Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Учебно-методическое пособие по дисциплине «Технологические подходы к разработке программного обеспечения»

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

Санкт-Петербург
2007

Оглавление

Введение 4
Цели и задачи дисциплины 4
Связь с другими дисциплинами 4
Структура и особенности курса 4
Тема 1. Технология программирования 6
1.1. Назначение технологии программирования 6
1.2. История развития технологии программирования 7
1.2.1. Дореволюционный период 7
1.2.2. «Революция в программировании» 8
1.2.3. Послереволюционный период 10
1.3. Типы программных проектов 11
1.4. Составные части технологии программирования 11
1.5. Проект, продукт, процесс и персонал 12
Тема 2. Жизненный цикл программы 13
2.1. Циклический характер разработки 13
2.2. Основные понятия технологии программирования 16
2.2.1. Процессы и модели 17
2.2.2. Фазы и витки 18
2.2.3. Вехи и артефакты 19
2.2.4. Заинтересованные лица и работники 19
2.3. Выявление и анализ требований 20
2.3.1. Требования к программному обеспечению 20
2.3.2. Схема разработки требований 22
2.3.3. Управление требованиями 22
2.4. Архитектурное и детальное проектирование 24
2.4.1. Архитектурное проектирование 25
2.4.2. Детальное проектирование 26
2.5. Реализация и кодирование 27
2.6. Тестирование и верификация 28
2.6.1. Процесс контроля качества 28
2.6.2. Методы «белого ящика» и «черного ящика» 29
2.6.3. Инспектирование и обзоры 29
2.6.4. Цели тестирования 31
2.6.5. Верификация, валидация и системное тестирование 31
2.7. Сопровождение и продолжающаяся разработка 32
Тема 3. Модели процесса разработки 34
3.1. Водопадные и конвейерные модели 34
3.2. Спиральные и инкрементные модели 35
3.3. Гибкие модели процесса разработки 37
3.4. Конструирование модели процесса 38
3.4.1. Выявление требований к процессу 39
3.4.2. Используемые фазы, вехи и артