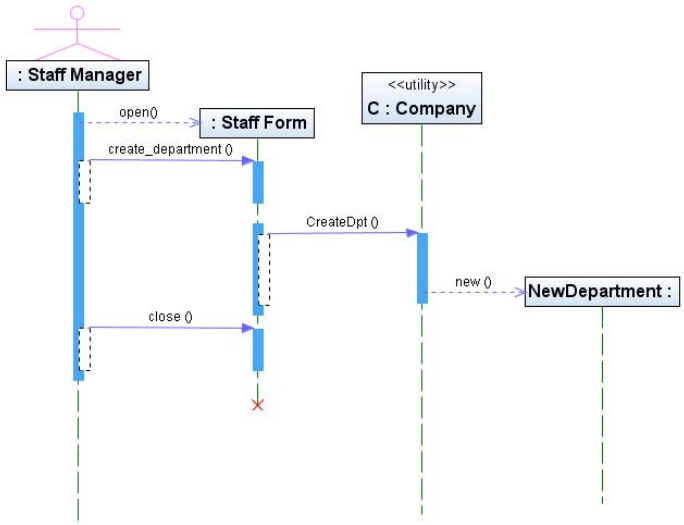



**Рис. 4.49. Диаграмма последовательности**

Продолжим описание нотации диаграмм последовательности.

---

[120] Обычно на диаграмме последовательности стараются располагать объекты таким образом, чтобы стрелки сообщений не пересекали линий жизни. Впрочем, это не всегда возможно.

[121] Некоторые авторы считают этот знак буквой "X". Посмотрите на рис. 4.49 и решите сами, кто прав.

Вообще говоря, подразумеваемая ось времени на диаграмме последовательности не является осью координат, на ней нет никакого масштаба и она задает только отношения "раньше–позже" для сообщений. Если же нужно в явном виде указать ограничения по времени, например, указать, что время задержки доставки сообщения должно быть ограничено сверху, то на диаграмму в нужном месте (имеет значение только положение по вертикали) рядом с началом или концом стрелки сообщения помещают произвольные идентификаторы, которые называются *метками времени*, и добавляют ограничение, задающее требуемое условие на значения меток времени.

| *Метка времени* — это именованная точка на оси времени. |
|---|

Допустим, что наша информационная система отдела кадров предназначена для организации, имеющей удаленный филиал. В этом филиале имеется и работает свой экземпляр информационной системы, который очевидно, должен быть проинформирован, что в головной организации создано новое подразделение. Возможно, эта информация дойдет с некоторой задержкой, поскольку для связи используется медленный канал. Такую ситуацию можно промоделировать диаграммой, приведенной на рис. 4.50.

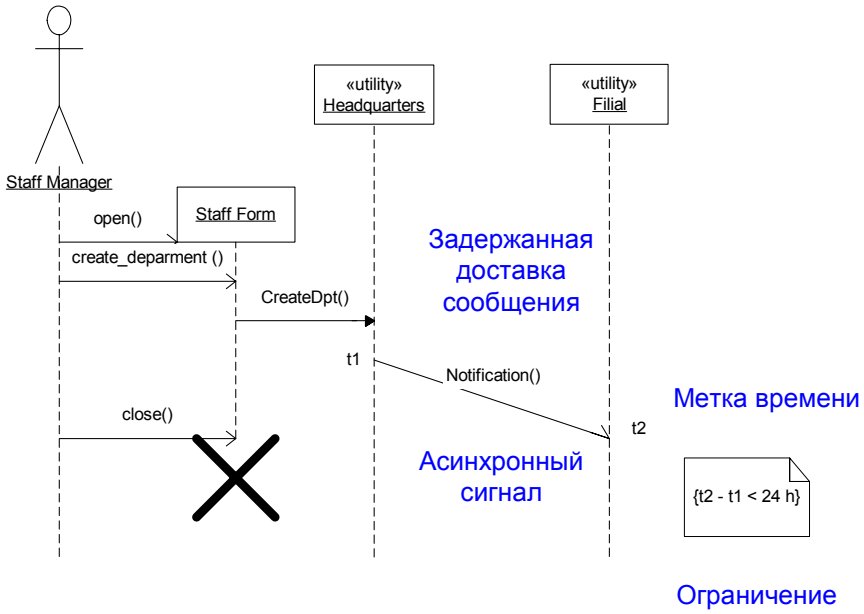



**Рис. 4.50. Метки времени и задержанная доставка сообщения**

Мы уже отмечали, что сообщение передает не только данные, но и поток управления. Чтобы показать, что некоторый объект в определенный период взаимодействия имеет *фокус управления*, или, как еще говорят, *активизирован*, на диаграмме последовательности соответствующую часть линии жизни объекта изображают в виде узкой полоски. Фактически такая полоска означает выполнение операции объекта и называется *активацией* объекта. Начало активации соответствует приему сообщения вызова операции, а конец активации — завершению выполнения операции и возврату управления. При этом, если во время выполнения данной операции будет вызвана еще раз вызвана операция этого же

объекта (та же самая операция, или другая), то это отмечается с помощью еще одной полоски активации, которая накладывается сбоку на первую. И так далее, глубина стека вызовов и, соответственно, количество наложенных полосок активации для одного объекта в UML, естественно, не ограничиваются.

### Стек вызовов процедур

> За время развития языков программирования были придуманы и реализованы самые различные механизмы обращения к подпрограммам, некоторые из них довольно причудливы. Один из них получил наибольшее распространение в традиционных языках программирования. При этом способе административная система времени выполнения поддерживает так называемый *стек вызовов*. При вызове подпрограммы в этот стек помещается новый элемент (его часто называют фреймом), в котором хранятся переданные аргументы, локальные переменные подпрограммы и адрес возврата (т. е. информация о том, куда нужно вернуть управление после завершения выполнения подпрограммы). Если в процессе выполнения вызванной подпрограммы происходит еще один вызов, то на стек вызовов помещается новый фрейм и т. д. При завершении подпрограммы верхний элемент стека вызовов снимается со стека и управление возвращается по сохраненному адресу возврата. Данная структура данных является именно стеком, поскольку фреймы всегда помещаются на стек и снимаются со стека в соответствии с дисциплиной LIFO (Last In First Out: первым пришел — последним ушел). На вершине стека вызовов всегда находится фрейм подпрограммы, выполняющейся в данный момент. Это соответствует следующей семантике вызова: при вызове выполнение вызывающей программы приостанавливается, управление (и аргументы) передается вызываемой подпрограмме, а когда выполнение последней закончится, возобновляется выполнение вызывающей программы. Другими словами, выполнение вызывающей программы не может возобновиться ранее, чем закончится выполнение вызванной. Данный механизм является простым, эффективным и удобным, что объясняет его широкое распространение и использование в большинстве случаев по умолчанию.

Для наглядности на диаграмме последовательности можно показать в явном виде возврат управления (и, может быть, возвращаемое значение), хотя это не обязательно: возврат управления подразумевается при использовании сообщения типа вызов операции. Более того, если использовать полоски для явного указания активации объектов, стрелки возврата не нужны: их легко можно мысленно восстановить.

Рассмотрим применение этой группы обозначений на следующем примере из информационной системы отдела кадров. Допустим, при создании нового подразделения немедленно выполняется операция `createPos` по созданию новой должности в этом подразделении (для начальника — свято место пусто не бывает), а после успешного создания подразделения форма демонстрирует менеджеру штатного расписания измененную организационную диаграмму компании (рис. 4.51). Здесь мы используем как активации, так и возвраты, чтобы показать применение всех средств, хотя, может быть, это немного перегружает диаграмму.

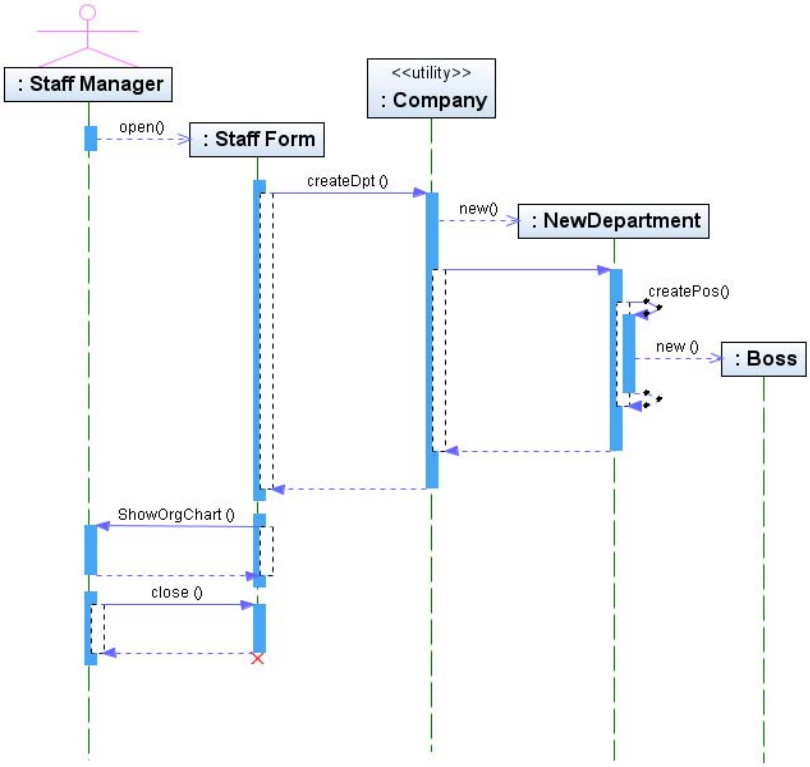



**Рис. 4.51. Активации и возвраты**

Следующее средство, которое нам необходимо обсудить — это ветвления. Как сказано в разд. 4.4.1, сообщение может иметь сторожевое условие. Если сторожевое условие ложно, то сообщение просто не будет отправлено. Соответственно, далее процесс взаимодействия пойдет иным путем. Таким образом, сторожевые условия на сообщениях позволяют отобразить на одной диаграмме взаимодействия *несколько* альтернативных сценариев. Но UML позволяет больше: можно из одной точки активации отправить несколько различных сообщений (в том числе адресованным разным объектам), снабженных альтернативными сторожевыми условиями. Не более чем одно из этих условий должно оказаться истинным — в противном случае это будет означать одновременный вызов двух операций в одном потоке управления, что вряд ли может иметь смысл и считается нарушением непротиворечивости модели.

Что будет, если два альтернативных сообщения (вызовы разных операций) отправляются из данной точки активации одному и тому же объекту? Вызываемый объект в любом случае будет активизирован (мы предполагаем, что ровно одно из сторожевых условий истинно). Однако, это *разные* активации. Чтобы отразить это обстоятельство на диаграмме, в UML введено еще одно обозначение: *альтернативные линии жизни* для одного объекта. Альтернативная линия жизни изображается как пунктирная линия, которая ответвляется от основной линии жизни объекта, а ниже сливается в ней. Как основная, так и альтернативные линии жизни могут содержать активации.

Рассмотрим пример (он получился немного искусственным и был создан вопреки возможностям используемого инструмента — все-таки диаграммы состояний

больше приспособлены для показа протоколов чисто линейных программ). Допустим, мы хотим отобразить два сценария создания подразделения: "под начальника" и исходя из потребностей организации. В обоих случаях при создании подразделения создается должность начальника, но в первом случае она немедленно заполняется имеющимся "нужным" человеком (операция `occupy`), а во втором объявляется вакантной (операция `setVacant`). Чтобы не перегружать диаграмму, мы включили описание только части взаимодействия, а именно, часть, соответствующую выполнению операции `createDpt` класса `Company` (рис. 4.52).

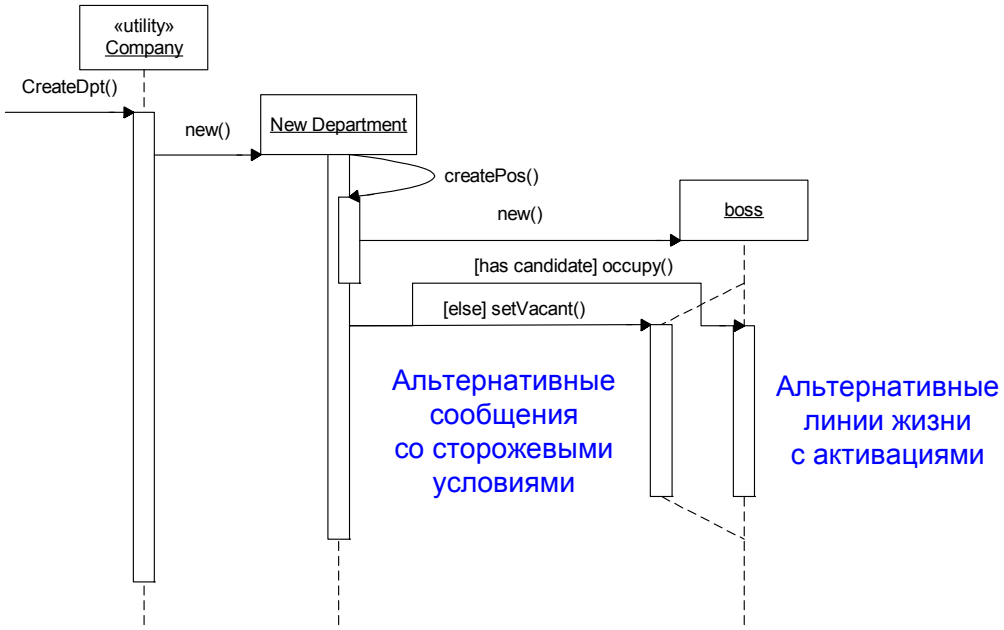



**Рис. 4.52. Сообщения со сторожевыми условиями и альтернативные линии жизни**

Последний, сравнительно редко используемый элемент нотации диаграмм последовательности, который мы хотим рассмотреть — это отображения на диаграмме последовательностей состояний пассивных объектов. Данный прием имеет много общего с траекторией объекта на диаграмме деятельности — визуализация смены состояний объекта по ходу вычислений, поэтому мы покажем его на том же примере: прием сотрудника на работу (см. рис. 4.42–4.46). Допустим для простоты, что менеджер персонала выполняет свою часть работы с помощью информационной системы отдела кадров, а собственно принятие решения о найме выполняется "вручную" где-то вне отдела кадров и менеджер персонала становится известным данное решение помимо нашей системы. Тогда деятельность менеджера сводится к тому, чтобы провести интервью и полученную информацию сохранить во вновь созданном объекте класса `Person`, а получив решение, либо выполнить дальнейшие действия (заполнить документы и т. д., подробности мы опускаем), либо сообщить кандидату об отказе. В первом случае созданный объект переходит в состояние `Accepted`, а во втором — `Rejected`. Сам созданный объект участвует в

этом взаимодействии пассивным образом: он хранит данные и меняет свое состояние по командам извне. Сообщений данный объект никаких не посылает. В таком случае можно на диаграмме последовательности на линии жизни данного объекта вместо активации показать состояния, в которые он переходит получая сообщения, как показано на рис. 4.53.

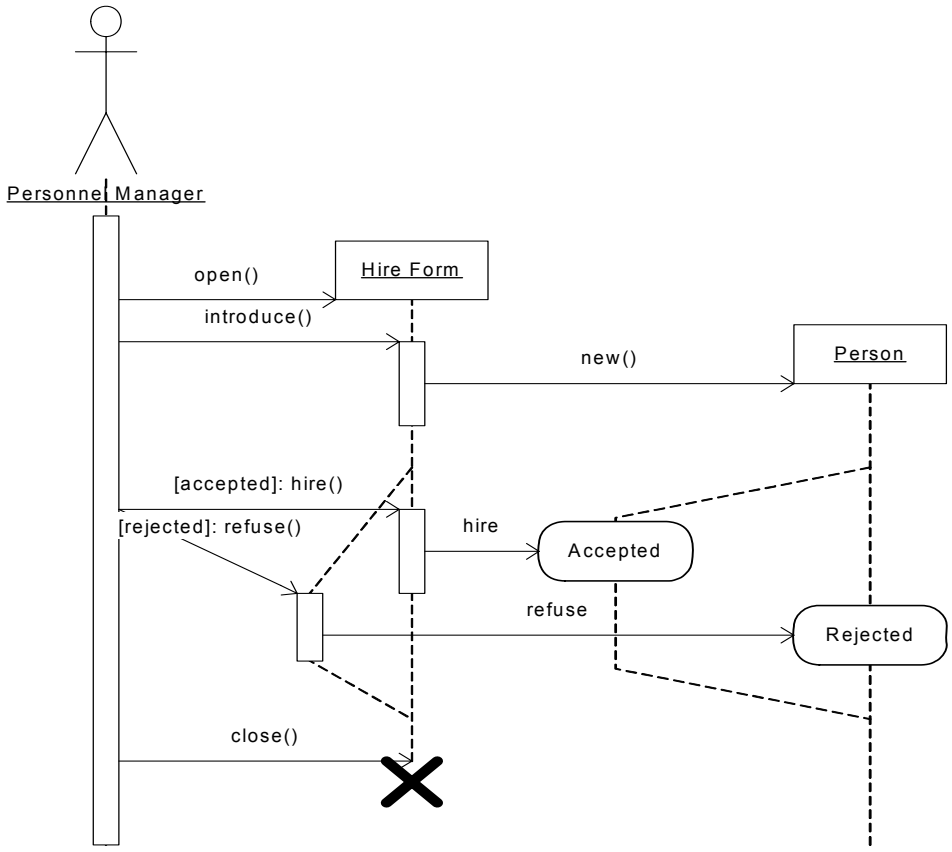



**Рис. 4.53. Состояния объекта на диаграмме последовательности**

Диаграмма на рис. 4.52–4.53 выглядят не очень убедительно. Действительно, сообщения передаются мгновенно, и это естественно изображать прямыми горизонтальными линиями, поскольку время течет на диаграмм последовательности сверху вниз. Но тогда линии альтернативных сообщений должны быть нарисованы горизонтально и проходить через одну точку. Как заметил еще Евклид, в обычной геометрии на плоскости это невозможно. Нам пришлось прибегнуть к ухищрениям: на рис. 4.52 линия сообщения не прямая, а на рис. 4.53 не горизонтальная.

К счастью эта же проблема встала перед проетировщиками программного обесечения очень давно и тогда же была решена. Дело в том, что диаграммы последовательности UML по существу заимствованы из другого графического языка описания поведения — MSC (Message Sequence Chart), который был разработан и успешно применяется производителями встроенных систем. Часть конструкций MSC была заимствована в UML 1, а оставшиеся в UML 2. В том числе были заимствованы *составные шаги взаимодействия*. Составные шаги позволяют на диаграмме последовательности, которая фактически является диаграммой

протокола взаимодействия, отражать и алгоритмические аспекты, а не только последовательность передачи сообщений. Составне шаги позволяют графически изображать на диаграмме последовательности ветвления, циклы и другие полезные конструкции управления. Делается это очень просто: на диаграмме рисуется рамка, в углу которой указавается тип составного шага, а внутри шага указываются частичные последовтельности сообщений в соотвествии с правилами, присущими шагам данного типа. Например, для ветвлений используется имя alt, а альтернативные последовательности сообщений рисуются внутри рамки просто последовательно, друг под другом. Подпоследовательности отделяются пунктирной лининей и для них указываются соотвествующие сторожевые условия. На рис. 4.54. приведена та же диаграмма последовательности, что и на рис. 4.52 с использованием составного шага взаимодействия. Согласитесь, что нотация MSC достачно выразительна и удобна.

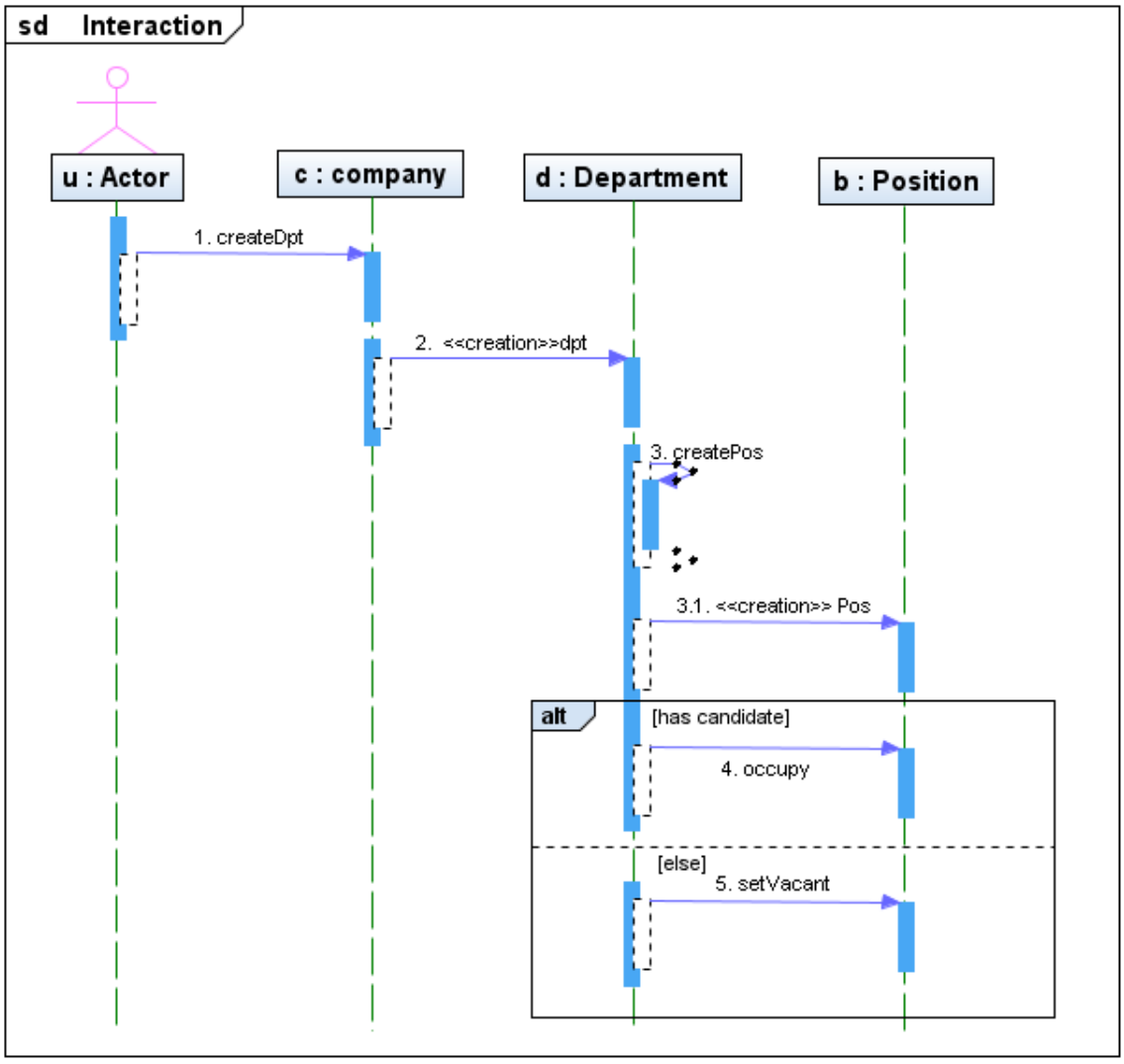



**Рис. 4.54. Составной шаг взаимодействия**

### *4.4.3. Диаграммы кооперации*

Графических ухищрений на диаграмме кооперации[122] значительно меньше — данный тип диаграмм графически очень лаконичен и, тем не менее, чрезвычайно выразителен. Поэтому при рассмотрении элементов диаграмм кооперации мы больше внимания уделяем семантике, оставляя самоочевидную нотацию для примеров. В частности, описания некоторых семантических тонкостей, относящиеся также и к диаграммам последовательности и опущенные в двух предыдущих разделах, здесь восполнены. Для начала мы остановимся на трех важных семантических понятиях, присущих диаграммам кооперации, которые не имеют броского графического выражения, но являются важным аспектом прагматики диаграмм взаимодействия и могут быть просто не замечены и пропущены читателем, если он знакомится с языком, глядя только на картинки. Поскольку данные понятия относятся к сфере семантики и прагматики, с ними не связано громких ключевых слов и броских графических символов, а потому возникает терминологическая путаница, особенно в русских переводах. Читателю не следует пугаться необычных слов в следующих абзацах. Итак, первые три понятия этого раздела:

- роль классификатора;
- роль ассоциации;
- контекст взаимодействия.

Как уже много раз было сказано, сущностями на диаграммах взаимодействия (в частности, кооперации) являются объекты. Это действительно так, но с одной оговоркой: объект на диаграмме взаимодействия может выступать в двух ипостасях. Это может быть конкретный индивидуальный объект и тогда диаграмма описывает конкретное взаимодействие с участием данного объекта (диаграмма является экземпляром взаимодействия). Но также же это может быть слот во фрейме взаимодействия, подлежащий заполнению конкретным объектом, и тогда диаграмма описывает множество взаимодействий, задавая их общую схему (диаграмма является дескриптором взаимодействия).

#### Представление знаний с помощью фреймов

Одной из ключевых проблем искусственного интеллекта является представление знаний, т. е. выбор формы хранения знаний в памяти компьютера. (Примером другой ключевой проблемы искусственного интеллекта является исследование алгоритмов поиска решения.) Были предложение различные способы представления знаний: с помощью правил продукции, с помощью логических формул и др. Одной из самых удачных была концепция фреймов, предложенная Минским более 30 лет тому назад. Согласно этой концепции, знания представляются в виде множества взаимосвязанных структур, которые Минский назвал *фреймами*.[123] Фрейм предназначен для описания одного понятия или

---

[122] Напомним, что в UML 2.0 эти диаграммы переименованы в диаграммы коммуникации.

[123] Этому термину не повезло: иногда его переводят (рамка, кадр), иногда оставляют в виде транслитерации, но всегда используют без всяких оговорок о многозначности данного термина, хотя в разных областях информатики он обозначает совершенно разные вещи: от группы битов в телекоммуникационном сообщении до области на странице Интернета.

ситуации. Он имеет имя и набор именованных *слотов*. Слот может быть не заполнен или заполнен, т. е. содержать конкретное значение, процедуру или (ссылку на) другой фрейм. Несмотря на внешнюю простоту этой идеи, она оказалась удивительно плодотворной, как в смысле конкретной разработки искусственно интеллектуальных программ, так и в смысле развития фундаментальных понятий в других областях информатики и программирования.

Второй из рассмотренных случаев, т. е. когда на диаграмме подразумевается слот объекта, подлежащий заполнению объектом, называется в UML *ролью классификатора*.[124] Синтаксически роль классификатора и конкретный объект почти неразличимы: в обоих случаях изображается стандартная фигура классификатора (прямоугольник), в которой вписано имя и классификатор, разделенные двоеточием. Различие заключается в том, что в случает роли классификатора имя предлагается не подчеркивать.

**ЗАМЕЧАНИЕ**

К сожалению, данное тонкое семантическое различие многие инструменты игнорируют и подчеркивают имя роли классификатора. В использованный при подготовке книги трафарет Visio, хотя и правильно называет фигуру — Classifier Role — но изображает ее неправильно, с подчеркиванием имени. Впрочем, поскольку также поступают и многие другие инструменты, данная неправильность стала уже правилом. Вообще говоря, вопрос о подчеркивании имени роли классификатора является спорным. С одной стороны, это не конкретный объект: в слот может быть подставлен любой подходящий объект (имя не нужно подчеркивать), с другой стороны, это все-таки слот объекта (имя объекта). В таких спорных случаях проще полагаться на инструмент (оставляя в уме собственное мнение). В частности, на некоторых наших диаграммах имена подчеркнуты, а на других — нет.

Семантически объект и слот объекта —это разные вещи. В случае конкретного объекта имя — это личное имя объекта, а в случае роли классификатора имя — это имя слота (или, как пишут авторы языка, имя роли, которую объект данного классификатора играет во взаимодействии). Самое замечательное состоит в том, что с прагматической точки зрения, это и не важно: ведь схема взаимодействия остается одной и той же, она не зависит от того, чем именно заполнен слот (лишь бы там был объект подходящего класса и посылал те сообщения, которые от него требуются).

Мы не уверены в абсолютной правомерности следующей аналогии, но все же приведем ее. Мы надеемся, что следующие две фразы для читателей прозвучат как цитаты (может из забытого источника). От перемены мест слагаемых сумма не меняется: 2+3=3+2. Сложение коммутативно: $a+b=b+a$. Первый пример — это экземпляр, второй — дескриптор. По нашему мнению, использование роли классификатора на диаграмме взаимодействия аналогично использованию алгебраической переменной $a$ при описании арифметических законов.

Совершенно аналогично понятию роли классификатора вводится понятие *роли ассоциации*. Отношения между объектами (ролями классификаторов) — это

[124] Название, прямо скажем, не очень удачное, а потому трудно переводимое. Все-таки речь идет об объекте, хотя и являющегося экземпляром определенного классификатора.

экземпляры ассоциации, т. е. связи. Но если связь связывает роли классификаторов, т. е. слоты объектов, то она сама является слотом — слотом связи, который называется ролью ассоциации. На диаграмме роль ассоциации синтаксически неотличима от связи (разве что имя ассоциации не подчеркнуто, если оно отображается).[125]

**ЗАМЕЧАНИЕ**

> У связи, в отличии от объекта, нет личного имени. Ее индивидуальность определяется набором объектов, которые она связывает.

Диаграмма кооперации (равно как и диаграммы последовательности) описывает поведение как *взаимодействие*, т. е. как протокол обмена сообщений между объектами. Один и тот же объект может участвовать в различных взаимодействиях, играя в них различные роли. Таким образом, взаимодействие всегда происходит в определенном контексте, который определяется множеством участвующих во взаимодействии объектов и связей. Несколько утрируя, можно сказать, что диаграмма кооперации, в которой не указаны сообщения (и которая, тем самым, синтаксически неотличима от диаграммы объектов) является *контекстом взаимодействия*.

Обычно контекст выбирается с расчетом на то, чтобы описать взаимодействие, имеющее определенную цель, скажем, выполнить сценарий варианта использования или операцию (это наиболее типичные примеры применения диаграмм кооперации). Варьируя контекстом (добавляя и убирая роли классификаторов и ассоциаций), можно скрывать или показывать детали взаимодействия, описывая поведение на разных уровнях абстракции. Чтобы пояснить данный тезис, нужно вернуться к понятию номера сообщения, упомянутому в разд. 4.4.1 и объяснить его детально.

*Номер сообщения* определяется в соответствии с положением сообщения в последовательности сообщений данного взаимодействия. Если во взаимодействии используются только простые или асинхронные сообщения (см. табл. 4.5), то сообщения просто нумеруются, обычно подряд: 1, 2, 3 и т. д. Сообщения с меньшими номерами предшествуют сообщениям с большими номерами. Если же используются вложенные потоки управления, т. е. сообщения типа вызова операции (см. табл. 4.5), то сообщения нумеруются более сложным образом. Допустим сообщение вызова операции имеет номер *x*. Тогда сообщения, отправляемые при выполнении этой операции будут иметь номера *x*.1, *x*.2 *x*.3 и т. д. Первое сообщение, отправляемое при выполнении вызова *x*.1 будет иметь номер *x*.1.1 и т. д. Количество точек в номер соответствует уровню вложенности потока управления, т. е. глубине стека вызовов (см. разд. 4.4.2). Таким образом, например, сообщение с номером 1.2 предшествует сообщению с номером 1.2.3, а сообщение 2.1, напротив, следует за сообщением 1.2.3. Если первым во взаимодействии является сообщение вызова, то его номер часто не указывают (такое сообщение как бы имеет неявный номер 0), чтобы не загромождать диаграмму повторяющимся всюду номером первого сообщения и ненужной точкой.

Такая иерархическая десятичная нумерация удобна, поскольку задает относительный порядок сообщений на данном уровне вложенности вызовов и

---

[125] Беда в том, что инструменты забывают подчеркивать имя связи.

позволяет описывать взаимодействие с нужной степенью детальности. Например, если требуется раскрыть детали выполнения операции, вызванной сообщением с номером x, т. е. показать, какие сообщения отправляются в процессе выполнения данной операции, то достаточно добавить на диаграмму сообщения с номерами *x*.1, *x*.2 и т. д. (а также соответствующие целевые объекты — роли классификаторов). Если же детали не нужны на данной диаграмме, то сообщения можно удалить, при этом номера других сообщений не меняются. Другими словами, в последовательности сообщений, образующей взаимодействие, можно показывать больше или меньше элементов, и при этом перенумерации элементов не требуется.
На диаграмме последовательности время жизни объекта относительно данного взаимодействия показывается графически, с помощью смещения вниз символа объекта, создаваемого в процессе взаимодействия и с помощью символа уничтожения объекта (косой крест) на линии жизни уничтожаемого объекта. На диаграмме кооперации для этой цели используются специальные ключевые слова, которые указываются для ролей классификаторов в форме стандартных стереотипов и/или в форме стандартных ограничений для сообщений (табл. 4.6).

**Таблица 4.6.** Ключевые слова для описания времени жизни объектов во взаимодействии на диаграмме кооперации

| Ключевое слово | Способ использования | Описание |
|---|---|---|
| `create` | стереотип операции в сообщении | Операция создает объект, т. е. данное сообщение является вызовом конструктора |
| `destroy` | стереотип операции в сообщении | Операция уничтожает объект, т. е. данное сообщение является вызовом деструктора |
| `destroyed` | ограничение роли классификатора | Объект уничтожается в процессе описываемого взаимодействия |
| `New` | ограничение роли классификатора | Объект создается в процессе описываемого взаимодействия |
| `transient` | ограничение роли классификатора | Объект создается и уничтожается в процессе описываемого взаимодействия. Такой объект называется *временным*. Данное ограничение эквивалентно одновременному указанию ограничений `new` и `destroyed` |

Рассмотрим пример из информационной системы отдела кадров, уже разобранный нами на рис. 4.49. Рис. 4.55 семантически эквивалентен рис. 4.49 — эти диаграммы описывают одно и то же взаимодействие и поведение. Мы рекомендуем читателю просто сравнить эти две диаграммы.

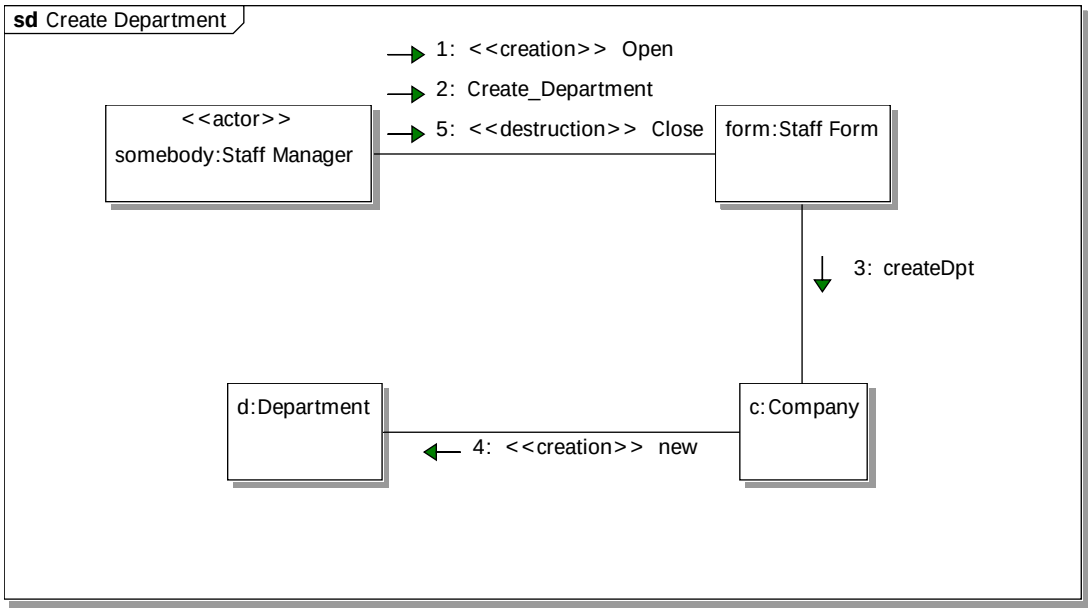



**Рис. 4.55. Диаграмма кооперации**

На диаграмме последовательности невозможно отобразить связи, имеющие место, но не используемые для посылки сообщений, а на диаграмме кооперации это вполне возможно, а иногда бывает очень полезно. Большую часть деталей статической структуры контекста, которые можно показать на диаграмме классов, можно показать и на диаграмме кооперации. Сюда относятся имена ролей на полюсах связей (ролей ассоциаций), направление навигации, символы агрегации и композиции, квалификаторы и т. д. (см. разд. 4.2.6).

Роли классификаторов и ассоциаций могут применяться на диаграмме кооперации по разному. Некоторые связи (роли ассоциаций) могут быть экземплярами реальных (хранимых) ассоциаций, а другие могут быть сугубо временными, нужными только для однократной передачи сообщения. Аналогично и объекты (роли классификаторов) могут быть глобально существующими в системе, а могут быть локальными объектами, временно используемыми для организации взаимодействия. Чтобы отобразить все эти особенности, на диаграмме кооперации используются стандартные стереотипы для полюсов ролей ассоциации. Список этих стереотипов приведен в табл. 4.7.

**Таблица 4.7.** Стереотипы полюса роли ассоциации на диаграмме кооперации

| Стереотип | Описание |
|---|---|
| «association» | Объект на полюсе роли ассоциации связан с объектом на противоположном полюсе фактической связью, реализующей ассоциацию |
| «global» | Объект на полюсе роли ассоциации имеет глобальную область определения относительно объекта на противоположном полюсе |
| «local» | Объект на полюсе роли ассоциации имеет локальную область определения относительно объекта на противоположном полюсе, т. е. является временным объектом для выполнения операции |
| «parameter» | Объект на полюсе роли ассоциации является параметром операции объекта на противоположном полюсе |

| | |
|---|---|
| «self» | Роль ассоциации является фиктивной связью, введенной для моделирования вызова операции данного объекта |

Рассмотрим в качестве примера реализацию операции рисования многоугольника (см. предварительно, рис. 4.19). Мы подробно прокомментируем обозначения на рис. 4.56, чтобы объяснить все детали нотации на диаграмме сообщений. Взаимодействие, описывающее поведение операции, инициируется сообщением `draw`. Для этого сообщения номер не задан (обычная для UML практика опускания очевидных вещей), а посылающий сообщение объект изображен в виде анонимного действующего лица, символизирующего источник вызова операции. Для выполнения операции `draw` вызывает операция `drawSegment` ($n$–1 раз, где $n$ — количество вершин многоугольника). Соответствующее сообщение (оно имеет номер 1) нагружено на связь со стереотипом «`self`», которая не является реальным экземпляром ассоциации, а введена в модель только для того, чтобы послать сообщение объекту из операции этого же объекта. При каждом вызове операции `drawSegment` определяются две вершины, которые являются началом и концом стороны многоугольника. При этом в модели не показаны никакие сообщения: подразумевается, что нужную информацию можно получить, указывая значение квалификатора ($i$ и $i$+1, соответственно).[126] После этого с помощью сообщения с номером 1.1 создается временный объект `line` и затем вызывается его операция `draw` (сообщение с номером 1.2). То обстоятельство, что объект `line` временный (существует только на время выполнения операции `drawSegment`), указывать с помощью стандартного ограничения `transient` не обязательно: достаточно стереотипа «`local`», указанного для имени полюса роли ассоциации `segment`.

[126] Мы просим читателя извинить кривоватые квадратики квалификаторов и неудачно расположенные значения $i$ и $i$+1. Мы не чувствуем себя виноватыми — так сделан использованный трафарет Visio. Но согласитесь, что UML выдерживает испытание небрежностью разработчиков инструмента — диаграмма все равно читается.

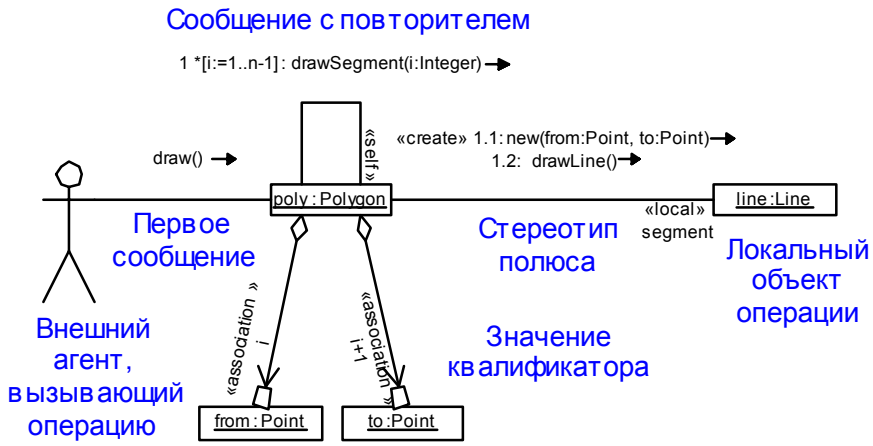



**Рис. 4.56. Структурные связи и стереотипы полюса роли ассоциации на диаграмме кооперации**

Обычно объект, участвующий во взаимодействии, отображается на диаграмме кооперации один раз, даже если он получает и отправляет несколько сообщений. Однако иногда нужно показать, что в процессе взаимодействия объект меняет свое состояние. Для этого используется зависимость с соответствующим стереотипом.

- «become» — зависимость с данным стереотипом проводится от объекта в исходном состоянии к тому же самому объекту, но в измененном состоянии. Имя состояния, как обычно, указывают в квадратных скобках.
- «copy» — данный стереотип означает, что создается новый объект — копия исходного. После создания копия объекта существует как новый независимый объект.

Рассмотрим еще раз пример из информационной системы отдела кадров, относящийся к созданию подразделения. Допустим, что кадровая политика организации требует немедленного назначения начальника при создании подразделения (что вполне разумно, по нашему мнению). В терминах нашей модели это означает, что при создании новый объект `boss` класса `Position` находится в состоянии `vacant`, но ему сразу же посылается сообщение `occupy`, в результате чего объект `boss` переходит в состоянии `occupied`. Описание данного взаимодействия может быть представлено диаграммой на рис. 4.57 (полезно сопоставить эту диаграмму с диаграммой на рис. 4.53). Иногда для зависимости указывают номер (как у сообщения), чтобы показать, при выполнении какой именно операции происходит изменение состояния объекта.[127]

[127] На нашей диаграмме это не указано, т. к. использованный инструмент не поддерживает данную возможность. В данном случае перед стереотипом «become» можно было бы указать номер 2.2.1.

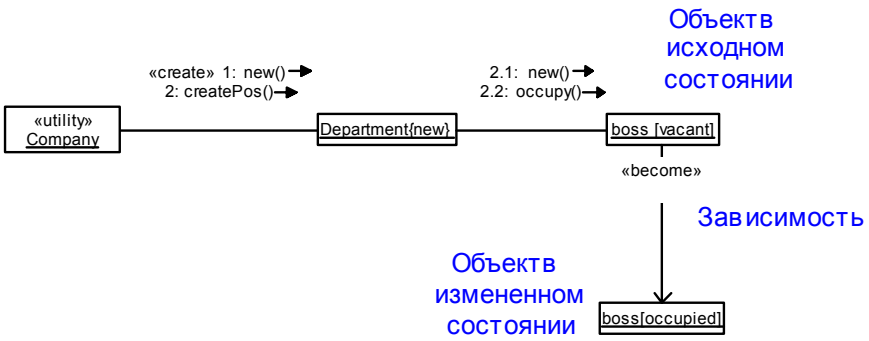



**Рис. 4.57. Использование зависимости на диаграмме кооперации**

Нам осталось описать один дополнительный элемент нотации, применяемый на диаграмме кооперации — мультиобъект.

*Мультиобъект* — это множество объектов, к которому сообщение можно отправить, как к единому целому.

Например, допустим, что в информационной системе отдела кадров имеется множестве объектов класса `Person`, представляющих информацию о людях. Разумно предположить, что имеется функция `findPerson`, позволяющая найти конкретный объект по значению некоторого атрибута, скажем, по имени. В таких условиях предыдущий сценарий создания подразделения с немедленным заполнением вакансии руководителя можно более детально и точно описать с помощью диаграммы кооперации, приведенной на рис. 4.53. Фактически, данная диаграмма кооперации является детальной реализацией операции `createDpt` (полезно сравнить рис. 4.53 и 4.58).

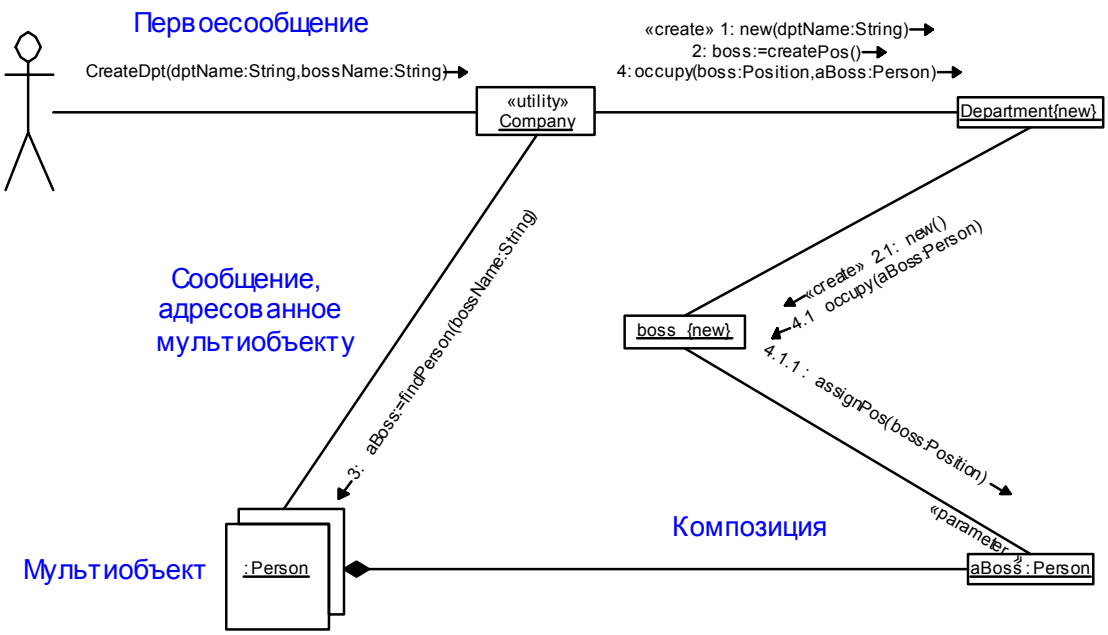



**Рис. 4.58. Применение мультиобъекта на диаграмме кооперации**

# 4.5. Моделирование параллелизма

Термин *параллельность* в программировании, вообще говоря, означает "одновременное" выполнение нескольких активностей. Слово "одновременное" в данном контексте означает, что невозможно (или не нужно) указать, какая из активностей происходит раньше другой во времени. Другими словами, параллельные процессы не упорядочены во времени. Тем самым термин "параллельный" противопоставляется термину "последовательный": последовательные активности упорядочены во времени, причем строго.

В UML с каждой параллельно выполняемой активностью связывается поток управления. Таким образом, моделирование параллельного (равно как и квазипараллельного) поведения средствами UML сводится к описанию параллельных потоков управления и способов взаимодействия между ними.

Средства описания параллелизма в UML отнюдь не противопоставлены средствам описания последовательного поведения, напротив, они образуют единое целое, поскольку параллельное программирование скорее общее правило, нежели экзотическое исключение. Мы отдели обсуждение средств описания параллельного поведения от средств описания последовательного только с целью упростить изложение основных идей каждого из типов канонических диаграмм, используемых для описания поведения. В последующих разделах поочередно рассматриваются отдельные опущенные выше детали и конструкции диаграмм описания поведения, относящиеся к моделированию параллелизма, причем эти средства начинаются с простых и часто используемых и описываются в той же последовательности, которая выбрана для канонических диаграмм.

### *4.5.1. Взаимодействие последовательных процессов*

Начнем с обсуждения самого распространенного случая параллелизма. Допустим, что имеются[128] несколько параллельно выполняющих процессов, каждый из которых имеет свой собственный поток управления. Если эти процессы никак не взаимодействуют друг с другом (самый распространенный случай), то нам ничего и не нужно моделировать. Поведение каждого из процессов может быть описано любым из рассмотренных выше способов (конечным автоматом, блок-схемой или диаграммой взаимодействия). Если процессы не взаимодействуют, то они независимы и с точки зрения конечного результата поведения (состояния системы в целом) неважно, как именно реализованы параллельные процессы: как физически параллельные выполняемые на многопроцессорном компьютере, квазипараллельные или последовательно выполняемые в произвольном порядке. Но с точки зрения других аспектов поведения, таких как, например, производительность, время реакции, пропускная способность способ реализации параллелизма является очень существенным. Мы вернемся к этому вопросу в разд. 4.5.5, а здесь сосредоточимся на моделировании поведения взаимодействующих процессов, которое должно обеспечивать определенную функциональность, т. е. изменения состояния системы.

Взаимодействие в UML моделируется с помощью сообщений: синхронных (вызовы операций) и асинхронных (посылка сигналов), см. разд. 4.4.1. Типичный вариант: взаимодействие машин состояний разных классов с помощью действий, выполняемых на переходах.

Рассмотрим несколько примеров на эту тему из информационной системы отдела кадров. Пусть у нас определены два класса: `Position` (должность) и `Person` (сотрудник), представленные на рис. 4.59. У объектов этих классов имеются по два состояния: должность может быть вакантна (`Vacant`) или занята определенным сотрудником (`Occupied`) и сотрудник может быть не занят (`Free`) ли назначен на определенную должность (`Assigned`). Соответственно, у каждого из классов есть конструктор (`new`) и деструктор (`destroy`) и по паре операций, которые ответственны за изменение состояния объекта: у класса `Position` операции называются `occupy` (занять должность) и `free` (освободить должность), а у класса `Person` — `assign` (назначить на должность) `demote` (освободить от должности). У каждого класса есть скрытый атрибут, предназначенный для хранения ссылки на объект другого класса (т. е. между этими классами существует ассоциация) и предусмотрена операция без побочного эффекта, возвращающая значение данного (`getPerson` и `getPosition`, соответственно).

---

[128] Вопрос о том, как возникают (и исчезают) параллельные потоки управления чуть более сложен и мы вернемся к нему несколько раз в следующих четырех разделах.

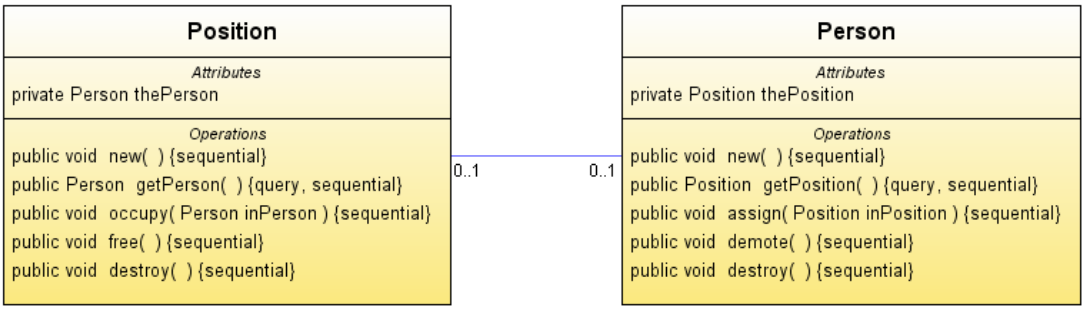



**Рис. 4.59. Классы Position и Person**

Рассмотрим теперь, как должна выполнятся операция назначения сотрудника на должность. Мы оставим в стороне вопрос о том, в каком классе разумно определить данную операцию (на самом деле это совершенно не важно), и положим, что операция назначения сотрудника на должность имеет два параметра — сотрудника и должность:

```
assignP2P(person:Person,position:Position)
```

Допустим, что требуется обеспечить элементарный порядок в учете кадров (на программистском языке — целостность данных): если сотрудник А назначен на должность Б, то и в должности Б должно быть записано, что ее занимает сотрудник А и наоборот. Другими словами, занятые должности и сотрудники должны взаимно однозначно соответствовать друг другу, а свободные должности и сотрудники должны быть действительно свободны и не должны содержать неадекватных ссылок друг на друга.[129] Требуемое поведение операции `assignP2P` можно описать с помощью диаграммы объектов (фактически, это контекст взаимодействия, см. разд. 4.4.3), на которой показано, как должны измениться связи между объектами в результате выполнения операции (рис. 4.60). В данном описании контекста рассматривается типичный сценарий, в котором до выполнения операции сотрудник занимает некоторую должность, а целевая должность вакантна.

[129] Мы не утверждаем, что наше решение с хранением взаимных ссылок является единственно возможным, и даже не утверждаем, что это решение является хорошим — мы рассматриваем пример на взаимодействие конечных автоматов.

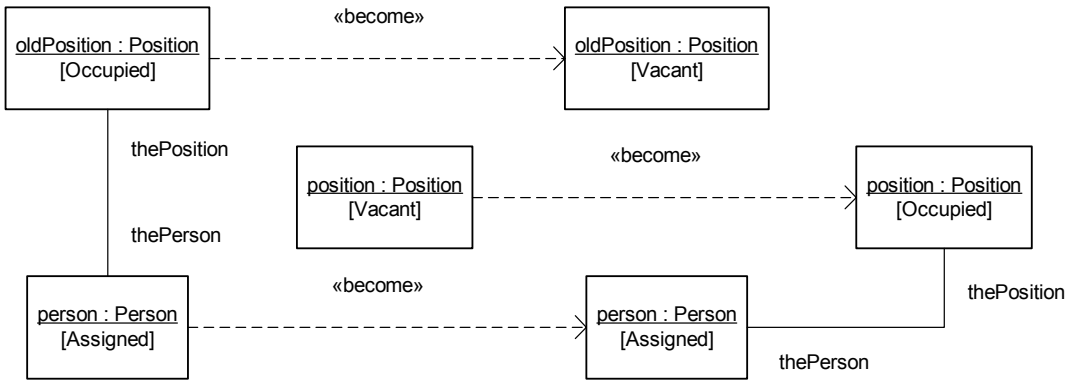



**Рис. 4.60. Изменение связей и состояний объектов при выполнении операции перевода сотрудника**

Мы видим, что при назначении сотрудника на должность задействованы три объекта: Требуемое поведения может быть обеспечено за счет взаимодействия автоматов, реализующих поведение каждого из этих объектов. На рис. 4.61 и 4.62 приведены диаграммы машин состояний для классов `Position` и `Person`, соответственно. Обратите внимание на изоморфизм графов переходов этих машин состояний — это не случайное обстоятельство. Классы `Position` и `Person` в нашей модели совершенно равноправны и их поведение по отношению друг к другу совершенно симметрично.

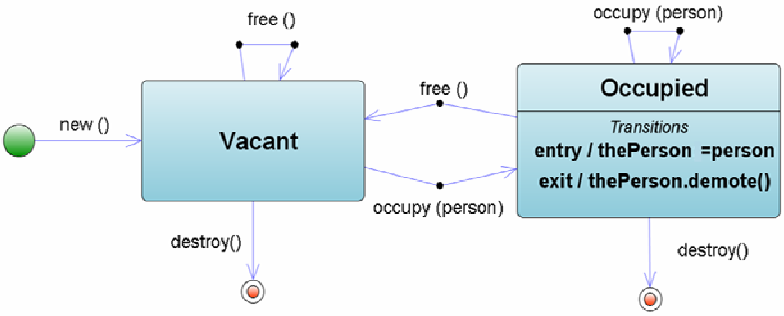



**Рис. 4.61. Машина состояний класса Position**

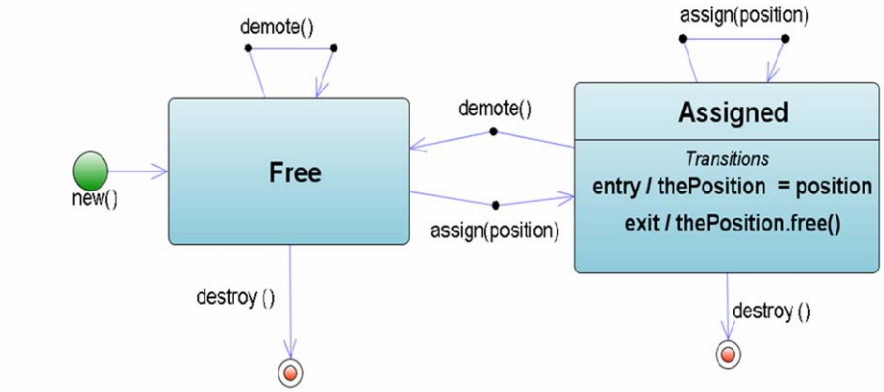



**Рис. 4.62. Машина состояний класса Person**

В таком случае, требуемая операция назначения сотрудника на должность может быть реализована двумя вызовами операций объектов, являющихся аргументами операции:

```
position.occupy(person)
person.assign(position)
```

Мы подошли к кульминации данного примера: указанные две операции можно вызвать в любом порядке или *параллельно* , более того, их можно вызывать с ожиданием возврата управления или без ожидания — в любом случае взаимодействие автоматов, приведенных на рис. 4.61 и 4.62, обеспечит требуемое поведение (если только в процесс обмена сообщениями не вмешается "посторонний" вызов операции одного из этих объектов). Наш пример достаточно прост для того, чтобы последнее утверждение можно было считать очевидным, но для усиления примера мы приводим на рис. 4.63 и рис. 4.64 диаграммы последовательности, описывающие возможные протоколы взаимодействия при выполнении операции `assignP2P`. Для наглядности, мы указали на диаграммах состояния объектов.

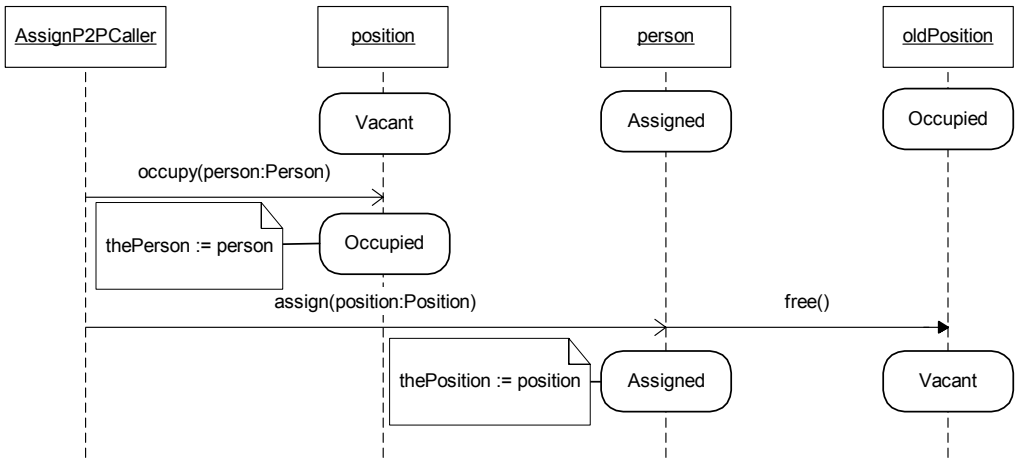



**Рис. 4.63. Первый сценарий выполнения операции assignP2P**

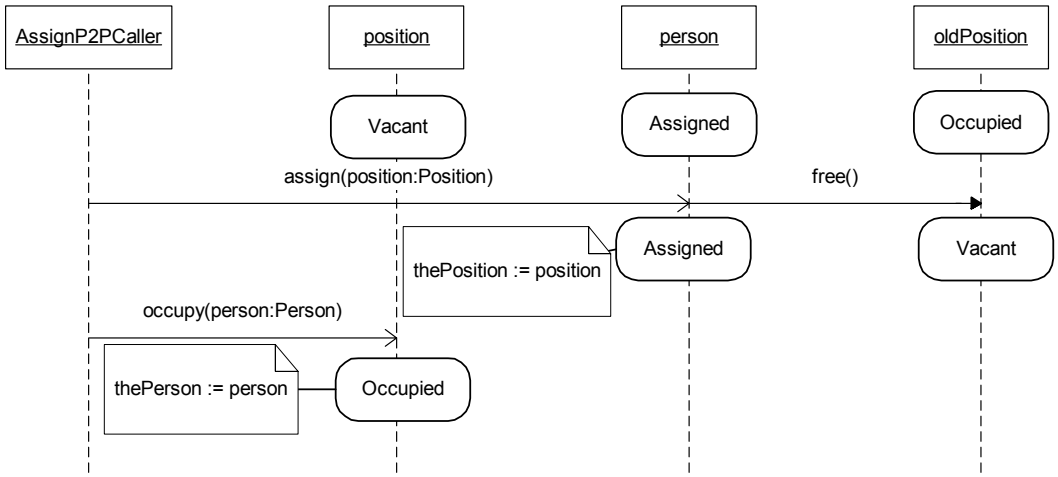



**Рис. 4.64. Второй сценарий выполнения операции assignP2P**

В качестве очень полезного упражнения[130] мы оставляем читателю разбор случая, когда два сообщения: `occupy(person)` объекту `position` и `assign(position)` объекту `person` отправляются *одновременно*.

Параллельное программирование — мощный способ моделирования поведения, и взаимодействующие конечные автоматы — великолепное средство параллельного программирования.

Утверждение предыдущего абзаца, безусловно, верно, но у медали есть и оборотная сторона. Исчерпывающее изложение приемов параллельного программирования не входит в множество целей данной книги, но умолчать о типичных трудностях, подстерегающих на этому пути, мы не вправе. Рассмотрим два несложных, но типичных примера ошибок, которые можно допустить, моделируя поведение с помощью взаимодействующих автоматов.

Первый пример — "усовершенствование" предыдущего примера. Рассмотрим следующее рассуждение. Объекты класса `Position` и класса `Person` либо существуют сами по себе, либо образуют пары и хранят ссылки друг на друга. Пара объектов с правильными ссылками образуется при совместном выполнении соответствующих операций `occupy` и `assign`. Значит, для обеспечения целостности данных нужно гарантировать, чтобы любой вызов операции `occupy` сопровождался вызовом операции `assign` и наоборот. То есть будет достаточно вызвать одну из операций, а вторая всегда будет вызвана автоматически. Это можно сделать, включив в машину состояний объекта вызов парной операции, например, так, как показано на рис. 4.65.[131]

---

[130] Это упражнение нетрудно выполнить "устно", глядя на диаграммы 4.63 и 4.64 и отслеживая переходы и выполнение действий в автоматах. При этом не следует забывать, что действия на переходах и на входе/выходе атомарны и мгновенны.

[131] Для наглядности и экономии места мы поместили на одну диаграмму две машины состояний, убрав не нужные в данном контексте начальные и заключительные состояния.

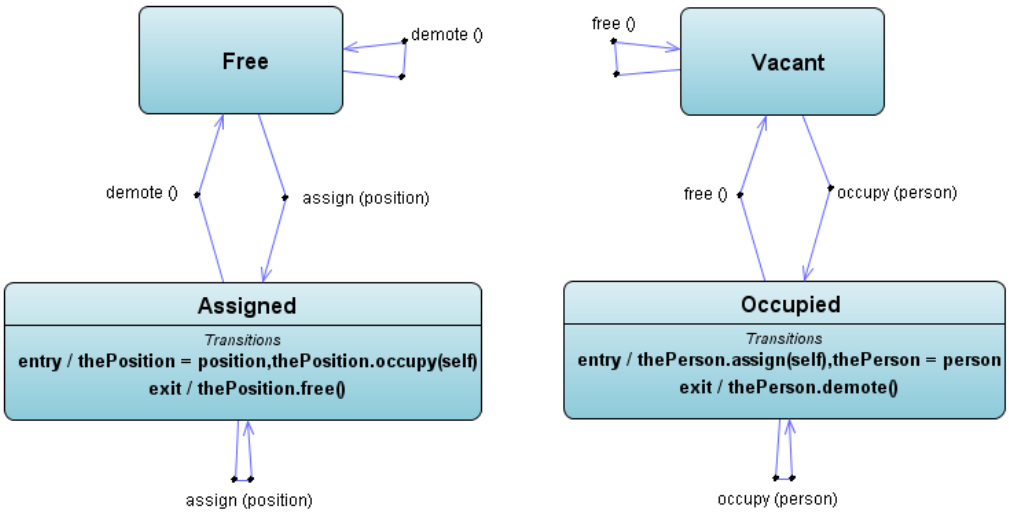



**Рис. 4.65. Бесконечная взаимная рекурсия**

Диаграммы на рис. 4.65 и предшествующее рассуждение выглядят очень правдоподобно, но на самом деле являются ошибочными. Действительно, как нетрудно сообразить, вызов одной операции повлечет вызов парной, та, в свою очередь, еще раз вызовет исходную и т. д., возникнет бесконечная взаимная рекурсия. "Улучшение" привело к тому, что модель стала противоречивой.

Второй пример — так называемая ситуация тупика, или взаимной блокировки. На рис. 4.66 приведены две абстрактных диаграммы последовательности, иллюстрирующих природу взаимной блокировки. Процесс слева запускается при получении сигнала A, после чего посылает сигнал B. Процесс справа запускается при получении сигнала B, после чего запускает сигнал A. Очевидно, что эти процессы не смогут начать работу — произойдет "зависание" системы.

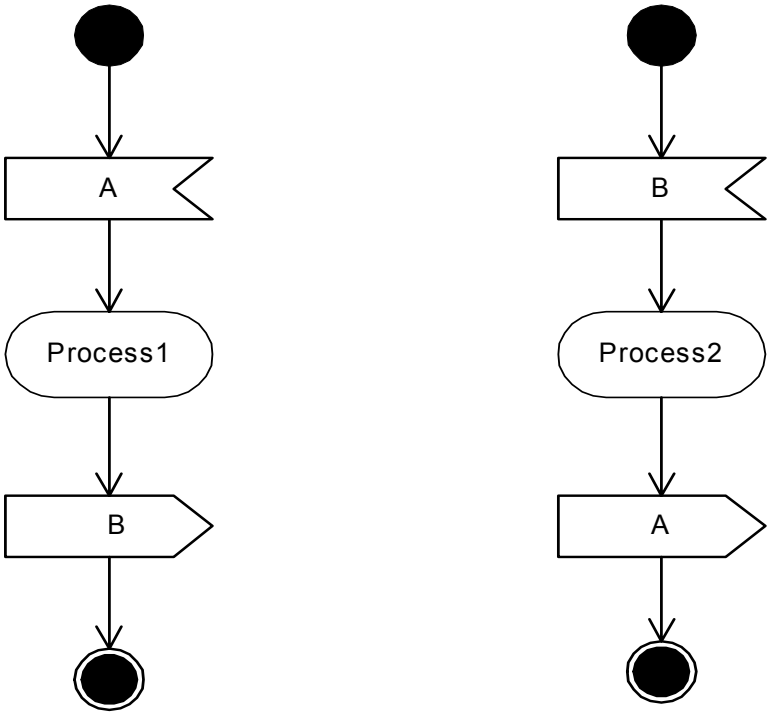



**Рис. 4.66. Взаимная блокировка**

Многие трудности такого рода, возникающие при описании поведения, хорошо изучены и известны различные приемы их преодоления. Мы не будем здесь их описывать — пары примеров достаточно, чтобы обратить внимание читателя на то обстоятельство, что при моделировании взаимодействия параллельных процессов есть над чем подумать.

### *4.5.2. Параллельные состояния и составные переходы*

В примерах предыдущего раздела не использовались никакие специальные средства моделирования параллелизма, мы применяли средства "параллельности по умолчанию", которая в принципе заложена в объектно-ориентированную парадигму — поведение программы определяется взаимодействием объектов путем обмена сообщениями. Но в UML предусмотрен целый спектр специальных средств, предназначенных для более тонкого, детального и явного моделирования поведения при параллельном выполнении приложения. В этом разделе рассматриваются средства, применяемые на диаграммах состояний.

На диаграммах состояний параллельное выполнение моделируется с помощью параллельных[132] составных состояний (см. разд. 4.2.3). По существу, идея параллельных состояний очень проста и легко воспринимается: внутрь составного состояния вложено несколько машин состояний, которые работают параллельно. Чтобы интуитивное понимание этой идеи обрело прагматику, необходимую для практического применения при моделировании, необходимо ввести и точно определить несколько специфических понятий и конструкций, а именно:

---

[132] Напомним, что в UML 2 параллллельные состояния переименованы в ортогональные. Такое переименование оправдано, поскольку фактически параллельные составные состояния фактически являются прямым произведением в терминах теории автоматов.

- активное состояние;
- конфигурация активных состояний;
- область параллельного составного состояния;
- составной переход;
- синхронизирующее состояние.

Выше несколько раз было использовано (без специального определения) интуитивно ясное понятие активного состояния. Пора дать более строгое определение. Начнем с фиксации интуитивного очевидного.

*Активное состояние* — это состояние, в котором находится машина состояний в данный момент.

Применительно к машине состояний, в которой все состояния простые (т. е. машина состояний — обычный конечный автомат) данное определение однозначно: действительно, по определению такая машина во время своей работы всегда находится ровно в одном из своих простых состояний (напомним, что простые переходы мгновенны, а в специальных состояниях машина *не* находится).

Если машина содержит последовательные составные состояния, то ее структуру удобно представить в виде корневого (ориентированного) дерева: корнем является составное состояние, соответствующее всей машине в целом (это объясняет, зачем нужен атрибут `top` у метакласса `StateMachine` на рис. 4.4), а листьями являются простые состояния. На рис. 4.67 приведена соответствующая иллюстрация: слева машина состояний в нотации UML, а справа — представление этой машины в виде дерева.[133]

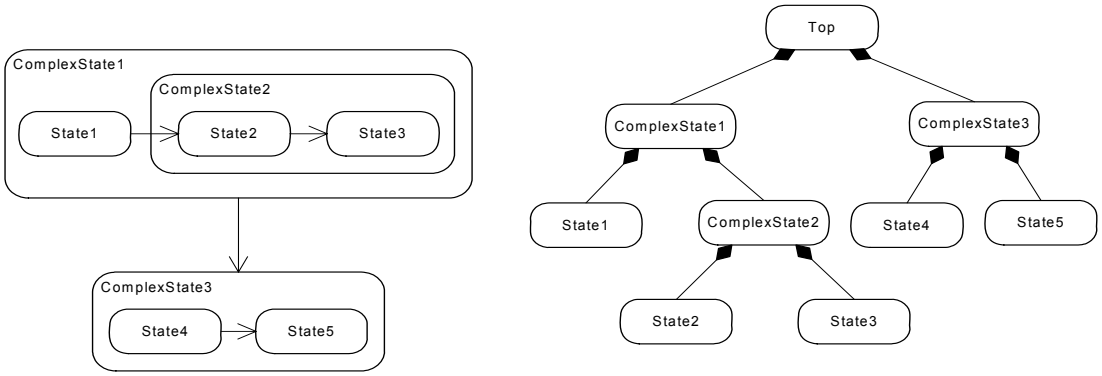



**Рис. 4.67. Машина состояний и ее представление в виде дерева**

Из нашего объяснения семантики последовательных составных состояний и простых переходов между ними, приведенного в разд. 4.2.3, следует, что в последовательной машине состояний (т. е. в машине, в которой все составные состояния последовательные) также ровно одно простое состояние является активным в каждый момент времени работы машины. Но наряду с активным простым состоянием естественно считать активными и все составные состояния, объемлющие данное простое состояние — ведь машина пребывает и в этих объемлющих состояниях также. Все активные состояния образуют так называемую

[133] Обращаем внимание, что дерево справа *не* является допустимой диаграммой UML, хотя и нарисовано с использованием нотации UML.

*конфигурацию активных состояний*, это путь от корня к листу в дереве вложенности состояний.

Обратимся теперь к параллельным составным состояниям. Начнем с нотации. Параллельное составное состояние может иметь три раздела: раздел имени, раздел составляющих (действие на входе, действие на выходе, внутренняя активность и внутренние переходы) и раздел вложенных состояний. Относительно разделов имени и составляющих здесь нечего добавить к тому, что сказано в разд. 4.2.1 и 4.2.3, разве что стоит отметить, что оба этих раздела могу быть опущены (в то время как для простых состояний раздел имени является обязательным). В разделе вложенных состояний в случае последовательного составного состояния просто изображается вложенная машина состояний, а в случае параллельного составного состояния данный раздел делится горизонтальными пунктирными линиями на несколько *областей*, в каждой из которых изображается вложенная машина состояний (рис. 4.68).

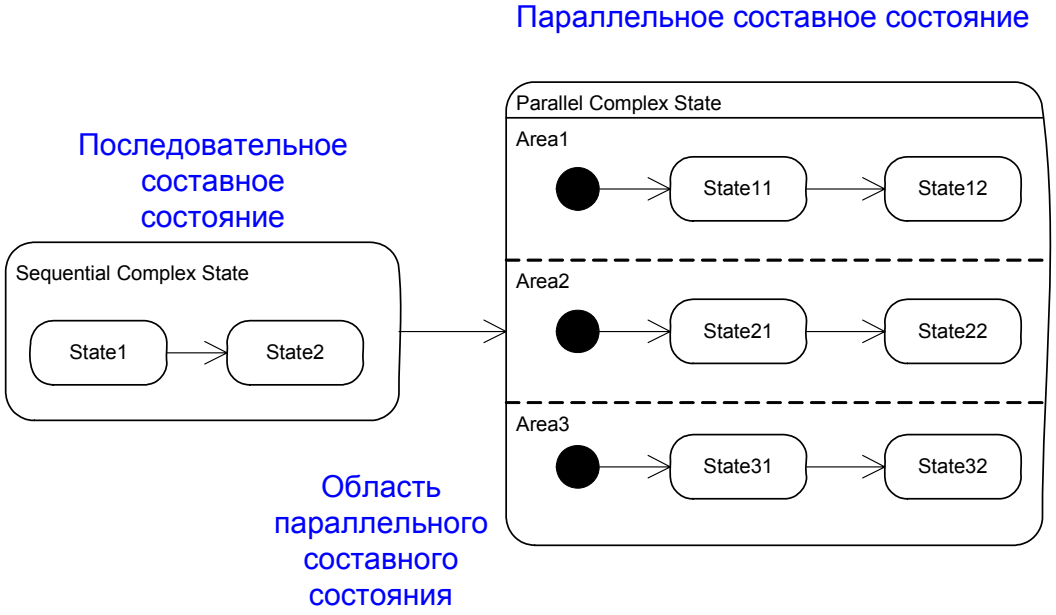



**Рис. 4.68. Параллельное (ортогональное) составное состояние**

Данная нотация весьма наглядна, но до некоторой степени скрывает одно важное семантическое обстоятельство, которое иногда ускользает от внимания пользователей UML при поверхностном знакомстве. А именно, каждая область параллельного составного состояния — это, в свою очередь, составное состояние и может иметь все составляющие состояния. На диаграммах обычно областям не дают имен (на нашей диаграмме они как раз специально указаны), поэтому кажется, что вложенная машина состояний как бы "висит в воздухе" внутри составного параллельного состояния, но на самом деле вложенная машина "завернута" в составное состояние, которое и является непосредственной составляющей данного параллельного составного состояния. Таким образом, семантика параллельных и последовательных составных состояний совершенно симметрична. И в том и в другом случае составное состояние имеет несколько вложенных состояний. Если составное состояние активно, то в случае

последовательного составного состояния ровно *одно* из вложенных состояний активно в каждый момент. В случае параллельного составного состояния *все* вложенные состояния активны в каждый момент. Подобно тому, как структуру машины состояний с последовательными составными состояниями удобно представлять в виде обычного ориентированного дерева, структуру машины состояний удобно представлять в виде дерева И/ИЛИ.

### Дерево и/или

*Дерево И/ИЛИ* — это один частный случай ориентированного гиперграфа, который нашел удивительно широкое применение при решении самых разнообразных задач. В дереве И/ИЛИ имеется ровно один узел, в который не входят дуги (корень дерева); во все остальные узлы дерева входит ровно одна дуга. Среди них имеется некоторое количество узлов, из которых не исходят дуги (листья дерева). Все не листовые узлы разбиваются на два класса: узлы типа "И" и узлы типа "ИЛИ". Из узла типа "ИЛИ" исходит несколько обычных дуг, каждая из которых ведет в свой узел, а из узла типа "И" исходит одна гипердуга, которая ведет сразу в несколько узлов.

Таким образом, движение по дереву И/ИЛИ (гиперпуть) от корня к листьям происходит следующим образом: находясь в узле типа "ИЛИ", нужно выбрать одну из исходящих дуг и перейти по ней, попадая в первый *или* второй *или* третий и т. д. дочерний узел, а находясь в узле типа "И" нужно выбрать всю (единственную) гипердугу и перейти по ней, попадая в первый *и* второй *и* третий и т. д. дочерний узел. Отсюда и происходит название "дерево И/ИЛИ".

Деревья И/ИЛИ очень часто применяются в искусственном интеллекте. Например, при программировании дискретной игры для двух играющих (игроки по очереди делают ходы, в результате чего дискретно меняется позиция) с точки зрения одного из играющих игру очень удобно представить деревом И/ИЛИ, в котором узлы соответствуют позициям, а дуги — ходам. При этом позиции, в которых делает ход данный игрок, будут иметь тип "ИЛИ", а позиции, в которых очередь хода у противника, будут иметь тип "И". Действительно, при составлении стратегии игры для первого игрока, он может выбирать тот *или* другой ход когда очередь хода за ним, но когда ходит противник, выбирать не приходится — нужно учесть все возможные ответы противника, т. е. они соединены связкой "И".

Общепринятого способа изображения произвольных гиперграфов на диаграммах мы не знаем, но для деревьев И/ИЛИ есть несколько вариантов их изображения, которые достаточно часто используются и многим знакомы. Во-первых, можно указать тип узла (например, поставив в узел метку "И" или "ИЛИ"). Во-вторых, можно указать тип исходящей дуги (например, соединив линией части гипердуги).

На рис. 4.69 представлено дерево И/ИЛИ для машины состояний, изображенной на рис. 4.68. Узлы типа "ИЛИ" помечены "OR", а узел типа "И", соответственно, — "AND".

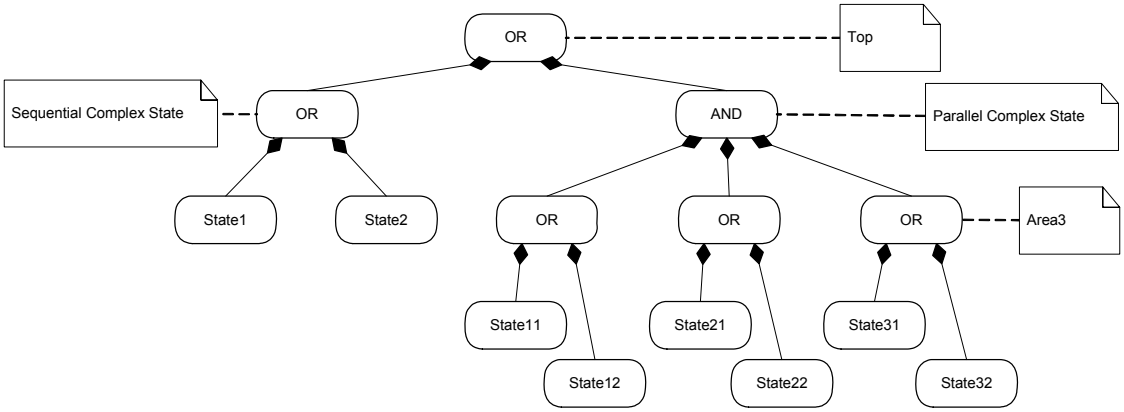



**Рис. 4.69. Дерево И/ИЛИ для машины состояний с параллельными составными состояниями**

Теперь мы можем дать точное определение:

| *Конфигурация активных состояний*— это путь (гиперпуть) от корня к листьям в дереве (дереве И/ИЛИ) машины состояний. |
|---|

Всего машина может иметь столько различных конфигураций активных состояний, сколько существует различных путей от корня к листьям. Например, на рис. 4.59 в дереве справа имеется 5 различных путей и, соответственно, 5 конфигураций активных состояний у машины слева, а в дереве на рис. 4.69 имеется 10 различных гиперпутей от корня к листьям и соответственно, 10 конфигураций активных состояний у машины на рис. 4.68.[134]

Что же означает переход *из* параллельного составного состояния или *в* параллельное составное состояние? Ответ на этот вопрос подобен ответу на аналогичный вопрос для последовательных составных состояний, который дан в табл. 4.3. В случае параллельных составных состояний нужно дополнительно учесть, что *все* области участвуют в переходе.

- Переход *в* параллельное составное состояние означает одновременный переход в начальные состояния всех областей данного параллельного составного состояния (т. е., фактически, параллельный запуск всех вложенных машин состояний); в случае наличия перехода в параллельное составное состояние каждая область должна иметь единственное начальное (или историческое состояние), в противном случае модель считается противоречивой.
- Переход по событию и/или со сторожевым условием (т. е. любой переход не по завершении) *из* параллельного составного состояние означает распространение данного перехода на все вложенные состояния все областей; если унаследованный переход конфликтует с локально определенным переходом, то последний имеет приоритет (модель не считается противоречивой).
- Переход по завершении *из* параллельного составного состояния срабатывает в том и только в том случае, когда машина состояний в каждой области перешла в заключительное состояние (т. е., фактически, это означает синхронное завершение всех вложенных машин состояний); в случае наличия перехода по завершении из параллельного составного состояния каждая область должна иметь заключительные состояния; если какая либо из вложенных машин имеет

[134] Очень советуем читателю проверить, что он видит все 10 гиперпутей на рис. 4.69.

заключительное состояние, то и все остальные должны иметь заключительные состояния и должен быть определен единственный переход по завершении, в противном случае модель считается противоречивой.

Таким образом, семантика переходов, не пересекающих границу состояния, для параллельных и последовательных составных состояний аналогична. Для описания переходов, пересекающих границу параллельного составного состояния вводится специальное понятие — составной переход.

*Составной переход* — это переход, который начинается и/или заканчивается в нескольких состояниях.

- Если переход имеет одно исходное и несколько целевых состояний, то это соответствует разветвлению потока управления на несколько параллельных потоков; при этом целевые состояния должны быть вложенными состояниями областей параллельного составного состояния — по одному на каждую область.
- Если переход имеет несколько исходных и одно целевое состояние, то это соответствует слиянию нескольких потоков управления в один; при этом исходные состояния должны быть вложенными состояниями областей параллельного составного состояния — по одному на каждую область.
- Если переход имеет несколько исходных и несколько целевых состояний, то это соответствует синхронизации нескольких параллельных потоков управления; при этом при этом исходные состояния должны быть вложенными состояниями областей одного параллельного составного состояния — по одному на каждую область и целевые состояния должны быть вложенными состояниями областей параллельного составного состояния (возможно, другого) — также по одному на каждую область.

В любом случае составной переход должен переводить машину состояний из одной допустимой конфигурации активных состояний в другую допустимую конфигурацию активных состояний, в противном случае модель синтаксически неправильна. Неформально говоря, это значит, что параллельное составное состояние нельзя покинуть и нельзя войти в него "частично" — в составном переходе должны участвовать по одному "представителю" (по одному вложенному состоянию) от каждой области параллельного составного состояния, участвующего в переходе.

Составной переход может иметь событие и сторожевое условие. Как всегда, переход срабатывает, если произошло событие и выполнено условие. При этом, для того чтобы сработал переход, имеющий несколько исходных состояний, необходимо, чтобы все они были активны при наступления события.

**ЗАМЕЧАНИЕ**

Составной переход — это переход по одному событию. В UML нет понятия перехода по нескольким одновременным событиям, т. к. нет понятия одновременных событий — события в машине состояний всегда возникают последовательно, одно за другим. Подразумевается, что даже если события возникают физически одновременно, имеется системный механизм (очередь событий или нечто подобное), который позволяет машине состояний выполнить переход по одному событию прежде чем начать обрабатывать следующее событие.

Приведем пример из информационной системы отдела кадров. Рассмотрим в качестве примера жизненный цикл сотрудника на предприятии — тот же пример, что был использован при описании последовательных простых и составных состояний (см. разд. 4.2), но с добавлением некоторых деталей.

Детали состоят в следующем. Состояние "работающий сотрудник" (на диаграммах обозначается `Employee`) — очевидно, составное — самое важное для системы и должно быть рассмотрено с наибольшей степенью подробности. Мы уже выделили некоторые вложенные состояния: сотрудник находится в офисе (`inOffice`), в командировке (`onAssignment`), болен (`isIll`), в отпуске (`onVacations`). Но параллельно с этим набором состояний, описывающим статус сотрудника в смысле присутствия на рабочем месте, сотрудник переживает и другие смены состояний, связанные с его статусом на предприятии. Пусть в нашей информационной системе отдела кадров будет три статуса: сотрудник проходит испытательный срок (`Trial`), сотрудник работает по контракту (по трудовому соглашению на определенный срок — `Contractor`) или постоянно (`Permanent`). Ясно, что смена этих состояний не зависит (формально) от смены состояний из первого списка и, т. о., образует параллельную машину состояний. На рис. 4.70 представлено составное состояние `Employee` (пока без внешних переходов), а внутренние переходы в каждой машине мы определили с помощью одной операции с параметром, указывающим состояние, в которое нужно перейти.[135]

[135] Это характерный прием проектирования. В операциях по изменению состояний нижнего уровня вложенности, как правило, оказывается больше общего, чем различий, поэтому целесообразно объединить их в одну операцию, учитывая различия с помощью параметров.

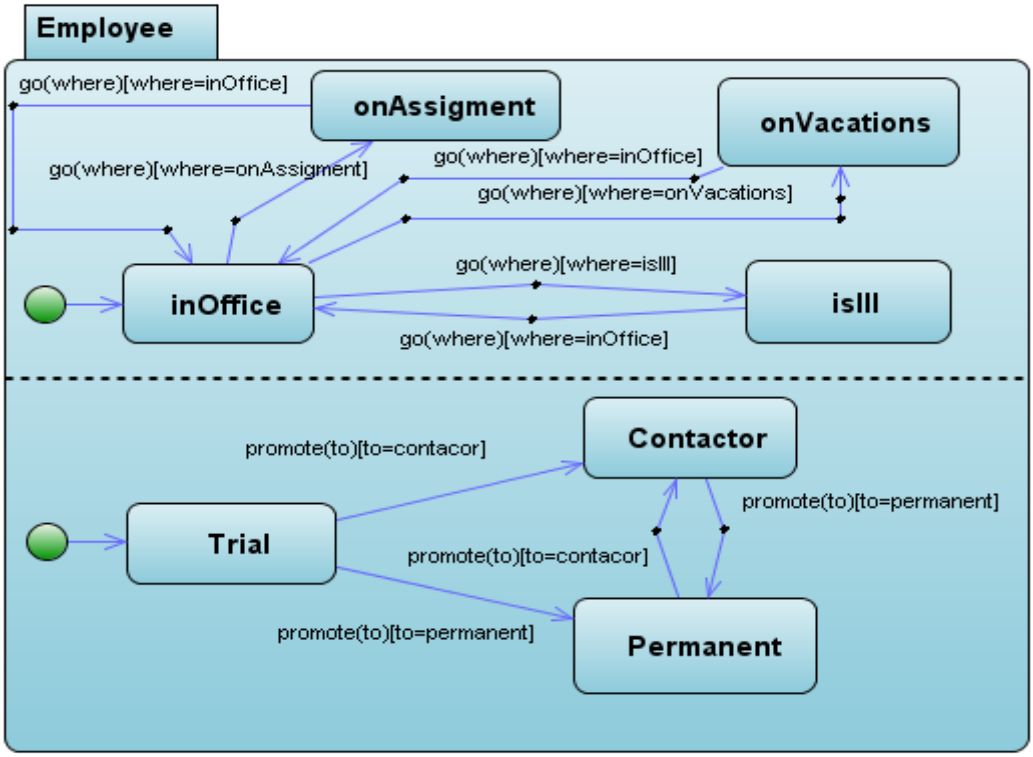



**Рис. 4.70. Параллельное составное состояние Employee**

Рассмотрим теперь переходы. При первом приеме на работу сотрудник должен пройти испытательный срок и в первый день работы должен быть в офисе. Поэтому переходы из соответствующих начальных состояний областей машины состояний ведут в состояния `inOffice` и `Trial`. Таким образом, внешний переход из состояния `Candidate` можно провести *в* состояние `Employee` как *простой* переход по событию `hire`. С другой стороны, переход из состояния `Retired`, очевидно, не должен приводить к новому испытательному сроку — допустим, что кадровая политика организации предусматривает повторный прием по контракту. В таком случае переход по событию `hire` из состояния `Retired` разумно определить как *составной* переход в состояния `inOffice` и `Contractor`. Аналогично, переход в случае увольнения должен происходить, во-первых, когда сотрудник находится в офисе (увольнять заочно не принято), и во-вторых, когда сотрудник занимает должность постоянно. Прекращение трудовых отношений с контрактником или проходящим испытательный срок обычно даже не называется увольнением. Это можно отразить с помощью соответствующего составного перехода из состояний `inOffice` и `Permanent`. Далее, переход в себя по событию `move`, видимо, не должен менять конфигурацию активных состояний, поэтому в параллельных вложенных машинах целесообразно заменить начальные состояния историческими. На рис. 4.71 приведена соответствующая диаграмма состояний. События на переходах во вложенных параллельных областях мы не стали указывать повторно, чтобы не загромождать диаграмму. Составные переходы изображаются с помощью специального значка, который выглядит как узкая закрашенная полоска (может быть расположен вертикально или горизонтально) и

называется *линейкой синхронизации*. Все сегменты составного перехода начинаются или заканчиваются на линейке синхронизации. В зависимости от того, сколько сегментов переходов начинается и заканчивается на линейке синхронизации, они получает специальное название (обозначение не меняется). Если на линейке начинается один сегмент и заканчивается несколько, то линейка синхронизации называется *соединением*. Если на линейке заканчивается один сегмент и начинается несколько, то линейка синхронизации называется *развилкой*. В данном случае на рис. 4.66 использованы одна развилка и одно соединение.

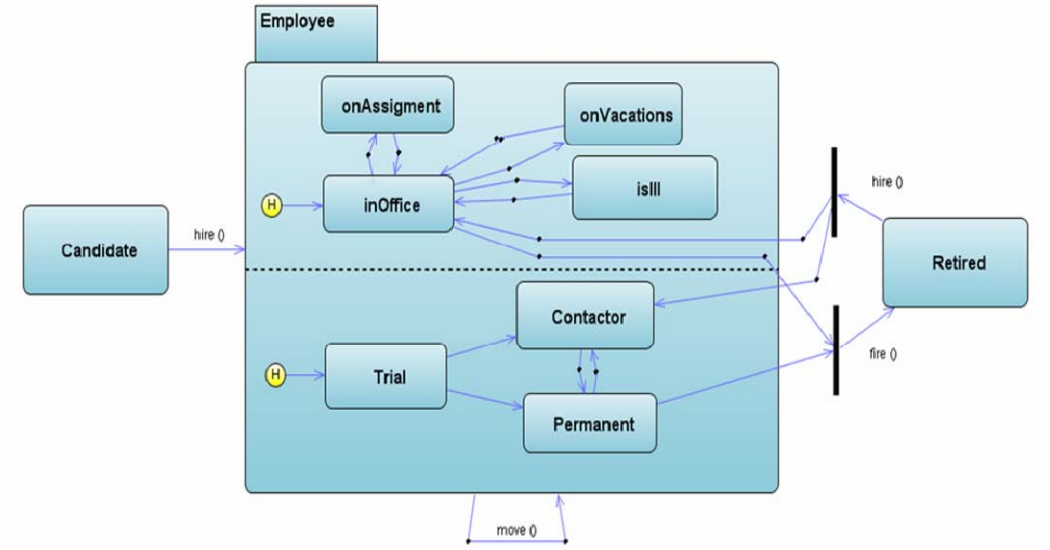



**Рис. 4.71. Составные переходы**

Поведение машин состояний в параллельных областях составного состояния `Employee` на диаграммах рис. 4.70 и 4.71 независимо — каждая их групп параллельных состояний "живет своей жизнью". Довольно часто возникает потребность синхронизировать работу параллельных вложенных машин состояний. Это можно сделать разными способами, например, использовать в одной машине состояний в сторожевых условиях на переходах проверку состояний другой машины. Но в UML есть специальное средство, называемое синхронизирующим состоянием, которое позволяет провести синхронизацию более наглядным и естественным образом.

*Синхронизирующее состояние* — это специальное состояние, используемое в диаграммах состояний в параллельных составных состояниях.

Синхронизирующее состояние имеет входные и выходные сегменты переходов (может быть по одному или по несколько). Входные сегменты синхронизирующего состояния должны начинаться в развилках одной из параллельных машин состояний (назовем ее исходной машиной состояний), а выходные сегменты должны заканчиваться в соединениях другой параллельной машины состояний (назовем ее целевой машиной состояний). Если срабатывает переход, проходящий через развилку в целевой машине состояний, то синхронизирующее состояние становится активным и, таким образом, может сработать переход, проходящий

через соединение в целевой машине состояний. Другими словами, семантика синхронизирующего состояния состоит в том, чтобы поставить срабатывание перехода в целевой машине в зависимость от срабатывания перехода в исходной машине.

Мы приведем здесь довольно замысловатый пример из информационной системы отдела кадров, чтобы продемонстрировать полезность применения синхронизирующих состояний. Допустим, что в организации принято следующее правило работы с молодыми специалистами: после окончания испытательного срока сотрудник направляется на тренинг, где проходит дополнительную профессиональную подготовку, после чего назначается на постоянную должность или же с ним заключается временный контракт. Находясь на тренинге, сотрудник, естественно, находится вне офиса (обозначим это состояние `onTraining`) и в то же время, пока что он "ни рыба, ни мясо" — испытательный срок уже закончен, а систематическая работа еще не началась (обозначим это состояние `Interlude`). На рис. 4.72 приведено соответствующим образом измененное параллельное составное состояние `Employee`. Чтобы не загромождать рисунок, мы опустили все переходы, которые не относятся к рассматриваемому в данном примере случаю.[136] Синхронизирующие состояния (их два в данном случае) изображены с виде небольших кружков на границе областей с числом внутри, смысл которого объясняется чуть ниже. При рассмотрении рис. 4.72 не следует забывать принятые соглашения: переход (в том числе и составной) управляется одним событием, в данном случае событие вместе со сторожевым условием мы поместили рядом с линейкой синхронизации составного перехода. На сегментах переходов, инцидентных синхронизирующим состояниям, никаких событий и сторожевых условий нет (и быть не может!).

[136] Мы хотим повторить еще раз свой совет: не следует рисовать перегруженных диаграмм. Вместо одной сложной диаграммы лучше нарисовать несколько простых, показывая на одной диаграмме одни детали, а на другой — другие. Инструмент обязан сохранить во внутреннем представлении модели *все* детали.

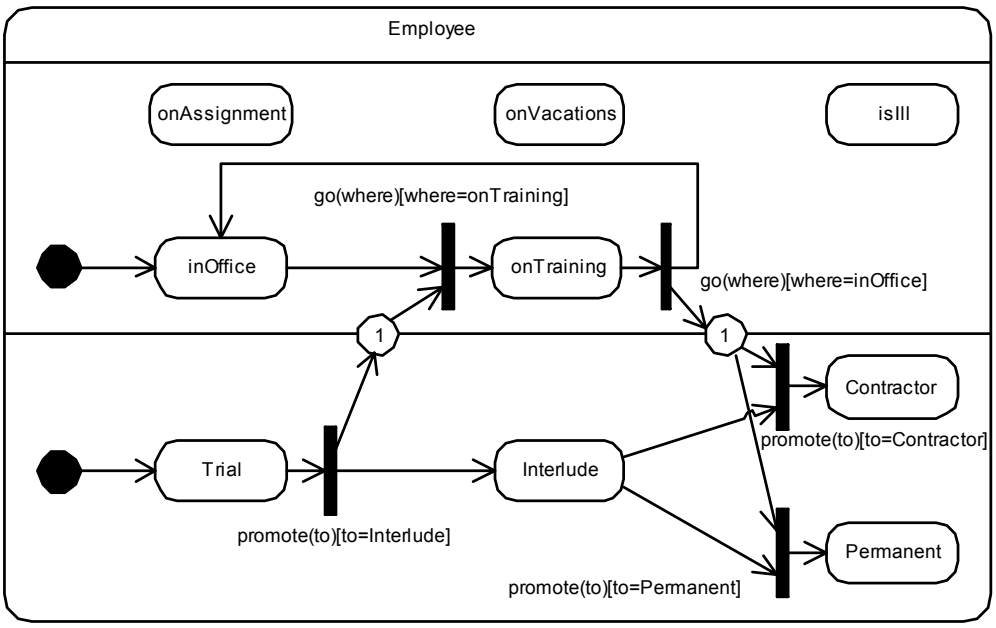



**Рис. 4.72. Синхронизирующие состояния**

Рассмотрим еще раз, как будет происходить изменение конфигурации активных состояний в диаграмме на рис. 4.72. Вначале активна пара состояний `inOffice` и `Trail`. (Может ли сотрудник, проходящий испытательный срок, заболеть, получить отпуск, отправиться в командировку — все это вопросы мы на данной диаграмме не рассматриваем). Когда испытательный срок заканчивается[137] (событие `promote` с аргументом `Interlude`) срабатывает составной переход и активными становятся первое синхронизирующее состояние (на диаграмме оно слева) и состояние `Interlude`. Теперь при наступлении события `go` с аргументом `onTraining` сработает составной переход в верней области и активным станет состояние `onTraining` (вместе с состоянием `Interlude`). Обратите внимание, что пока первое синхронизирующее состояние не активно (т. е. испытательный срок не закончен), все попытки отправить сотрудника на учебу не сработают. Далее изменение конфигурации активных состояний произойдет при наступлении события `go` с аргументом `inOffice` — свежеобученный сотрудник предстанет перед начальством для решения своей судьбы. При этом активными будут состояния `inOffice`, `Interlude` и второе синхронизирующее состояние. Как только будет принято решение (произойдет событие `promote`), сотрудник перейдет в состояние `Permanent` или `Contractor` в зависимости от значения аргумента данного события.

Нам осталось рассмотреть два важных обстоятельства, связанных с синхронизирующими состояниями. Во-первых, разъяснить, что означает число, которое указывается в синхронизирующем состоянии и, во-вторых, обсудить, какая информация может быть передана через синхронизирующее состояние из исходной машины состояний в целевую.

---

[137] Судя по нашему личному опыту работы с кадрами, испытательный срок лучше завершать не по времени (не по событию таймера), а по достижении определенного результата, например, по выполнении индивидуального задания.

Обычное состояние либо активно, либо нет. Можно, сказать, что величина активности обычного состояния является целым числом в интервале `0..1`. А величина активности синхронизирующего состояния может быть любым целым числом. Таким образом, синхронизирующее состояние на самом деле является счетчиком активности. При переходе *в* синхронизирующее состояние значение счетчика увеличивается на 1, при переходе *из* синхронизирующего состояния — уменьшается. Число, которое указывается в кружке синхронизирующего состояния на диаграмме — это максимальная величина счетчика. Можно указать символ `*`, что означает неограниченный счетчик. Если же указано конкретное число и величина счетчика превышает его, то возникает ошибка времени выполнения. При входе в параллельное составное состояние, которому принадлежит синхронизирующее состояние, величина счетчика сбрасывается в ноль. В терминах сетей Петри текущее значение счетчика уместно назвать маркировкой синхронизирующего состояния. В разобранном примере на рис. 4.72 использованы синхронизирующие состояния с пределом 1. Такие состояния называются *семафорами* — они либо разрешают переход в целевой машине, либо запрещают его.

Вообще говоря, синхронизирующие состояние может быть не просто счетчиком, а каналом для передачи данных между синхронизируемыми машинами состояний. В этом случае синхронизирующее состояние следует рассматривать как очередь объектов. Срабатывание перехода в исходной машине добавляет объект в конец очереди, срабатывание перехода в целевой машине изымает объект из начала очереди. Для изображения такого синхронизирующего состояния отлично подходит объект в состоянии (см. разд. 4.3.4).

Для иллюстрации использования неограниченных очередей и синхронизации с передачей информации мы рассмотрим один классический пример из области параллельного программирования. Пусть имеются два параллельных процесса — `Writer` и `Reader`. Процесс `Writer` асинхронным образом создает записи, а процесс `Reader` также асинхронно их обрабатывает в порядке создания. Каждый из процессов может по собственной инициативе завершить свою работу. На рис. 4.73 приведена соответствующая диаграмма состояний.[138]

[138] Как и многие другие диаграммы в книге, эта диаграмма нарисована вопреки возможностям инструмента.

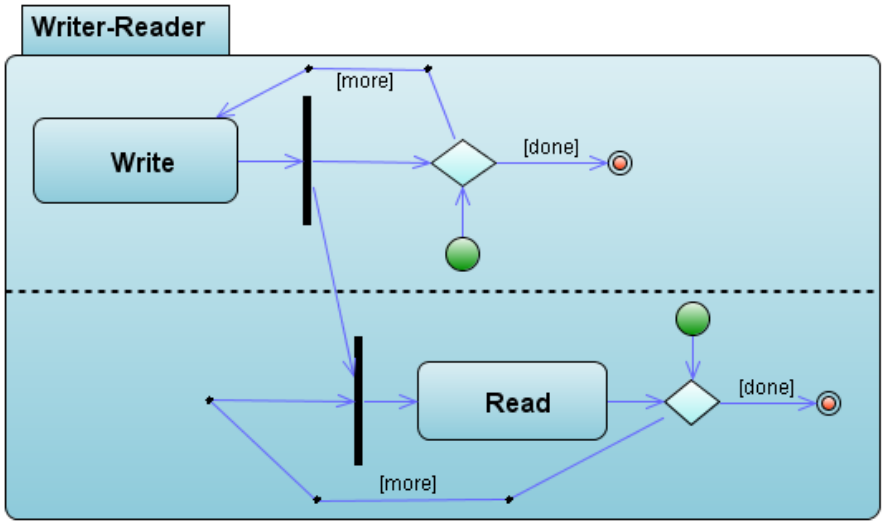



**Рис. 4.73. Параллельные процессы Читатель и Писатель**

### *4.5.3. Развилки, соединения и обусловленные потоки управления*

Диаграмма деятельности является частным случаем диаграммы состояний, но имеет различные дополнительные обозначения. Это правило распространяется и на моделирование параллелизма: основные средства моделирования параллельного поведения на диаграммах деятельности являются частным случаем рассмотренных в предыдущем разделе, плюс некоторые дополнительные особенности. Основными средствами являются развилки и слияния, а дополнительными — обусловленный поток управления и динамическая параллельность. Рассмотрим их все по порядку.

Развилки и слияния — это средства визуализации составных переходов (разд. 4.4.3). Диаграмма деятельности — это диаграмма состояний, в которой используются (в основном) переходы по завершении. Что же такое составной переход по завершении? В сущности, это очень простая и естественная конструкция, может быть даже более естественная, чем общий случай составного перехода. *Составной переход по завершении* на диаграмме деятельности имеет следующие особенности.

- Во-первых, никто не использует данный термин: все применяют названия частных случаев нотации: *развилка*, *соединение*, *линейка синхронизации*. Все эти случаи являются составными переходами по завершении.
- Во-вторых, семантика данных конструкций определяется в терминах потоков управления. В общем случае имеется линейка синхронизации, которой инцидентны входящие (один или более) и исходящие (один или более) сегменты переходов. Семантика состоит в следующем. Сначала все деятельности, инцидентные входящим сегментам, должны завершиться. После этого параллельно запускаются все деятельности, инцидентные исходящим сегментам.
- В третьих, развилки и соединения должны быть *сбалансированы*. Правило сбалансированности аналогично правилу вложенности на диаграммах

состояний.[139] Мы определим его следующим образом. Назовем диаграмму деятельности *строго сбалансированной*, если ее можно получить путем декомпозиции одной деятельности, причем на каждом шаге декомпозиции одна из деятельностей декомпозируется либо на некоторое количество последовательных деятельностей, либо на некоторое число параллельных деятельностей. Диаграмма деятельности является сбалансированной, если ее можно сделать строго сбалансированной путем введения промежуточных деятельностей и синхронизирующих состояний на имеющихся переходах. Неформально говоря, сбалансированность означает возможность выполнить блок-схему от начала к концу: потоки управления не должны возникать "ниоткуда" и исчезать "в никуда". На рис. 4.74 приведен пример сбалансированной (слева) и не сбалансированной (справа) диаграмм деятельности.

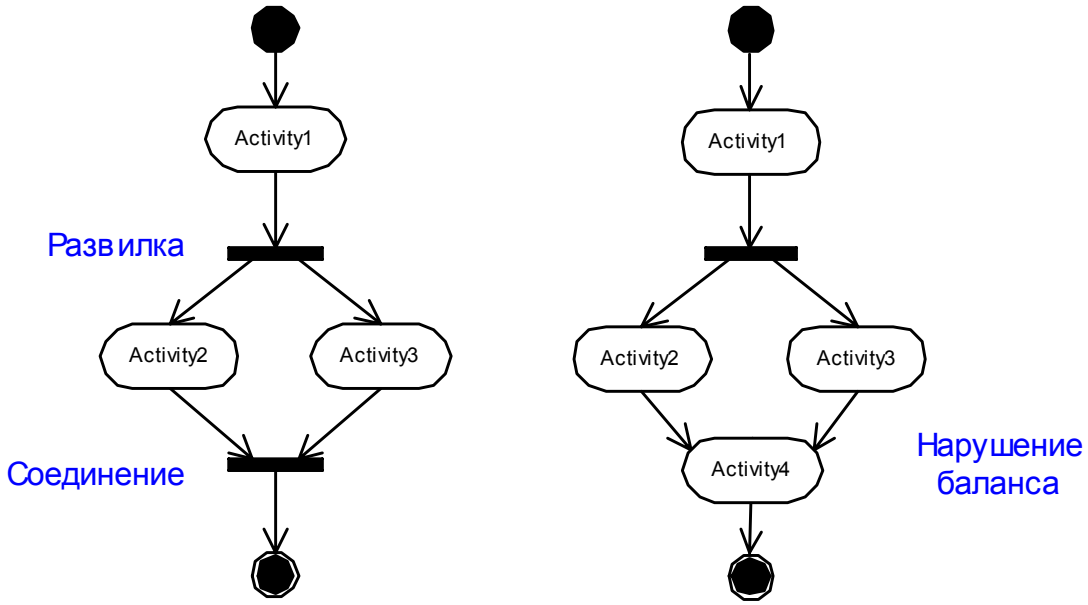



**Рис. 4.74 Баланс развилок и соединений**

**ЗАМЕЧАНИЕ**

Декомпозиция, о которой идет речь в правиле сбалансированности, не обязательно единственна.

Развилки и соединения настолько наглядное и естественное средство, что мы уже использовали их в примерах без особых пояснений. Приведем еще один пример из информационной системы отдела кадров. На рис. 4.75 приведена диаграмма деятельности, реализующая операцию перевода сотрудника на другую должность (операция `move`). В данном случае мы предполагаем, что при выполнении перевода новая должность должна быть вакантна. Если данное условие не выполнено, то операция не выполняется и посылается соответствующий сигнал (что является еще одним примером моделирования параллелизма на диаграммах деятельности, см. разд. 4.3.5). Далее заметим, что манипуляции со старой и с новой должностью могут проводиться параллельно, что отражено на диаграмме введением двух

[139] Напомним, что на диаграмме состояний составные состояния образуют дерево "И/ИЛИ".

потоков управления с помощью развилки. Работу с новой должностью можно в свою очередь распараллелить. Между потоками передается информация с помощью объекта в состоянии, который фактически является синхронизирующим состоянием для потоков управления. Когда все потоки управления завершены, выполнение операции успешно заканчивается, что моделируется с помощью соответствующего соединения.

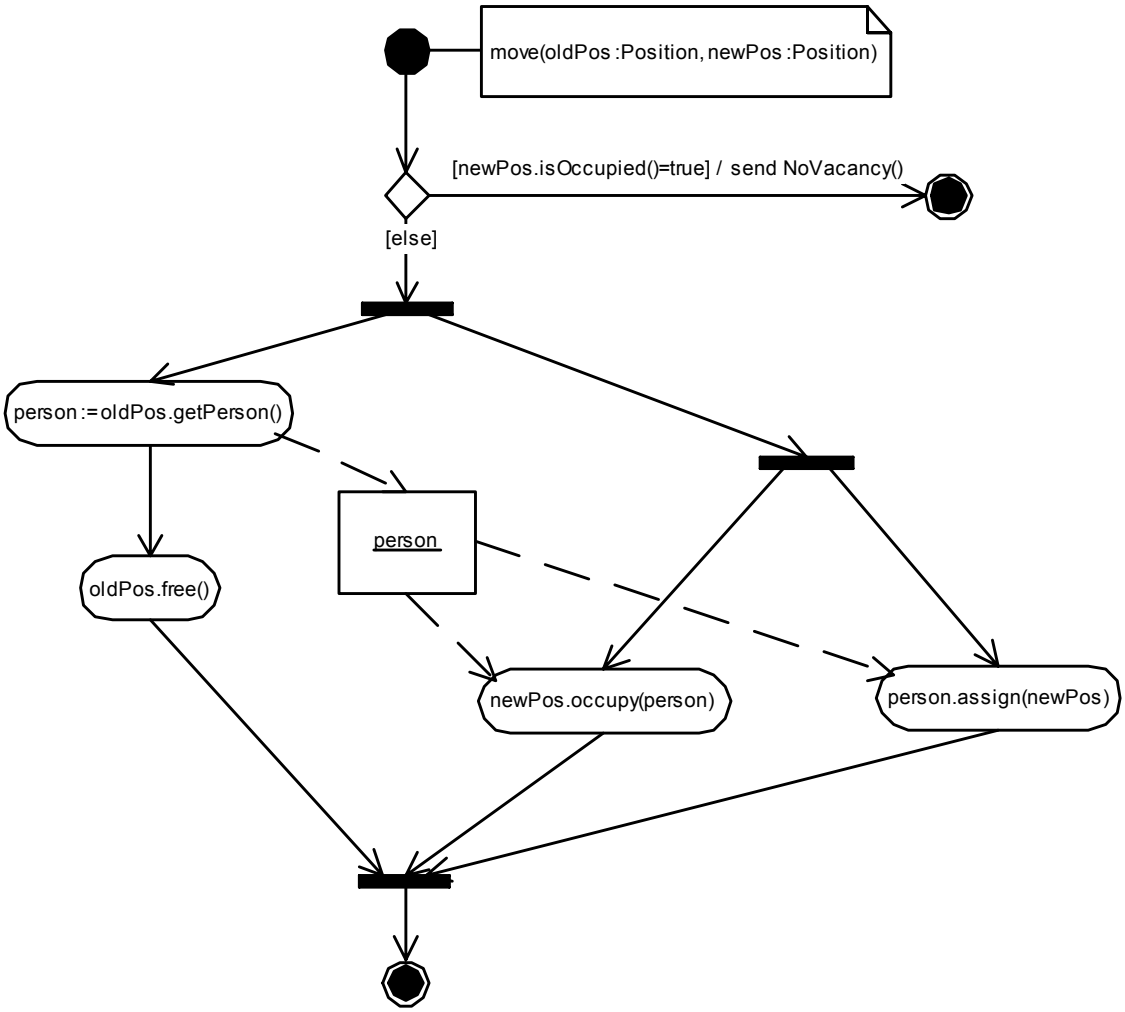



**Рис. 4.75. Диаграмма деятельности операции перевода сотрудника**

Допустим теперь, что операция `move` должна выполняться по другому: если целевая должность занята, то ее нужно предварительно освободить (т. е. отстранить от должности сотрудника, который ее занимает), а потом выполнить перевод. Задача очень похожа на предыдущую и можно воспользоваться известным рецептом математика из анекдота.[140] Но используем этот пример для демонстрации приема моделирования, который называется *обусловленным потоком управления*. Сегмент перехода, исходящий из развилки, может иметь сторожевое условие. В этом случае поток управления, начинаемый данным сегментом перехода, называется

[140] Математика спросили: как приготовить чай?. Он ответил, что нужно налить воды, вскипятить ее и заварить чай. А если уже есть кипяток? — Нужно вылить его и задача сведется к предыдущей.

обусловленным. Если сторожевое условие выполнено, то поток выполняется как обычно, в противном случае поток не выполняется — сразу считается завершенным. В диаграмме на рис. 4.76 этот прием применен при моделирования операции перевода: поток справа является обусловленным.

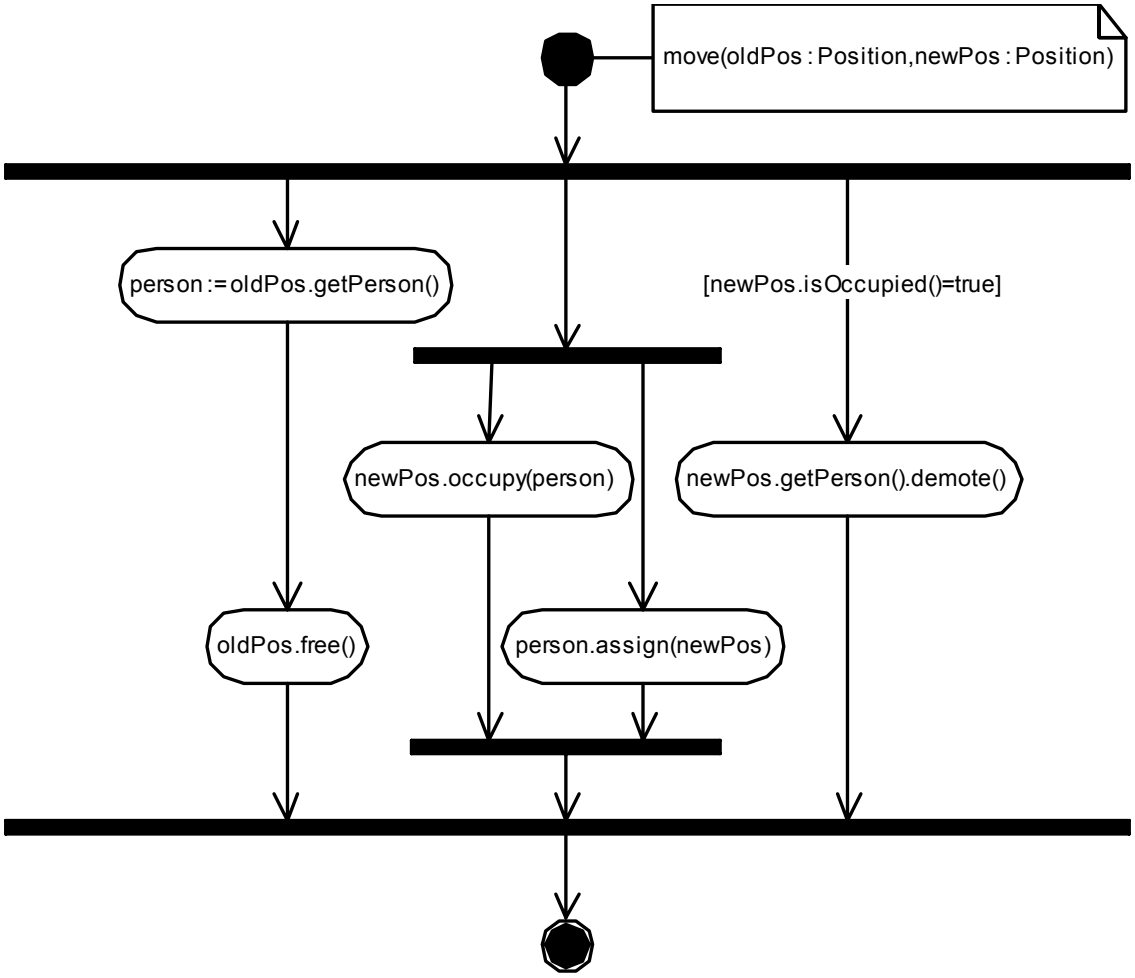



**Рис. 4.76. Обусловленный поток управления**

Обусловленный поток управления не является самостоятельным понятием в UML — это не более чем синтаксическое сокращение. На рис. 4.77 показано, как обусловленный поток управления (слева) эквивалентным образом выражается через ветвление (справа).

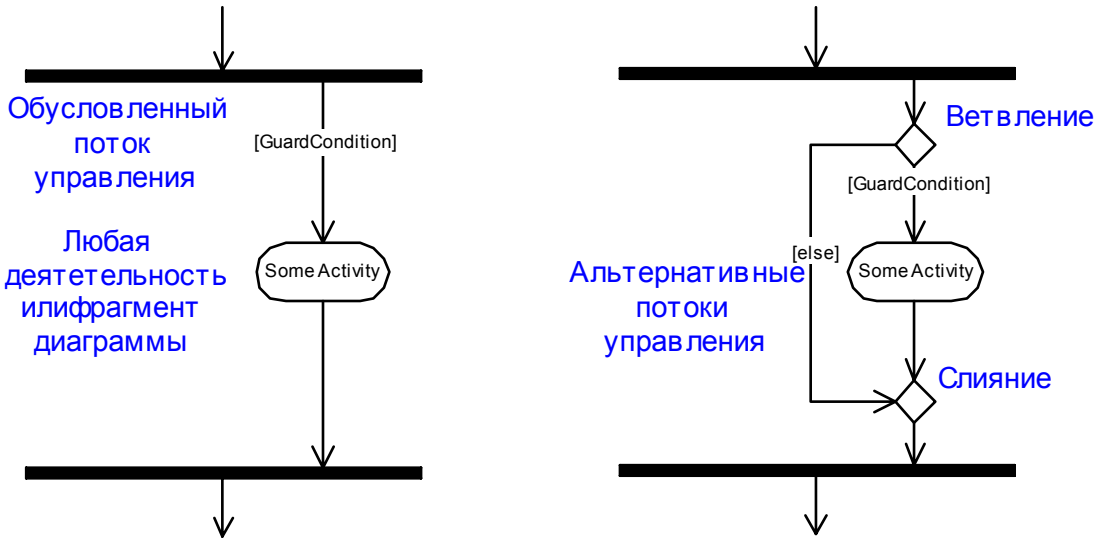



**Рис. 4.77. Выражение обусловленного потока управления через ветвление**

Подводя итоги раздела, отметим еще раз, что, по нашему мнению, средства моделирования параллелизма на диаграммах деятельности принадлежат к числу лучших в UML: они интуитивно понятны, наглядны и удобны в использовании.

### *4.5.4. Параллелизм на диаграммах взаимодействия*

Данный раздел получился короче остальных, что не удивительно, поскольку средства моделирования параллелизма на диаграммах взаимодействия довольно скудны.

На диаграммах кооперации графических средств моделирования параллелизма, фактически, не предусмотрено. Параллелизм указывается текстуально, с помощью номеров сообщений (см. разд. 4.4.3). В номера сообщений можно включать не только цифры, но и буквы. Сообщения, номера которые различаются в последней букве, считаются параллельными на данном уровне вызовов. Например, сообщения с номерами 3.1.1a и 3.1.1b считаются параллельными. Их относительный порядок не определен. Они могут быть отправлены в любом порядке или даже физически одновременно, если это допускает реализация. В то же время сообщения с номерами 3.1.1a и 3.1.1b *оба* предшествуют сообщению с номером 3.1.2 и *оба* следуют за сообщением с номером 3.1.

Другим средством моделирования параллелизма на диаграмме взаимодействия, родственным динамической параллельности (разд. 4.4.3) на диаграмме деятельности, является использование символа `||` в повторителе (разд. 4.4.3). Если символ повторителя имеет вид `*||`, то все сообщения, определяемые данным повторителем, считаются параллельными.

На диаграмме последовательности номера сообщений обычно не указываются. Как же отличить параллельные сообщения? Параллельные сообщения отличаются тем, что исходят из одной точки линии жизни (активации). Но внимательный читатель может заметить, что альтернативные сообщения тоже исходят из одной точки — чем же они отличаются от параллельных? К сожалению, хотя параллельность и альтернативность семантически сугубо разные вещи, графически на диаграмме

последовательности они не отличаются. Если исходящие сообщения помечены взаимоисключающими сторожевыми условиями, то это альтернативные сообщения, если сторожевых условий нет вовсе, то это явно параллельные сообщения. Если же сторожевые условия присутствуют, но не на всех сообщениях или же не являются взаимоисключающими (дизъюнктными), то, видимо, можно считать, что это нечто вроде обусловленных потоков управления.

Мы разделяем недоумение и недовольство читателя, которое могло возникнуть в этом месте при чтении наших объяснений. Что делать — моделирование параллелизма на диаграммах взаимодействия — не самая сильная сторона UML. Наше недовольство этой частью языка заходит так далеко, что мы не приводим примеров в этом разделе.

### *4.5.5. Активные классы*

Последнее[141] средство моделирования параллелизма — это *активные классы*. Семантически активный класс — это класс, экземплярами которого являются активные объекты. *Активный объект* — это объект, имеющий собственный поток управления. Точнее говоря, активный объект — это и *есть* поток управления. Создание активного объекта означает создание нового экземпляра стека вызова процедур и начало деятельности (состоящей в вызове операций, посылке сообщений и т. п.). Активному объекту противопоставляется пассивный объект. *Пассивный объект* — это объект класса, который *не* является активным. Пассивный объект ничего не может сделать "по собственной инициативе" — все его деятельность инициируется вызовом извне одной из операций. Вызов данной операции может произойти в процессе выполнения некоторой операции другого пассивного объекта, которая была вызвана из третьего объекта и т. д., но где-то в этой цепочке вызовов есть начало: оно то и называется активным объектом. Потоки управления активных объектов параллельны: поток управления одного активного объекта не является частью потока управления другого активного объекта. Активный объект может быть *создан* в другом потоке управления, но будучи созданным, он начинает свою собственную жизнь.

Семантика активных классов и объектов в UML определена не очень строго: оставлено вполне достаточно свободы, чтобы эти понятия можно было адекватным образом трактовать в окружении любой операционной системы. Положение здесь совершенно аналогично с не до конца определенной семантикой действий. Авторы языка не хотели попадать в плен конкретной операционной системы так же, как они не хотели попадать в плен конкретного языка программирования.[142]

Синтаксически активный класс в UML — это класс со стереотипом «`process`» или «`tread`». С точки зрения семантики UML эти стереотипы ничем не отличаются. С точки зрения прагматики два стереотипа введены для того, чтобы создатель модели

---

[141] Последнее по порядку нашего рассмотрения, но не по степени важности.

[142] Заметим, что ситуация здесь все-таки не совсем аналогичная. Автор хорошо помнит те времена, когда для *каждой* модели компьютера разрабатывалась *своя* операционная система. С тех пор положение кардинально изменилось: процесс унификации операционных систем идет быстрыми темпами, в отличие от языков программирования (см. врезку "Универсальный язык программирования" в разд. 5.3.1).

мог акцентировать внимание разработчика на необходимости выбора того или иного средства организации потока управления, доступного в конкретной операционной системе. Стереотип «process» следует применять, если подразумевается "тяжелый", ресурсоемкий поток управления, «tread» — если подразумевается "облегченный" поток управления.[143] На диаграмме активный класс выделяется тем, что граница прямоугольника нарисована жирной линией. Аналогичным образом выделяется прямоугольник активного объекта. Если этого не достаточно, можно использовать стандартное свойство {active} после имени объекта.

Стиль изображения активных объектов и классов на диаграммах еще не устоялся. Например, авторы языка рекомендуют рисовать кооперацию пассивных объектов *внутри* фигуры активного объекта, инициирующего данную кооперацию. Однако далеко не все инструменты позволяют это сделать.

## 4.6. Выводы

- Конечные автоматы — базовая алгоритмическая техника моделирования поведения
- Диаграмма состояний — вариант конечного автомата с различными дополнениями
- Диаграмма деятельности (= блок-схема) — частный случай диаграммы состояний с различными дополнениями
- Диаграммы кооперации и последовательности (диаграммы взаимодействия) семантически эквиваленты и являются протоколами выполнения (примерами)
- Параллелизм на уровне программы моделируется с помощью взаимодействующих машин состояний
- Параллелизм на уровне операционной системы моделируется активными объектами

[143] Разумеется, названия стереотипов «process» и «tread» выбраны в расчете на соответствующие ассоциации у разработчиков. Но «process» и «tread» в модели совсем не означают "процесс" или "нить" в терминах конкретной (пусть даже очень известной) операционной системы. Как именно следует трактовать эти стереотипы в конкретном окружении должны решать разработчики и производители инструментов.

# Тема 5. Дисциплина моделирования

- Как следует структурировать модель?
- Что такое образец проектирования?
- В каких случаях целесообразно применять образцы и каркасы?
- Как влияет применение UML на процесс разработки приложения?
- Какие элементы UML являются ключевыми в практическом моделировании?

## 5.1. Управление моделями

Предметом этой небольшой главы является прежде всего прагматика UML, т. е. обсуждение вопросов применения UML. Мы полагаем, что материал трех предыдущих глав в достаточной степени раскрывает детали синтаксиса и семантики языка и считаем необходимым вернуться в конце книги к тому, с чего мы начали — к обсуждению прагматики.

Мы намерены затронуть три темы, соответствующие трем разделам главы.

- Первая тема связана с управлением моделями, т. е. с теми механизмами, которые предназначены для успешного преодоления количественных трудностей, возникающих при практическом моделировании, связанных с объемом и сложностью моделей.
- Второй рассматриваемый вопрос связан с оценкой влияния применения UML на процесс разработки приложений. Мы полагаем, что применение UML положительно влияет на разработку и предлагаем собственную гипотезу для объяснения этого феномена.
- В третьем разделе предпринимается попытка ранжировать элементы моделирования UML с точки зрения их важности при практическом моделировании. Мы предполагаем выделить те элементы, без которых никак нельзя обойтись (при определенных начальных условиях) и противопоставить их тем элементам, использование которых, по нашему мнению, не всегда является обязательным при моделировании.

### *5.1.1. Пакетная структура*

Проблема управления моделями реально возникает в том случае, когда модель достаточно велика. Что значит "достаточно велика"? Если вы в состоянии нарисовать всю модель "за один присест", не откладывая уточнения "на завтра", то эта модель, по нашему мнению, недостаточно велика (для вас). Если вы в состоянии просмотреть представленную модель и сразу понять все, что имел в виду автор, не возвращаясь назад для повторного изучения и не задавая вопросов, то такая модель также недостаточно велика для вас. Это, разумеется, достаточно субъективные критерии. Тем не менее, судя по нашему опыту, для подавляющего большинства разработчиков такая модель, как модель информационной системы отдела кадров, рассматриваемая в этой книге, уже достаточно велика и нуждается в управлении. Заметим, что наш пример все-таки достаточно прост — большинство практических приложений сложнее нашей информационной системы отдела кадров, поскольку требуют рассмотрения различных деталей и частных случаев, которые опущены в нашем учебном примере.

Итак, что делать, если модель достаточно велика? Ответ очевиден: ее нужно разделить на части обозримого размера и рассматривать их по отдельности. Для этой цели в UML используется одно универсальное средство — пакет.

| *Пакет* — это группирующая сущность в UML. |
|---|

Хотя в UML предусмотрена только одна сущность для структурирования модели, этого оказывается достаточно. Рассмотрим свойства пакета в UML.

- *Отношение владения*. Говорят, что пакет владеет объявленными в нем элементами модели, а элементы принадлежат владеющему ими пакету. Пакет может владеть любыми элементами модели, в частности, пакетами. Отношение владения является строгой композицией, т. е. каждый элемент модели принадлежит ровно одному пакету. Всегда имеется корневой пакет (с именем по умолчанию). Таким образом, структура пакетов по отношению владения (или вложенности) образует в модели строгую иерархию, подобную иерархии папок и файлов в файловой системе.
- *Пространство имен*. Каждый пакет в модели образует пространство имен, т. е. область определения и использования имен. Рассмотрим понятие пространства имен в UML чуть подробнее. Имена однотипных элементов в одном пространстве имен уникальны — элемент данного типа однозначно определяется по имени в данном пространстве имен — в этом и состоит основное назначение пространства имен. В другом пространстве имен это имя может быть использовано для именования любого другого элемента. Например, все классы в данном пакете должны иметь различные имена, в то же время ассоциация (будучи сущностью иного типа) может иметь имя, совпадающее с именем класса. Пакет — не единственное средство образования пространства имен в UML. Все сущности, имеющие составляющие, которые не существуют отдельно от этих сущностей, образуют собственное пространство имен. Таковыми сущностями являются, в частности, классификаторы, ассоциации, машины состояний и кооперации. Например, атрибуты в разных классах могут иметь совпадающие имена, поскольку эти атрибуты принадлежат *различным* пространствам имен. Пространства имен вложены друг в друга. Пространства имен пакетов вложены в соответствии с иерархией отношения владения, пространства имен прочих элементов вложены в соответствии со специфической иерархией элементов. В частности, каждое составное состояние машины состояний образует вложенное пространство имен. Таким образом, пространства имен также образуют строгую иерархию. Для указания точного положения любого элемента в этой иерархии используются *составные имена* — последовательность имен вложенных пространств имен, разделенных двойным двоеточием (`::`), которая заканчивается именем элемента.
- *Видимость элементов*. Все элементы, которыми владеет пакет, обладают определенной видимостью. Закрытые (ключевое слово `private`, символ `-`) элементы видимы только в том пакете, где определены. Защищенные (ключевое слово `protected`, символ `#`) элементы видимы в том пакете, где определены, и во всех пакетах, которые обобщает данный (см. ниже про отношение обобщения для пакетов). Только открытые (ключевое слово `public`, символ `+`) элементы данного пакета могут быть видимы в иных пакетах.

На диаграммах пакет изображается в виде фигуры — прямоугольник с закладкой. Если внутри пакета ничего не изображено, то имя пакета пишется в основном прямоугольнике, в противном случае — в закладке. Внутри основного прямоугольника фигуры пакета можно помещать любые элементы модели — тем самым моделируется отношение владения: пакет владеет элементом, помещенным внутрь его фигуры. Однако, хотя всякий элемент модели, как мы только что отметили, обязательно принадлежит определенному пакету, на всех наших предыдущих диаграммах пакеты вообще не изображались. Значит ли это, что все наши предыдущие диаграммы синтаксически неправильны? Разумеется, нет. Отношение владения редко изображают графически — обычно его задают другими средствами. Большинство инструментов в явном виде предусматривает специальный интерфейс для задания, отображения и манипулирования отношением владения. Обычно это древовидная структура, изображающая дерево вложенности пакетов и элементов. Как правило, этот интерфейс позволяет создавать, перемещать из в пакета в пакет и удалять любые элементы модели, в том числе и пакеты.

Здесь необходимо отметить одну важную тонкость. Диаграммы также находятся в отношении владения с пакетами: каждой диаграммой владеет некоторый пакет. Однако диаграмма *не* образуют узла в дереве вложенности пакетов: диаграмма это *не* пакет, она *не* определяет пространства имен. Если поместить на диаграмму новый элемент, то этим элементом будет владеть непосредственно тот пакет, который владеет диаграммой, и имя элемента попадет в пространство имен данного пакета.[144]

Рассмотрим теперь, для чего же нужно строить иерархию пакетов и пространств имен при моделировании? Для начала повторим уже сказанное: для маленьких моделей этого делать *не* нужно. Но маленькие модели встречаются только в учебниках по объектно-ориентированному моделированию — в жизни модели большие или очень большие. Такие модели нуждаются в структуре, в противном случае они не отвечают своему основному назначению — служить средством визуализации, спецификации, проектирования и документирования. Если диаграмма в модели не помещается на экран так, чтобы тексты и значки оставались читаемыми, от модели мало проку. Если имена элементов модели состоят из десятков непонятных слов, модель трудно использовать. Если для понимания какой-то характеристики приложения нужно одновременно рассматривать десяток диаграмм, модель не отвечает своему назначению. Если для поиска ответа на конкретный вопрос нужно просмотреть всю модель подряд, она никуда не годится.

Мы полагаем, что модель имеет хорошую, удобную для работы структуру, если выполняются следующие три количественных критерия.

---

[144] Мы настойчиво советуем, особенно на первых порах, определять новые элементы в модели с помощью интерфейса для управления вложенными пакетами и пространствами имен, а не рисовать фигуру элемента на диаграмме. Это поможет определять элементы именно в нужном пространстве имен.

- Количество сущностей,[145] изображенных на всех диаграммах, примерно одинаково и равно 7±3. Если сущностей на диаграмме три или меньше, то диаграмма недостаточно информативна, чтобы включать ее в модель отдельно. Такую диаграмму либо не стоит включать в модель, либо можно объединить с другой однородной диаграммой. Если на диаграмме больше десяти сущностей, то она, как правило, слишком трудна для понимания с одного взгляда. Такую диаграмму целесообразно разбить на несколько диаграмм.
- Ширина ветвления в дереве вложенности пакетов с учетом диаграмм примерно постоянна и равна 7±3. Другими словами, пакету должны принадлежать от трех до десяти пакетов или диаграмм. Если их меньше, то не понятно, нужен ли такой малосодержательный пакет как отдельная сущность. Если больше, то в деталях пакета можно "утонуть".
- Число использующих вхождений элементов модели должно быть ограничено сверху значением 7±3, при этом больше половины элементов модели должно присутствовать на диаграммах. Любой данный элемент модели может присутствовать на некотором количестве диаграмм: 0, 1 или больше. Если на диаграммах нарисовано меньше половины существующих в модели элементов, то это не модель, а "вещь в себе", о назначении которой трудно догадаться. Если один и тот же элемент повторяется в разных местах больше десяти раз, то это "масло масляное".

Приведенные критерии характеризуют *форму* модели, но никак не затрагивают ее содержание. Структурировать модель по содержанию можно самими разными способами, мы перечислим три наиболее характерных.

- *По структуре приложения*. В основу структуры пакетов кладется одна из реальных структур приложения, например, компонентная структура. Вся система разбивается на пакеты, соответствующие компонентам (или подсистемам), они, если нужно, разбиваются на более мелкие части и т. д.
- *По фазам процесса разработки*. Модель делится на пакеты в соответствии с фазами ее создания (набор которых зависит от принятой дисциплины моделирования). Например, пакет для диаграмм концептуального уровня, пакет диаграмм детального проектирования, пакет диаграмм реализации и развертывания и т. д.
- *По представлениям*. Модель делится на пакеты по представлениям. Например, представление использования, логическое представление, представление реализации и т. д.

На рис. 5.1 представлена структура пакетов информационной системы отдела кадров, построенная средствами использованного нами инструмента (Visio 2000). В данном случае (в демонстрационных целях) мы определили сразу три структуры: по компонентам, по фазам и по представлениям. Разумеется, в реальной работе создавать сразу несколько альтернативных структур нет необходимости — достаточно одной. На рис. 5.1 в окне UML Navigator хорошо видна иерархия отношений владения, связывающих пакеты и диаграммы. Часть структуры пакетов и диаграмм раскрыта полностью (`Components`), другая часть не раскрыта (не

[145] Отношения на диаграмме мы *не* считаем, хотя отдаем себе отчет в том, что такой взгляд весьма спорен.

хватило места на экране). Справа видна диаграмма классов, на которой отражена общая структура пакетов для первого варианта.

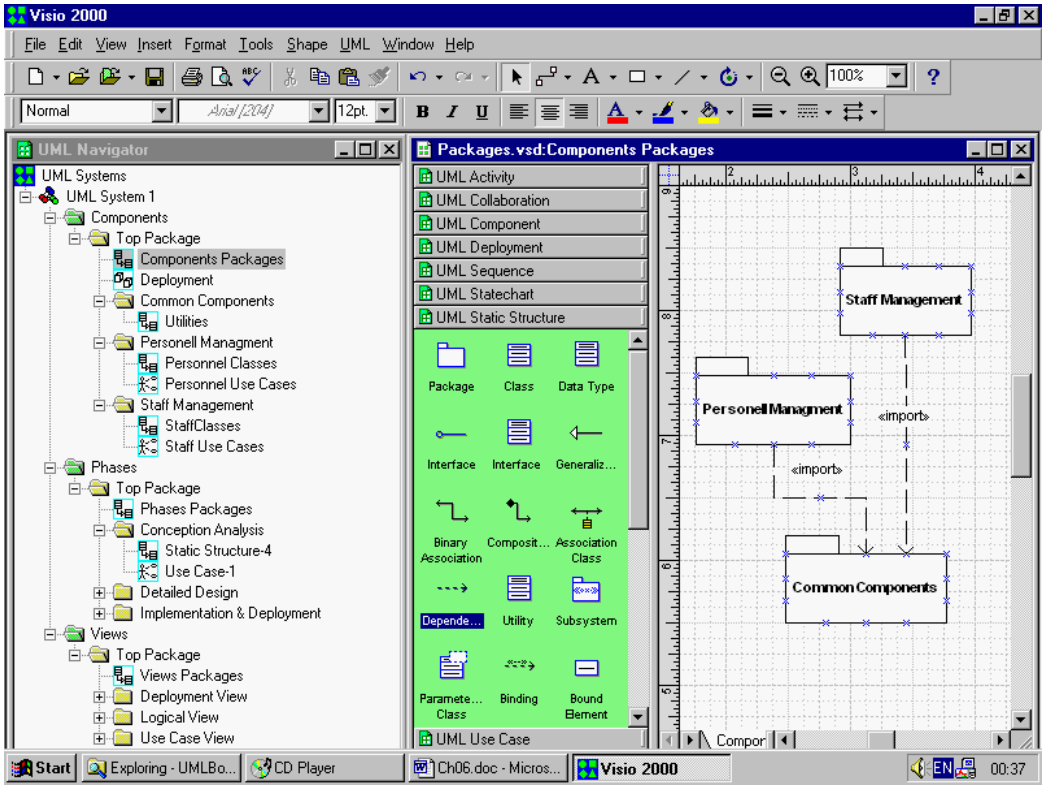



**Рис. 5.1. Структура пакетов в модели. Visio 2000.**

То важное обстоятельство, что модель — это одно, а совокупность диаграмм — это совсем другое хорошо видно на рис. 5.2. В рассматриваемом инструменте (Sun Java Studio Enterprise) модель и диаграммы сразу, на самом верхнем уровне, отделены друг от друга и хранятся в одтельных папках.

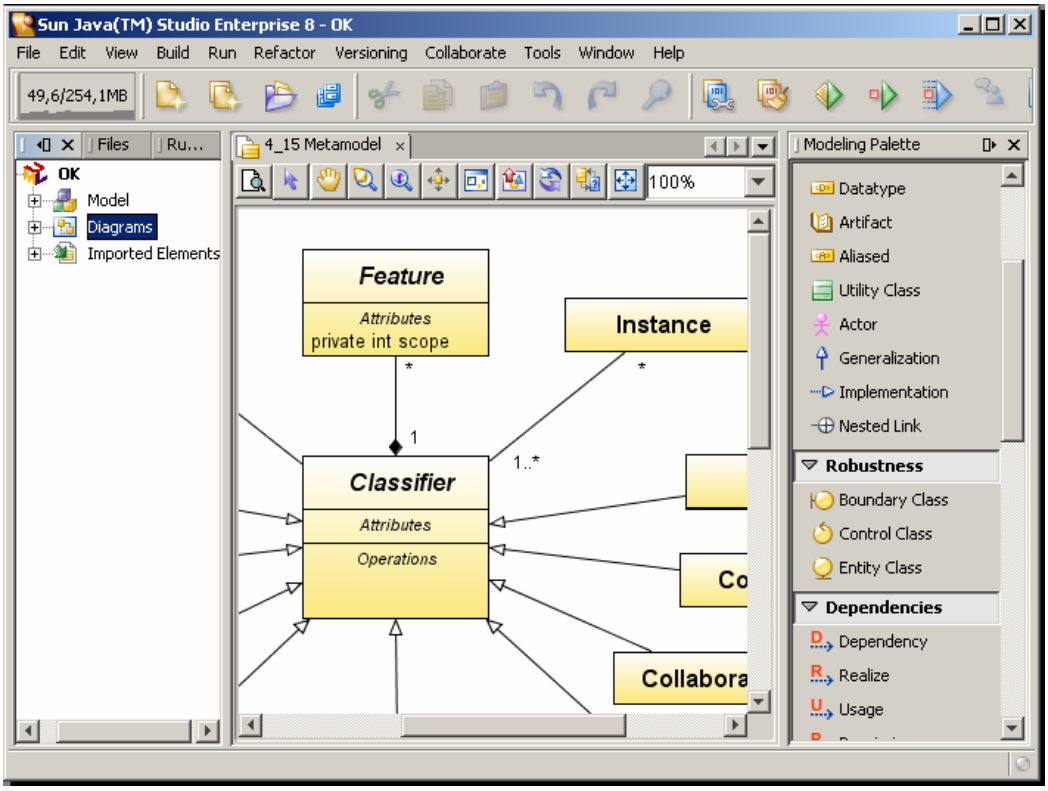



**Рис. 5.2. Структура пакетов в модели. Sun Java Studio Enterprise 8.**

Авторы языка рекомендуют структурировать модель по представлениям. Вслед за ними разработчики большинства инструментов пытаются *навязать* пользователю структуру модели. По нашему мнению, только в сравнительно простых моделях можно обойтись каким-то одним принципом структуризации. В более сложных моделях приходится определять достаточно много пакетов (особенно если придерживаться наших критериев локального ограничения сложности). Поэтому исходить нужно из потребностей модели, руководствуясь опытом и здравым смыслом — указанные принципы структуризации полезны, но все равно требуют размышлений при применении. Ситуация сходна с программированием: хорошая структура программы, также как и модели, дается не даром.

### *5.1.2. Отношения между пакетами*

Итак, допустим, что выбрана принципиальная структура пакетов в модели (и фигуры пакетов отображены на диаграмме, описывающей общую структуру модели). Необходимо обсудить вопрос о том, в каких отношениях могут находиться пакеты и как эти отношения отображаются на диаграмме. Мы рассмотрим отношения между пакетами в следующей последовательности:

- отношения владения (вложенности);
- индуцированные отношения;
- стереотипные зависимости;
- обобщение.

*Вложенность пакетов* влечет вложенность пространств имен. Всякий элемент модели, определенный в данном пространстве имен, видит все элементы,

определенные в этом же пространстве имен и во всех объемлющих пространствах имен. Имена в "параллельных" и вложенных пространствах имен не видимы для данного элемента. Если в цепочке вложенных пространств имен определены несколько разных элементов с одним именем, то для данного элемента видимым является ближайший при просмотре изнутри наружу. Значение свойства видимости самих элементов (`public`, `protected`, `private`), в том числе пакетов, не имеет значения для данного правила.

Например, на рис. 5.3 классы `A` и `B` принадлежат пакету `X`, класс `D` принадлежит пакету `Y`, а класс `C` и пакеты `X` и `Y` принадлежат некоторому объемлющему пакету (т. к. они определены на одной диаграмме). В этом случае класс `A` видит классы `B` и `C`, но не видит класс `D`; класс `B` видит классы `A` и `C`, но не видит класс `D`; класс `C` не видит классы `A`, `B` и `D`; класс `D` видит класс `C`, но не видит классы `A` и `B`.

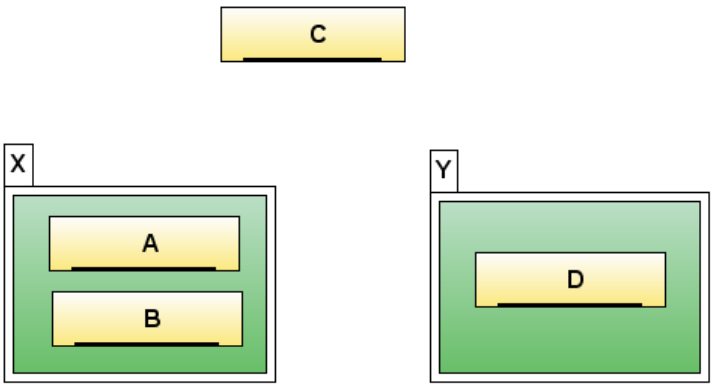



**Рис. 5.3. Вложенность пакетов**

*Индуцированное отношение* — это отношение между пакетами, которое просто отражает наличие такого же отношения между некоторыми элементами данных пакетов.

Например, если указано, что пакет `X` зависит от пакета `Y`, то это означает, что один или несколько элементов модели, которыми владеет пакет `X`, зависят от одного или нескольких элементов модели, которыми владеет пакет `Y`. При этом вовсе не подразумевается, что *все* элементы пакетов находятся в данном отношении — достаточно наличия одной пары.

Например, на рис. 5.4 показано, что операции некоторых классов, которыми владеют пакеты `Personnel Management` и `Staff Management` вызывают некоторые операции классов, которыми владеет пакет `Common Components`. В данном случае мы используем естественное деление информационной системы отдела кадров на пакеты по компонентам.

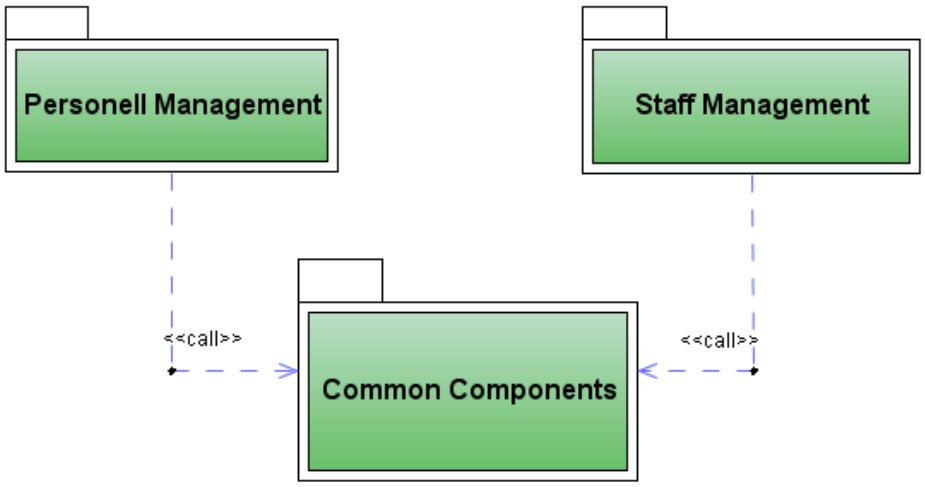



**Рис. 5.4. Индуцированные зависимости**

Существуют два специальных стандартных стереотипа отношения зависимости — «access» и «import», которые имеют сходное назначение, но различаются в некоторых деталях. В обоих случаях это отношение зависимости между пакетами, которое указывает на то, что зависимый пакет имеет доступ к открытым элементам независимого пакета (т. е. зависимый пакет "видит" открытое содержимое независимого пакета).

На рис. 5.5 представлен пример, аналогичный тому, что рассмотрен на рис. 5.2 (полезно сопоставить эти примеры). Для участвующих классов указаны значения свойства видимости: классы `A` и `D` — открытые, `B` и `C` — закрытые. В этом случае класс `A` видит классы `B` и `C`, но не видит класс `D`; класс `B` видит классы `A` и `C`, но не видит класс `D`; класс `C` видит класс `A`, но не видит классы `B` и `D`; класс `D` видит классы `A` и `C`, но не видит класс `B`.

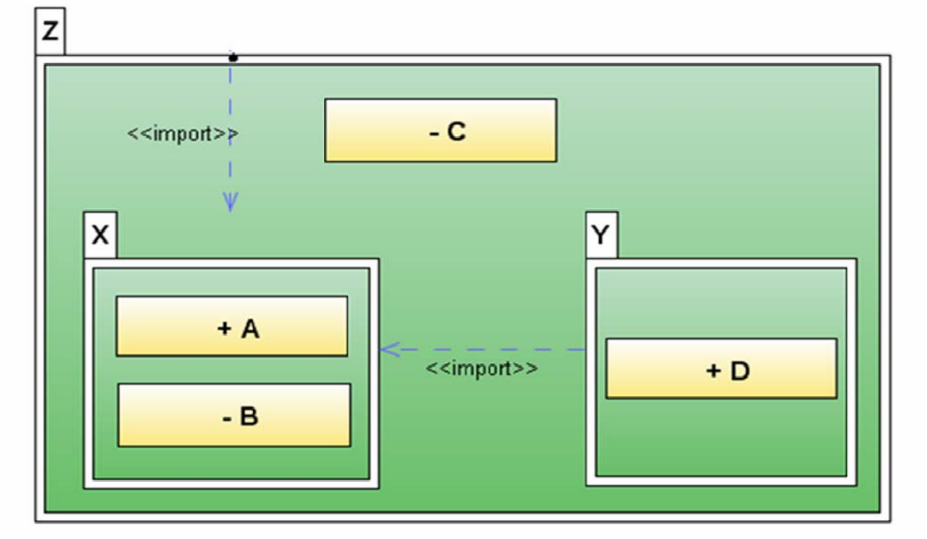



**Рис. 5.5. Зависимость со стереотипом «import». Расширение пространства имен.**

В отношении определения видимости семантика зависимостей «access» и «import» одинакова. Различие между ними заключается в том, что использование зависимости «access» не влияет на пространство имен зависимого пакета — для доступа к элементу независимого пакета нужно указывать составное имя (с именами вложенных пакетов разделенными ::), а при использовании зависимости «import» происходит расширение пространства имен зависимого пакета пространством имен независимого пакета (можно использовать простые имена, без ::). При этом накладывается ограничение: имена в этих пространствах имен не должны совпадать, в противном случае возникает конфликт имен и модель считается противоречивой.

**ЗАМЕЧАНИЕ**

> Отношения зависимости со стереотипами «access» и «import» не являются транзитивными. Если пакет X имеет доступ к пакету Y, а пакет Y имеет доступ к пакету Z, то из этого не следует, что пакет X имеет доступ к пакету Z. В то же время, если пакет X вложен в пакет Y, а пакет Y имеет доступ к пакету Z, то из этого следует, что пакет X имеет доступ к пакету Z.

Хотя пакеты не являются классификаторами, тем не менее для них можно указать *отношение обобщения*.[146] Обычно отношение обобщения для пакетов применяется в следующей ситуации. Допустим, что имеется несколько однородных в некотором смысле пакетов, например, несколько альтернативных вариантов реализации одной и той же функциональности. В таком случае можно определить в модели *абстрактный пакет*, который обобщает конкретные варианты. Например, на рис. 5.5. представлена одна из возможных структур пакетов информационной системы отдела кадров, касающаяся управления данными. Пакеты Personnel Management и Staff Management зависят от абстрактного пакета Data Management, который является обобщением пакетов Database Management и Files Management.

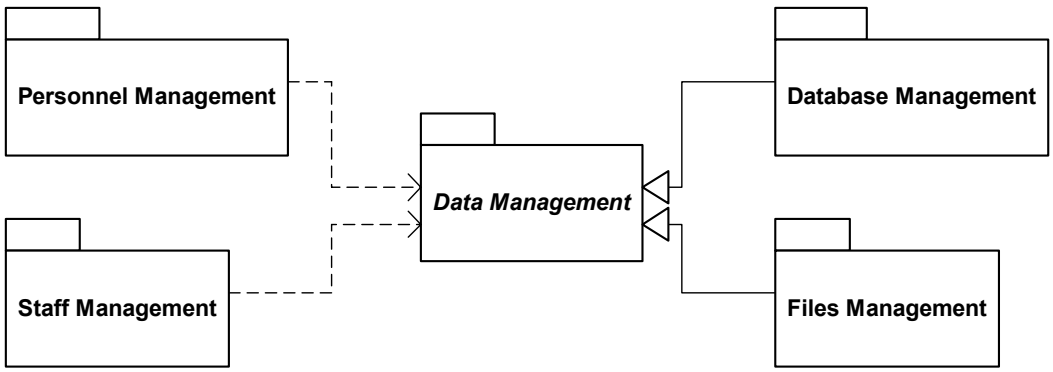



**Рис. 5.6. Отношение обобщения для пакетов**

### *5.1.3. Модели, системы и подсистемы*

В UML определены несколько понятий, которые имеют особое для значение при определении как структуры модели, так и структуры моделируемого приложения.

[146] Оба утверждения в этом предложении принадлежат, по нашему мнению, к числу сомнительных решений в UML.

Хотя назначение этих понятий совершенно ясно, они описаны в стандарте не очень отчетливо, к тому же заметно менялись от версии к версии UML, а потому в конкретных инструментах возникают различные разночтения, связанные с ними.

Мы имеем в виду понятия модели, системы и подсистемы, а также синтаксические средства их выражения в UML, в том числе стандартные стереотипы. Здесь мы описываем суть данных понятий, их определения в контексте стандарта UML и указываем некоторые варианты, наиболее часто встречающиеся в практических инструментах моделирования.

Самым общим из обсуждаемых понятий является понятие системы. При моделировании наше внимание сосредоточено на описании некоторой части реального мира. Например, при моделировании приложения рассматривается его программная реализация, аппаратура, на которой исполняются программы, и пользователи, взаимодействующие с приложением. Все это вместе называется *физической системой*.

Одна и та же физическая система может быть описана с различных точек зрения. Описание физической системы называется *моделью*. Каждая модель описывает всю физическую систему в целом, но модели могут сильно отличаться степенью детальности, используемыми средствами и расстановкой акцентов. Это определяется целью, с которой составляется модель. Например, мы можем различать концептуальную модель, логическую модель и модель реализации для одной и той же физической системы.

Если физическая система сложна и велика (а именно так обычно и бывает), то ее целесообразно мысленно (а иногда и физически) разбить на части, называемые *подсистемами* и рассматривать отдельно и детально каждую подсистему.

Например, информационная система отдела кадров конкретной организации, как физическая система, состоит из некоторого количества компьютеров, программных компонентов, составляющих приложение и развернутых на этих компьютерах, а также служащих отдела кадров, которым вменено в обязанность использовать приложение в повседневной работе. С точки зрения выполняемых функций в данной физической системе можно выделить две подсистемы: подсистему работы с персоналом и подсистему работы со штатным расписанием.

Таким образом, мы можем структурировать наше описание физической системы различным образом: можно разделить все описание на несколько моделей, чтобы описать систему в целом с различных точек зрения, а можно разделить систему на части и описать их по отдельности как подсистемы. Более того, можно комбинировать эти структуры произвольным образом: разбить модель на несколько подсистем, каждую из которых описать с помощью нескольких моделей и т. д.

Для моделирования данных аспектов в UML предусмотрены соответствующие средства: *модель* и *подсистема* являются понятиями метамодели UML. С точки зрения языка модели и подсистемы являются разновидностями пакетов, поскольку во всех случаях речь идет о группировке элементов модели. На рис. 5.7 представлена диаграмма соответствующего фрагмента метамодели UML.

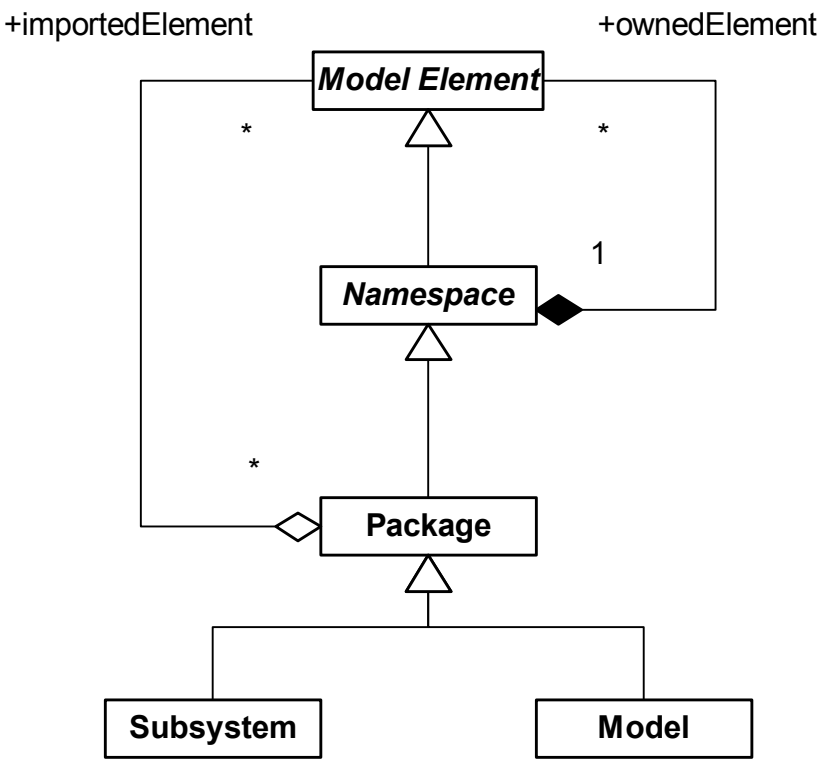



**Рис. 5.7. Метамодель управления моделями**

Синтаксически модели и подсистемы выделяются с помощью ключевых слов «`model`» и «`subsystem`», соответственно.

Все сказанное о пакетах, моделях и подсистемах выше в этом разделе нам представляется простым, очевидным и не вызывает никаких затруднений при рассмотрении с общей теоретической точки зрения. Затруднения начинаются при практической работе с конкретными инструментами. Нам встречались различные типы затруднений и в следующем списке мы перечисляем наиболее типичные.

- *Внеязыковой способ определения пакетов*. В некоторых инструментах такие разновидности пакетов, как модели и/или подсистемы *нельзя* определить обычным образом — нарисовав фигуру на диаграмме и указав соответствующее ключевое слово. Их нужно определять другим способом — с помощью специальных диалоговых окон и т. п. Например, пакеты верхнего уровня на рис. 5.1 суть модели. В использованном нами инструменте эти пакеты были созданы с помощью специальной команды и не могут быть отображены на диаграмме.[147]
- *Нестандартные имена стереотипов*. Спецификация UML предусматривает довольно большое число стандартных стереотипов для пакетов, они перечислены в табл. 5.1. Однако их назначение не всегда очевидно, а сами

[147] На рис. 6.1 это не заметно, но значки моделей в дереве вложенности пакетов даже по цвету отличаются от значков обычных пакетов.

названия не очень понятны. В результате разработчики инструментов зачастую подправляют этот список по своему вкусу.[148]

- *Расширение семантики пакетов*. Пакет в UML — это средство структурирования модели, но не моделируемой системы! Между тем инструментам, особенно инструментам, ориентированным на автоматическую генерацию кода, необходима информация для определения структуры этого кода: распределение кода по модулям, файлам, проектам и другим подобным конструкциям системы программирования. Но язык UML не зависит от системы программирования и поэтому подобные сведения в нем прямо не указываются. Разработчикам инструментов трудно устоять перед соблазном расширить семантику пакетов UML и интерпретировать структуру вложенности пакетов в терминах системы программирования. В результате пакетная структура, которая, может быть, была введена только для удобства описания модели, прямо переносится в структуру программного кода, что не всегда оправдано.

Впрочем, следует оговориться, что перечисленные затруднения, как правило, не являются существенными. Судя по нашему опыту, они легко преодолеваются (или обходятся) за несколько часов путем внимательного изучения справочной документации и конкретных примеров, прилагаемых к каждому инструменту.

**Таблица 5.1.** Стандартные стереотипы пакетов

| Стереотип | Описание |
|---|---|
| «facade» | Пакет, который содержит только *ссылки* на элементы, определенные в других пакетах. Используется для описания "внешнего вида" других пакетов. |
| «framework» | Пакет, содержащий образцы и шаблоны. Используется для описания повторно используемых архитектурных решений. |
| «metamodel» | Модель, которая описывает другую модель. Например, метамодель UML. |
| «modelLibrary» | Пакет, содержащий определения элементов моделирования, предназначенных для использования в других пакетах. |
| «profile» | Пакет, содержащий определения элементов моделирования, предназначенных для моделирования в определенной предметной области. |
| «stub» | Пакет, представляющий только открытые части другого пакета. |
| «systemModel» | Модель, содержащая несколько моделей одной физической системы. |
| «topLevel» | Пакет, который является конем иерархии вложенности пакетов. |

Мы хотим закончить этот раздел повторением важного тезиса, уже высказанного в разделе 5.1.1: UML позволяет придать описанию модели *любую* структуру — моделирующий субъект должен приложить определенные мыслительные усилия, чтобы выбрать *хорошую* структуру, подходящую для данного случая.

[148] Напомним, что в этом нет никакого нарушения стандарта — разработчики инструментов, равно как и пользователи языка, вправе определять и использовать свои стереотипы.

Универсального метода, как обычно, мы предложить не можем — только несколько советов для частных случаев. Вот один из них.

**ЗАМЕЧАНИЕ**

> Мы считаем целесообразным выделять подсистемы в соответствии с разбиением на компоненты связности диаграммы использования. Если на общей диаграмме использования, содержащей все идентифицированные варианты использования, явно выделяются компоненты связности, то это является, по нашему мнению, веским основанием для выделения подсистем.

### *5.1.4. Слияние пакетов в UML 2.0*

В UML 2.0 появился очень мощный механизм повторного использования использования моделей, которым сами разработчики стандарта интенсивно пользуются. Это так называемое *слияние пакетов*. При практическом моделировании слияние пакетов пока используется мало, поскольку эта операция довольно необычна и ее не так просто реализовать, но со временем, может быть положение изменится.

В операции слияния участвуют два пакета, которые условно называют «база» и «приращение», причем эта операция не коммутативная. Нотация: зависимость со стереотипом «merge» от базы к приращению.

Элементы базы сопоставляются с элементами приращения по именам и метаклассам. Приращение расширяет базовый пакет непротиворечивым образом.

Правила расширения свои для каждого метакласса. Результат прирашения является результатом операции слияния. Пусть, например (рис. 5.8), в пакете `Base` есть класс `A` (а также другие классы, не показанные на диаграмме). Пусть в пакете `Increment` также есть класс `A`, причем он участвует в отношении обобщения, так что класс `B` является суперклассом для класса `A` (этого не видно на диаграмме, потому что отношения между калссами внутри пакета на диаграмме пакетов не показываются). В результате слияния пакет `Base` расширяется (на рисунке результат расширения обозначен `IncrementedBase`), так что в него входит все что было в пакете `Base`, а также все то, что можно добавить из пакета `Increment` с сохранением непротиворечивости модели. В данном случае если в пакете `Base` не было класса `B` то он добавится вместе с отношением обобщения. Если в пакете `Base` был класс `B` но он не был свзяан с классом `A`, то добавится только отношениее обобщения. Если же в пакете `Base` был класс `B` и он был подклассом класса `A`, то ничего не добавиться, потому что отношения обобщения не должны образовывать циклов.

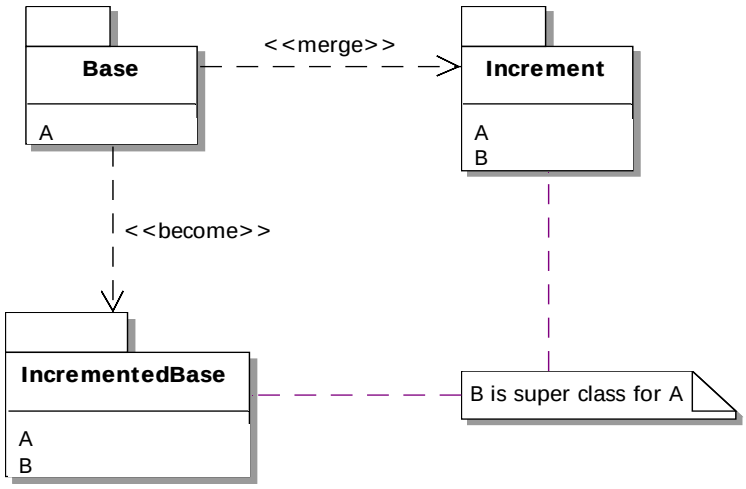



**Рис. 5.8. Пример слияния.**

### *5.1.5. Трассировка, гиперссылки и документация*

Одним из самых важных прагматических аспектов UML является, по нашему мнению, осознание того факта, что моделирование есть *процесс*, причем процесс не однозначный, итеративный, с возвратами и радикальным изменением уже сделанного. Отсюда неизбежно вытекает сосуществование диаграмм, представлений и моделей разного уровня абстракции, находящихся на разных этапах проработки, возможно даже противоречивых или несогласованных друг с другом. Разработчик модели нуждается в средствах управления всей этой разнородной информацией, ему нужны средства отслеживание версий, уровней детализации и т. д. Другими словами, ему нужны не только средства отображения в модели своих соображений о моделируемой физической системе, но и средства отображения информации о самой модели, ее состоянии и ходе изменений. Обычно такие средства называют средствами *трассировки*.

Разумеется, для этой цели можно пользоваться любыми подручными средствами. Например, проектируя функциональность, связанную с переводом сотрудника а информационной системе отдела кадров разработчик может записать на бумажке: "Рассмотреть возможность переноса операции `Move` в класс `Company`" или внести на диаграмму классов примечание с аналогичным текстом. Однако бумажки имеют удивительное свойство безвозвратно теряться, а примечание, будучи перенесенным средством автоматической генерации кода в комментарий к тексту программы, вызовет законное недоумение программиста.

Для удобной трассировки в UML предусмотрены специальные средства различного уровня. Мы начнем их рассмотрение со средств, встроенных в язык. Таковыми являются зависимости со специальными стереотипами — `«trace»` и `«refine»`. Данные зависимости являются внесистемными отношениями, т. е. отношениями между элементами модели, а не моделью отношений между моделируемыми сущностями.

Зависимость со стереотипом `«trace»` показывает причинные связи в модели, другими словами, показывает, что зависимая сущность имеет причиной своего существования независимую сущность. Например, если в модели информационной

системы отдела кадров имеется примечание с требованием включить в систему операцию перевода сотрудника, то для отражения данного требования может быть предусмотрен соответствующий вариант использования. Последний, в свое очередь, влечет существование пакета с диаграммами состояний, описывающий поведение объектов, представляющих должности и сотрудников (рис. 5.9).

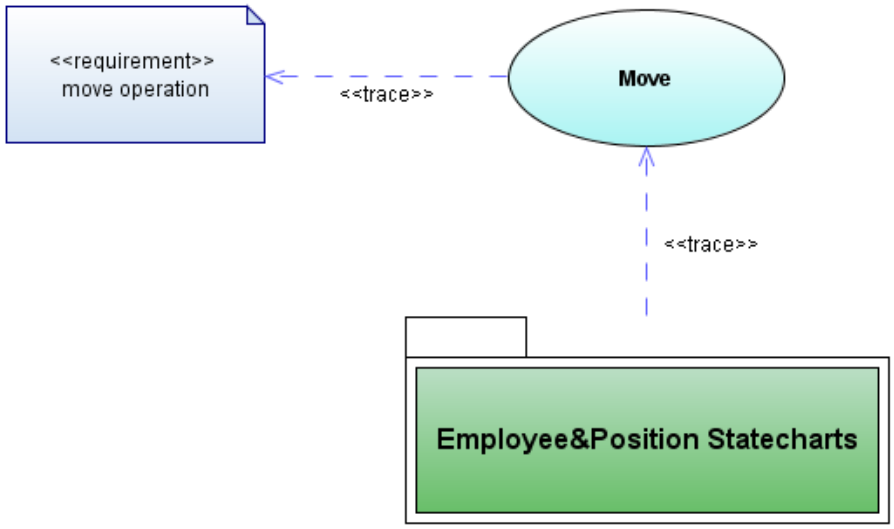



**Рис. 5.9. Зависимость со стереотипом** «trace».

Зависимость со стереотипом «refine» отражает последовательность разработки элементов модели. Если две сущности связаны зависимостью со стереотипом «refine», то это означает, что зависимая сущность является более детальным описанием того же самого моделируемого объекта, что и независимая сущность.

Данные зависимости имеют очевидную прагматику, но при их использовании возникает следующая практическая проблема. Определить зависимость трассировки в модели нетрудно, но как ее отобразить графически? Ведь, скорее всего, сущности, связанные данными зависимостями, отображены на разных диаграммах или даже в разных моделях (в смысле разд. 5.1.3). Рисовать специальные диаграммы для отражения истории работы над моделью — такую степень педантизма у современных разработчиков программного обеспечения трудно вообразить. А на обычных "рабочих" диаграммах зависимости со стереотипами «trace» и «refine» фактически "негде нарисовать". В результате данные зависимости, при всей их прагматической полезности и даже необходимости, на практике используются сравнительно редко.

К счастью, в современной практике работы с электронными документами имеется всем хорошо известное средство организации прослеживания и трассировки фрагментов документов — это *гиперссылки*. Все инструменты в той или иной степени поддерживают механизм гиперссылок в модели. Гиперссылки позволяют быстро переходить от одного элемента к другому, связанному с ним, и, таким образом, с лихвой перекрывают функциональность зависимостей со стереотипами «trace» и «refine». Следует заметить, что гиперссылки — это средство

инструмента,[149] а не языка UML, поэтому мы далее не будем развивать эту тему, а ограничимся одним советом.

**ЗАМЕЧАНИЕ**

> Если инструмент поддерживает гиперссылки, используйте их как можно шире — это значительно повысит читабельность модели.

Все инструменты в той или иной форме поддерживают документирование модели. Другими словами, для всех (или для почти всех) элементов во внутреннем представлении предусматривается специальное поле, где можно сохранить неформальное описание данного элемента в текстовом виде. Диалоговое окно, приведенное на рис. 5.10 довольно типично для большинства инструментов, поддерживающих UML.

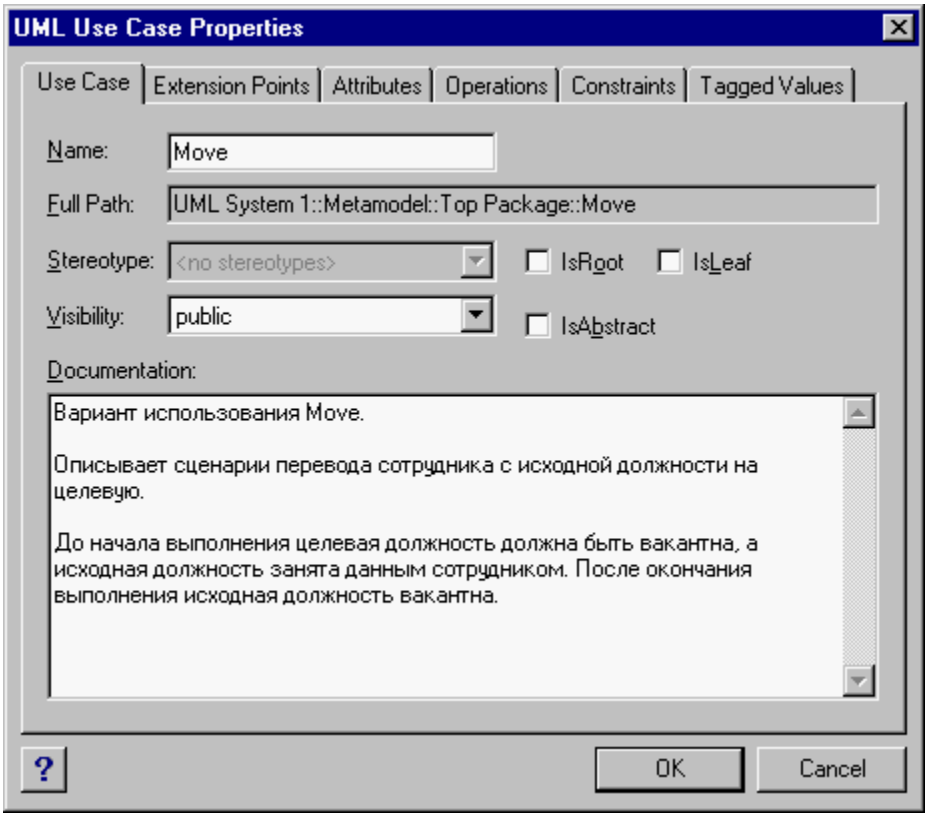



**Рис. 5.10. Документирование элементов модели**

Большая часть инструментов умеет весьма разумно распоряжаться текстом, введенным пользователем: вставлять его в виде комментария в генерируемый программный код, автоматически составлять текстовую документацию по модели и т. д. Другими словами, инструменты позволяют документировать модель должным образом — дело за разработчиками моделей.

---

[149] Инструменты далеко не равнозначны. В наилучших можно установить произвольное количество гиперссылок от данного элемента к любым другим элементам, в других же, как, например, в используемом нами для подготовки иллюстраций к этой книге, набор возможностей очень ограничен.

### 5.1.6. Образцы и каркасы

Образцы проектирования[150] – это сравнительно новая форма обмена программистским опытом и удачными проектными решениями. С момента выхода в свет книги "банды четырех"[151] образцы проектирования стали одной из наиболее горячих тем, обсуждаемых программисткой общественностью.

Неформально говоря, *образец проектирования* — это типичное решение типичной проблемы в данном контексте. Обычно описание образца состоит из четырех основных элементов.

- *Имя*. Ссылаясь на имя образца, мы можем кратко описать проблему проектирования, ее решения и их последствия. Это позволяет проектировать на более высоком уровне абстракции. Словарь общеизвестных имен образцов позволяет эффективно вести обсуждение с коллегами, лаконично документировать принимаемые архитектурные решения. Подбор хорошего имени — одна из важнейших задач при составлении описания образца.
- *Задача*. Описание контекста применения образца проектирования, т. е. описание конкретной проблемы проектирования и перечня условий, при выполнении которых имеет смысл применять данный образец.
- *Решение*. Описание элементов проектирования, отношений между ними, функции каждого элемента. Дается абстрактное описание задачи проектирования и ее обобщенное решение.
- *Результаты*. Здесь описываются следствия применения образца: влияние на степень эффективности, гибкости, расширяемости и переносимости системы.

Разумеется, образами проектирования пользовались и пользуются все разумные архитекторы испокон веков, и не только в области программирования, а буквально во всех областях человеческой деятельности. Почему же термин "образец проектирования" стал произноситься программистами столь часто именно сейчас? Мы рискнем высказать собственное мнение по этому вопросу: бум интереса к образцам проектирования в программировании непосредственно связан с появлением и широким распространением UML.

Действительно, до недавнего времени считалось, что хорошим архитектором, способным быстро, эффективно и надежно спроектировать сложную программную систему, может быть только достаточно опытный человек. Причем не просто потерявший много времени на неудачные самостоятельные попытки, а имеющий опыт успешной работы под руководством уже зарекомендовавшего себя архитектора и/или опыт работы в программирующей организации с богатыми традициями. При непосредственном общении ученик смотрит, как мастер решает сложные задачи, пытается поступать "по образцу", иногда ошибается, осмысливает ошибки, придумывает собственные приемы — короче, учится и набирается опыта. Именно поэтому хорошие архитекторы столь дороги: чтобы из новичка вырастить хорошего мастера, нужно затратить массу времени и привлечь блестящих учителей. Конечно, можно и нужно учиться по книгам, но написать книгу, которая

---

[150] Design Pattern — в программистский жаргон уже довольно прочно вошла калька с английского — "паттерн" (pattern), но мы, насколько это возможно, избегаем использования жаргона.

[151] GoF — Gang of Four (Gamma, Helm, Johnson, Vlissides)

может научить искусству программирования невероятно трудно — такие книги можно пересчитать по пальцам, в то время, как действующих мастеров программирования, обладающих ценнейшим опытом, многие тысячи.

Появление UML радикальным образом изменило ситуацию. По нашему мнению, появление UML имеет для программирование примерно такое значение, как изобретение нотной записи для музыки или введение буквенных обозначений для математики. Используя UML, архитектор программной системы может сообщить свои идеи (именно идеи, а не примеры готовых решений на языке программирования) в лаконичной и понятной форме, доступной для восприятия подавляющему большинству разработчиков. Индустрия разработки программного обеспечения — одна из самых объемных, именно поэтому всякое технологическое решение, обещающее существенное сокращение затрат (в данном случае на принятие квалифицированных архитектурных решений и на надежное проектирование), имеет такое большое значение и столь горячо обсуждается.

Рассмотрим подробнее понятие образца проектирования применительно к UML. Синтаксически в UML *образец* — это параметрическая кооперация классов (т. е. шаблон кооперации). Напомним (см. разд. 5.4.3), что в кооперации участвуют роли классификаторов, т. е., фактически, классификаторы (классы), входящие в кооперацию, можно рассматривать как параметры, а всю кооперацию в целом, как шаблон взаимодействия. Таким образом, чтобы применить некоторый образец проектирования (шаблон взаимодействия, т. е. кооперацию) в определенном контексте, достаточно связать параметры шаблона с конкретными значениями, т. е. указать, какие конкретные классы модели играют роли классификаторов, участвующих в данной кооперации (образце). Для этого в UML предусмотрен специальный синтаксис. Применяемый образец изображается в виде пунктирного овала, внутри которого написано имя кооперации. Этот овал соединяется пунктирными линиями с классами, которые являются фактическими аргументами, причем на линии указывается имя роли, которую класс играет в применяемой кооперации.

Рассмотрим все это более подробно на конкретном примере, в качестве которого мы используем классический образец проектирования, описанный в упоминавшейся книге «банды четырех» под именем Observer, а в других источниках упоминаемый под именем Publish-Subscribe.

- *Задача*. Поведение некоторых объектов системы (подписчиков — экземпляров класса `Subscriber`) должно зависеть от изменения состояния (события) другого объекта (издателя — экземпляра класса `Publisher`). Однако издатель не должен прямо взаимодействовать с подписчиками.[152]
- *Решение*. Ввести службу уведомления о событиях, с тем чтобы издатель мог опосредованно уведомлять подписчиков о наступлении события. Для этого вводится (единственный) объект класса `EventManager`, реализующий данную службу. Класс `EventManager` имеет метод `subscribe`, вызывая который

[152] Требование непрямого взаимодействия — одно из типичных условий в объектно-ориентированном проектировании. Оно может возникать по различным причинам. Например, класс Publisher предполагается повторно использовать в других системах, и поэтому его реализация не должна зависеть от того, какие классы подписываются на уведомление о его событиях.

подписчик подписывается на уведомлении о наступлении события, и метод `signalEvent`, посредством которого издатель уведомляет о наступлении события. При вызове метода `signalEvent` объект `EventManager` посылает уведомления о событии всем подписчикам, вызывая метод `notify`, переданный в качестве параметра при подписке. На рис. 5.11 приведена диаграмма кооперации, описывающая данный шаблон взаимодействия.


**Рис. 5.11. Диаграмма кооперации для образца проектирования Publish-Subscribe**

- *Результат*. Класс `Publisher` не зависит от класса `Subscriber`. Возможны различные модификации образца: при необходимости получать уведомления о различных событиях, нужно добавить соответствующий параметр (например, имя события); для увеличения эффективности можно обязанности ведения службы уведомления о событиях перепоручить прямо классу `Publisher`, но в этом случае снижается гибкость, поскольку реализовывать данную службу придется в каждом классе-издателе.

**ЗАМЕЧАНИЕ**

> Мы значительно упростили и сократили описание и обсуждение образца проектирования Publish-Subscribe (Observer), поскольку нашей целью является не описание конкретных образцов, а только обсуждение техники применения образцов в контексте языка UML. Для более детального изучения данного образца следует обратиться к первоисточнику.

Допустим теперь, что описано некоторое множество образцов проектирования и мы хотим применить один из них в конкретном контексте. Как происходит применение образца, если моделирование ведется средствами UML?

Рассмотрим пример из информационной системы отдела кадров. Пусть в нашей системе требуется уведомлять сотрудников об изменении состояния подразделения — вполне естественное требование, поскольку поведение сотрудников существенно зависит от состояния подразделения, и мы не хотим нагружать руководителя подразделения обязанностью персонально извещать каждого подчиненного — у руководителя и без того хлопот полон рот. В этом случае мы можем на диаграмме кооперации показать (рис. 5.12), что к классам `Department` и `Person` следует применить образец проектирования Publish-Subscribe, причем класс `Department` играет роль Publisher, а класс `Person` играет роль Subscriber.

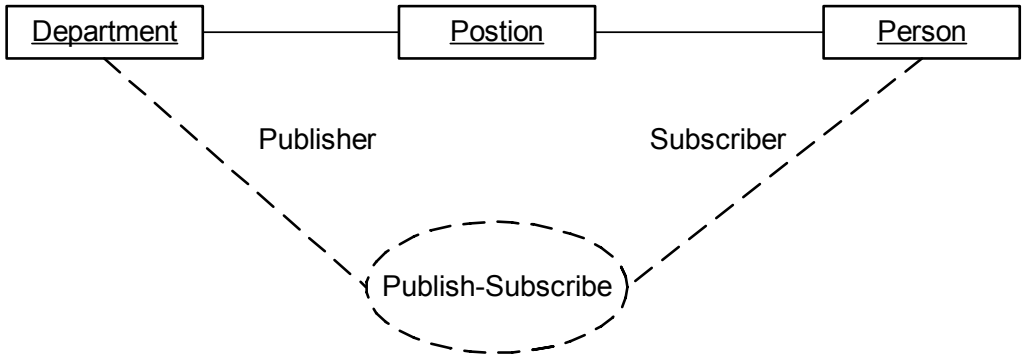



**Рис. 5.12. Применение образца проектирования**

По замыслу авторов UML, простая нотация на рис. 5.12 несет значительную семантическую нагрузку. Приведенная диаграмма определяет кооперацию, причем в этой кооперации определено гораздо больше элементов, чем нарисовано на диаграмме. В частности, подразумевается, что в модели определен класс `EventManager` (хотя он и не присутствует на диаграмме), между классами определены соответствующие ассоциации для вызова методов, сами классы обладают нужными методами для применения образца и т. д.

Любому разработчику совершенно ясно, что именно, почему и как нужно запрограммировать в информационной системе отдела кадров, чтобы обеспечить требуемое оповещение объектов класса `Person` об изменении состояния объекта класса `Department`. Такая ясность опирается на *понимание* образца проектирования Publish-Subscribe (тем более, что это образец, вероятно, хорошо известен большинству читателей и без наших объяснений). Однако было бы чрезмерным надеяться на то, что инструменты моделирования понимают образцы проектирования также хорошо, как люди. Поэтому в большинстве практических инструментов моделирование применение образцов носит менее интуитивный и более формальный характер. Весьма вероятно, что в используемом инструменте даже не найдется такой фигуры, как пунктирный овал, предназначенный для обозначения применения образца. Но с той же степенью вероятности найдется библиотека готовых образцов. Как правило, инструменты позволяют включить образец в модель (в форме заготовки диаграммы кооперации) и вручную связать (отождествить или переопределить) элементы образца с элементами модели. Другими словами, общая схема применения образца остается той же самой, но задача *понимания* образца остается за программистом.

В UML определено еще одно понятие, тесно связанное с образцами проектирования. Это *каркас* — совокупность логически связанных образцов. Синтаксически каркас в UML — это пакет со стереотипом «`framework`». Такой стереотип означает, что в пакете собраны образцы проектирования согласованные таким образом, чтобы их можно было применять совместно и систематически.

**ЗАМЕЧАНИЕ**

> Не все образцы проектирования, описанные в литературе, могут быть адекватно выражены в UML. Наилучшим образом язык подходит для описания таких образцов,

> в которых решение выражается в форме описания поведения взаимодействующих объектов.

Мы заканчиваем этот раздел списком тезисов и советов, отражающих наши рекомендации по использованию образцов проектирования при моделировании.

- Образец проектирования — это публикация обобщенного опыта профессионалов. Поэтому образец, как правило, полезен, разумен и применим в широком диапазоне ситуаций. Но ответственность за правильный выбор образца и его адекватное применение лежит, все-таки, на том, кто использует образец, а не на том, кто его публикует.
- Образец решает типичную и распространенную, но ограниченную подзадачу. Поэтому, как правило, образец лаконичен и тривиален. Редко бывает так, чтобы в образце использовалось больше элементов, чем может поместиться на одной простой диаграмме. Нет (и не может быть!) образцов, описывающих все систему в целом — даже интенсивно применяя образцы проектирования, архитектор все-таки должен думать собственной головой.
- В настоящее время опубликовано несколько десятков общепризнанных образцов и несколько сот используемых менее часто. Квалифицированный архитектор обязан знать (но не обязан каждый раз использовать) общепризнанные образцы, хотя бы по названиям.
- Если используемый инструмент моделирования содержит библиотеку образцов, то архитектору следует изучить данную библиотеку во всех деталях. Это настолько же полезно, насколько программисту полезно знать стандартные библиотеки используемого им языка программирования.
- Если в программирующей организации используется инструмент моделирования для разработки программного обеспечения, то создание корпоративной библиотеки образцов проектирования — это основа реального (а не формального) репозитория.

## 5.2. Влияние UML на процесс разработки

В предыдущем разделе мы косвенным образом затронули вопрос о том, как влияет систематическое применение UML на процесс разработки. Действительно, UML позволяет наглядно описывать образцы проектирования, применение образцов проектирования повышает качество проектирования архитектуры, что, по общему мнению, ускоряет реализацию. Таким образом, мы выделили один из факторов положительного влияния UML на процесс разработки программного обеспечения. Разумеется, это не единственный фактор. В первых главах книги мы привели достаточно много общих рассуждений на эту тему, а еще больше можно найти в других источниках, в частности, в книге [1]. В этом разделе мы собираемся сформулировать свое мнение по поводу того, в чем состоит основной фактор влияния UML на процесс разработки. Но для этого нам понадобится ввести в рассмотрение некоторые термины и понятия, после чего мы сформулируем наш основной тезис.

### *5.2.1. Технология программирования*

Дисциплина, которая изучает процессы разработки программного обеспечения, по-английски называется Software Engineering. Для обозначения этой дисциплины мы будем использовать устоявшийся отечественный термин "*технология*

*программирования*", хотя, может быть, этот термин немного устарел и не вполне точен, но нам так удобнее и привычнее.

Обсуждение вопросов технологии программирования в полном объеме выходит далеко за рамки этой книги.[153] Несмотря на то, что технология программирования начала развиваться одновременно с программированием, данная область знаний находится все еще скорее в стадии становления, нежели в стадии разработанной инженерной дисциплины, поэтому даже ее структура далеко не устоялась. У автора есть собственный взгляд на рациональный способ описания технологии программирования и мы им здесь воспользуемся — не с целью обсуждения или продвижения, поскольку столь краткое обсуждение не может претендовать на полноту и убедительность, а только для формулировки нашего тезиса о влиянии UML на практическое программирование.

Мы полагаем, что технологию программирования целесообразно рассматривать в трех аспектах.

- *Модель процесса*, т. е. порядок проведения типового проекта по разработке программного обеспечения. Сюда относятся понятия жизненного цикла программного обеспечения, определение модели процесса — выделение в нем фаз, вех, потоков работ и других составляющих. Характерным для данного аспекта является рассмотрение на уровне программирующей организации в целом.
- *Модель команды*, т. е. отношения между людьми в процессе разработки. Сюда относятся определение обязанностей работников — участников процесса, регламенты их взаимодействия, рабочие процедуры и т. п. Характерным для данного аспекта является рассмотрение на уровне группы (команды) или проекта.
- *Дисциплина программирования*, т. е. методы создания конкретных артефактов, входящих в состав программного обеспечения. Сюда относятся описание и применение образцов проектирования, стандарты кодирования, методы тестирования и отладки и т. д. Характерным для данного аспекта является рассмотрение на уровне отдельного работника.

По нашему мнению, едва ли не самым заметным свидетельством прогресса в технологии программирования является то, что, наконец, удалось договориться о терминах. В табл. 5.1 приведены объяснения некоторых терминов, использованных в предыдущем перечислении.

**Таблица 5.1.** Термины технологии программирования

| Термин | Определение |
|---|---|
| *Фаза* (phase) | Часть процесса работы над проектом. Обычно каждая фаза характеризуется вехой, достижение которой знаменует завершение фазы. |
| *Веха* (milestone) | Одномоментное идентифицируемое событие, сопровождающееся появлением и фиксацией некоторого артефакта, который называется результатом вехи. |

[153] Хотя мы их вынужденно постоянно затрагиваем, см., например, разд. 1.1.2.

| | |
|---|---|
| *Работник*[154] (worker) | Набор выполняемых функций и обязанностей, назначенных сотруднику в рамках конкретного проекта. |
| *Артефакт* (artifact) | Документ или иной материал, имеющий материальную форму и отчуждаемый от разработчика |

### *5.2.2. Повышение продуктивности программирования*

Очевидно, что задачей технологии программирования является исследование и улучшение процессов разработки программного обеспечения. При этом можно преследовать различные цели, т. е. улучшать процесс в различных направлениях. Приведем несколько примеров задач, исследуемых и решаемых технологией программирования.

- *Повышение надежности программного обеспечения*. Программы часто содержат ошибки — это трюизм для практикующих программистов. Пользователи возмущаются, но терпят, пока это возможно. Если же это терпеть невозможно (например, в приложениях, критически важных для жизни людей), то применяются специальные методы технологии программирования, позволяющие повысить надежность программ. Как правило, такие методы достаточно трудоемки и требуют значительных ресурсов.
- *Снижение совокупной стоимости владения программным обеспечением*. Программы довольно дороги — причем дорогой зачастую является не только и не столько разработка, сколько сопровождение, модификация, обучение пользователей. Некоторые методы технологии программирования позволяют за счет небольших дополнительных расходов на этапе разработки добиться значительного снижения затрат на этапе эксплуатации.
- *Повышение продуктивности программирования*. Под продуктивностью здесь мы понимаем объем[155] разработанного программного обеспечения в расчете на одного работника за единицу времени. Для малых и средних программирующих организаций данный показатель фактически определяет конкурентоспособность.

Мы остановимся именно на последней задаче, как наиболее насущной для нас и большинства наших читателей. За счет чего можно повысить продуктивность программирования? В соответствии с нашими взглядами на природу технологии программирования формальный ответ очевиден:

- за счет улучшения процесса;
- за счет организации команды;
- за счет совершенствования дисциплины программирования.

Сформулируем несколько утверждений относительно возможности повышения продуктивности за счет применения различных методов. Сразу оговоримся, что эти

---

[154] Часто используется также термин "роль". Но поскольку этот термин занят в UML, мы, вслед за авторами Унифицированного процесса будем использовать термин "работник".

[155] Проблема выбора измеряемых величин и единиц измерения для программ заслуживает отдельного обсуждения.

утверждения основаны на личных наблюдениях автора и не претендуют на статус всеобъемлющих законов природы.

Большинство программирующих организаций начинают борьбу за повышение продуктивности с формализации и документирования процесса. Мы считаем, что начальное введение формального процесса снижает продуктивность. Реальный рост продуктивности наблюдается только тогда, когда формализация процесса достигает достаточно высокого уровня зрелости, а именно, когда процесс является измеряемым, а значит, управляемым. В любом случае, за счет совершенствования процесса продуктивность программирования удается увеличить на проценты, но не в разы и не на порядки.

Эффективная организация работы команды может дать существенно больший выигрыш в продуктивности и с меньшими затратами. Однако, аналогично начальной формализации процесса, начальное введение иерархии подчиненности и формализация должностных обязанностей, по нашему мнению, снижают продуктивность. Продуктивность существенно возрастает, если применяются тонкие и сложные методы организации команды, такие как динамическое переназначение работников и управление по компетентности. Динамическое переназначение подразумевает, что данный сотрудник участвует, как правило, в нескольких проектах одновременно и на разных фазах выполняет разные обязанности. Идея состоит в том, чтобы каждый сотрудник делал не "то, что полагается", а то, что он умеет делать лучше всех. Управление по компетенции организовать еще труднее: каждое решение должно приниматься не "начальником", а наиболее компетентным именно в данном вопросе сотрудником.

Но, по нашему мнению, главным резервом для повышения продуктивности является дисциплина программирования. Программирование вообще является редким примером области человеческой деятельности, где разброс индивидуальной продуктивности на порядок является скорее правилом, нежели исключением. Мы считаем, что имеются два фактора, решающим образом влияющих на продуктивность программирования:

- сокращение объема внеплановых изменений артефактов;
- увеличение объема повторно использованных артефактов.

Внеплановые изменения возникает при исправлении ошибок. Причем наиболее болезненны, как хорошо известно, ошибки, допущенные на ранних фазах, т. е. ошибки проектирования. Речь идет не только и не столько об ошибках в коде программы — этот тип ошибок как раз сравнительно легко обнаружить и исправить. Неудачное архитектурное решение или плохой план пользовательской документации — примеры более серьезных ошибок.

**ЗАМЕЧАНИЕ**

> Изменения кода, вызванные изменениями требований, также, очевидно, уменьшают продуктивность. Однако это принципиально разные изменения. Грубо говоря, ошибки исправляются за счет разработчика, а изменения вносятся за счет заказчика. Поэтому исправление собственных ошибок всегда снижает продуктивность в денежном выражении, а переработка программного обеспечения по новым требованиям может и не сказываться на финансовой эффективности программирующей организации.

Повторное использование артефактов иногда наивно понимается как копирование текста из кода одной программы в код другой. Такая практика, разумеется, полезна,

но на продуктивность практически не влияет. Все, что удается сэкономить — это время на ввод текста. Но программисты хорошо знают, что время на ввод текста программы едва ли превышает 1% от общего времени ее разработки. Повторное использование компонентов (модулей, классов) также сопряжено с трудностями: либо в момент создания компонента необходимо приложить заметные дополнительные усилия на подготовку к повторному использованию, либо компонент окажется неготовым и повторно использовать его не удается. Впрочем, по нашему мнению, наличие собственных заготовок для повторного использования (инструментов, библиотек, компонентов в репозитории) является главным признаком зрелости программирующей организации. Мы считаем, что наиболее перспективным артефактом для повторного использования являются образцы проектирования (см. разд. 5.1.5), поскольку в этом случае повторно используется голова, а не руки программиста.

Мы подошли к формулировке тезиса о влиянии UML на процесс разработки программного обеспечения. Рассмотрим рис. 5.13 (который полезно сопоставить с рис. 1.2). На этом рисунке мы изобразили модель процесса разработки (несколько погрешив против канона UML ради наглядности). С основным циклом последовательного выполнения фаз процесса разработки сопряжены два внешних цикла движения определенных артефактов. Верхних цикл, в который вовлечен заказчик, отражает выявление несоответствий требованиям и, тем самым, определяет объем внеплановых изменений. Нижний цикл, в который вовлечен разработчик, отражает преобразование разработанных компонентов в готовые к применению образцы проектирования и, тем самым, определяет объем повторного использования.

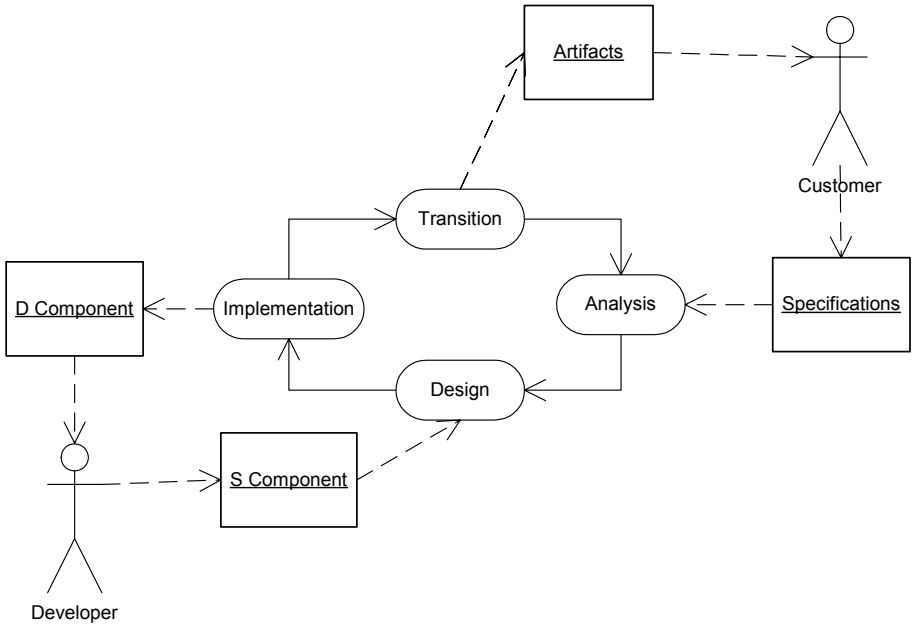



**Рис. 5.13. Циклы повышения продуктивности**

Тезис: применение UML оказывает положительное влияние на продуктивность процесса разработки программного обеспечения. Причина влияния заключается в том, что применение UML унифицирует представления информации в циклах

повышения продуктивности. Унификация обеспечивает ускорение прохождения циклов, повышение сохранности, облегчение восприятия и повышение надежности принимаемых решений.

Следует подчеркнуть, что названные причины имеют место, только если UML действительно применяется по существу. Если в программирующей организации имеются один или два энтузиаста, которые пытаются описывать свои решения на UML, а большинство над ними снисходительно посмеивается, то такое применение ни на что не повлияет. Если заказчик не может или не хочет верифицировать модель на ранних стадиях проектирования с целью выявления неправильно понятых требований, то толку не будет. Положительное влияние UML оказывается значительным, только если язык применяется массовым и систематическим образом.

## 5.3. Применение элементов UML

В данном разделе обсуждается исключительно прагматика UML — мы считаем, что нотация и семантика языка достаточно подробно изложены в предшествующих главах и знакомы читателю. Материал раздела носит субъективный характер — все, сказанное ниже, является личным мнением автора. Наше мнение основано на длительном (с ноября 1997 года) и разнообразном (применение при разработке заказного программного обеспечения, описание бизнес-процессов, перевод и научное редактирование оригинальных работ по UML, преподавание в различных аудиториях, разработка инструментов, поддерживающих UML) опыте использования UML для решения самых разнообразных задач. Признаемся: с переменным успехом. Именно поэтому мы думаем, что обсуждение нашего опыта может быть полезно читателю.

### *5.3.1. Уровни моделирования*

Первый вопрос, который надлежит поставить перед собой, приступая к применению UML: чего мы хотим достичь? Здесь, как говорится, возможны варианты. Мы рассмотрим три из них, наиболее нам близкие.

*Концептуальное моделирование*. В этом варианте целью моделирования является достижение *понимания* моделируемой физической системы (приложения, бизнес-процесса…), ее назначения и области применения. В среде профессиональных преподавателей, которые должны добиваться понимания материала обучаемыми, бытует представление о трех уровнях понимания:

- первый уровень — когда появляется приятное чувство, что все понял;
- второй уровень — когда можешь повторить сказанное;
- третий уровень — когда можешь найти ошибку.

Для понимания системы на первом уровне моделирование на UML не обязательно — достаточно устного объяснения и, может быть, нескольких наглядных слайдов с картинками. Второй уровень легко достигается с помощью связного текста на естественном языке. Применение UML оправдано, когда нужно достичь третьего уровня — не просто познакомиться с идеей системы, но *увидеть* ее в целом и, может быть, найти пробелы или ошибки.

По нашему мнению,[156] для концептуального моделирования полные диаграммы использования обязательны, а также желательны и полезны диаграммы классов (с минимумом подробностей) в объеме словаря предметной области, диаграмма компонентов или размещения для перечисления основных артефактов и, может быть, диаграммы взаимодействия или деятельности для типовых сценариев основных вариантов использования. В некоторых случаях (не во всех, а только в тех, когда это существенно для понимания логики работы приложения) оказываются полезны диаграммы состояний. Очень важно не забыть упомянуть в примечаниях те ключевые аспекты, которые не отражены в графической нотации. Главные требования к концептуальной модели — обозримость и согласованность. Диаграмм не должно быть много и они не должны быть сложными, но расхождения в обозначениях и названиях крайне нежелательны.

В нашей книге *нет* полной концептуальной модели информационной системы отдела кадров — мы не ставили перед собой такой цели, нашей задача заключалась в том, чтобы проиллюстрировать использования элементов языка UML. Но если бы мы взялись за реальную разработку информационной системы отдела кадров, то в концептуальную модель мы бы включили диаграммы, подобные диаграммам главы 2.

*Спецификация требований*. Во втором варианте применения UML основной целью моделирования является получение артефакта, который мог бы послужить техническим заданием на разработку. Техническое задание должно определять требования к системе и обеспечивать эффективную проверку того, что требования действительно выполнены. Также как и в случае концептуального проектирования, основой спецификации требований являются диаграммы использования, но их детальность и полнота должны быть существенно выше. Реализация вариантов использования в форме диаграмм взаимодействия обязательна — это основа для построения набора тестов приемо-сдаточных испытаний. Далее, на диаграммы использования возлагается значительно большая ответственность, поэтому разрабатывать их необходимо намного тщательнее. Например, если на диаграмме *не* изображен некоторый вариант использования, то это означает, что система *не* должна иметь соответствующую функциональность. В спецификации требований диаграмма компонентов не просто желательна, а обязательна — техническое задание должно определять, хотя бы на уровне названий, *что* именно должно быть разработано. Между концептуальной моделью и спецификацией требований нет непроходимой границы — довольно часто концептуальная модель за несколько итераций перерастает в спецификацию требований. По нашим наблюдениям, удовлетворительная спецификация требований оказывается в 3–5 раз объемнее концептуальной модели.

*Детальное проектирование*. Третий вариант применения UML требует известной решимости и, одновременно, трезвого расчета. Детальное проектирование нацелено уже не столько на описание, сколько на *конструирование* артефактов разрабатываемой системы. Модель детального проектирования служит основным документом, которым руководствуются программисты при кодировании, тестеры

[156] Наше мнение сходится с мнением авторов языка, см. [Руководство пользователя], советы по моделированию.

при проверке, а технические писатели — при составлении программной документации для системы. Если нет уверенности в способности команды построить и использовать хороший детальный проект исключительно средствами UML, то не стоит тратить на него ресурсы. Лучше не иметь никаких детальных диаграмм, чем иметь плохие. По нашим наблюдениям,[157] ошибка, допущенная в детальной диаграмме, обходится в несколько раз дороже, чем обычная ошибка в коде. Далее, детальное проектирование в несколько раз объемнее спецификации требований, поэтому, как правило, в процесс детального проектирования вовлекаются несколько человек, а значит, обостряются проблемы целостности модели, согласованности обозначений, сохранности артефактов. С другой стороны, если имеется опыт детального проектирования на UML, накоплены образцы проектирования в форме диаграмм кооперации, точно известны возможности инструмента по генерации кода на целевом языке программирования, то систематическое применение UML для детального проектирования ускоряет и удешевляет разработку в несколько раз. Поэтому стоит рискнуть и попробовать. Не расстраивайтесь (но будьте к этому готовы!), если самый первый опыт детального проектирования на UML окажется неудачным — на ошибках учатся. Невозможно научиться плавать, стоя на берегу.

В модели детального проектирования, как правило, используются *большая часть* элементов языка UML, рассмотренных в этой книге. Особое значение имеют диаграммы классов, которые должны быть определены с максимальной степенью детальности. Количество классификаторов возрастает в несколько раз по сравнению с концептуальной моделью. Помимо основных сущностей из словаря предметной области нужно определить типы данных, интерфейсы, сигналы, исключения и т. д. Как правило, сама модель имеет сложную структуру и без использования диаграмм пакетов также не обойтись.

Приведенная классификация уровней моделирования не претендует на универсальность. В литературе описаны и другие варианты, например, применение UML для описания бизнес-процессов, в том числе процесса разработки программного обеспечения. Но для разработки офисных приложений, о которых идет речь в главе 1, мы считаем указанные способы применения UML типичными. Во всяком случае, мы так делали и получалось неплохо — попробуйте и вы.

### *5.3.2. Советы по применению UML*

"Единственно правильной" модели нет и не может быть. Если программирование — это искусство, то что говорить об архитектурном проектировании и концептуальном моделировании? Это, видимо, элитарное искусство. Поэтому, если читатель рассчитывает найти в данном разделе советы в следующем стиле: "чтобы построить правильную модель, делай раз, делай два, делай три …", то его ждет разочарование — нет у нас в запасе подобных советов.[158] Довольно обширный список "положительных" советов по моделированию можно

---

[157] Другие авторы еще более категоричны в данном вопросе.

[158] Более того, тех, кто пытается давать такие советы в форме элементарных и однозначных процедур из трех шагов, мы подозреваем в сознательной профанации.

найти в книге [1]. Мы не хотим конкурировать и противопоставлять свое мнение мнению авторов языка, а потому зайдем с другой стороны: в этом разделе мы расскажем как *не* надо применять UML. Поскольку советы почерпнуты из опыта наших ошибок, они могут оказаться полезными читателю — учиться на чужих ошибках дешевле.
Не переоценивайте возможности инструмента.
Не пренебрегайте возможностями инструмента.
Не переоценивайте собственные возможности. Коэффициент замедления 2
Не пренебрегайте собственными возможностями. Расширяйте язык.

## 5.4. Выводы

- 7 ± 2 сущности на одной диаграмме.
- Диаграмма должна охватываться «одним взглядом».
- Управление моделями – для того, кто моделирует, а не для компьютера.
- В проекте сосуществуют разные модели в разных представлениях на разных уровнях абстракции.
- Образцы проектирования полезно знать.
- Стандарты полезны, но не универсальны — требуется подгонка для каждой программирующей организации
- Нет наилучшего процесса для всех типов проектов и всех типов организаций, но для каждого типа проектов и для каждого типа организаций в отдельности — есть.
- UML унифицирует представления артефактов в циклах повышения продуктивности, тем и хорош.

# Литература


1. UML User Guide (Three Amigos) Руководство пользователя (ДМК, 2000, Питер 2003)
2. UML Language Reference (Three Amigos) Специальный справочник (Питер, 2002)
3. UML Language Reference (Three Amigos) UML 2-е издание (Питер, 2006)

# Предметный указатель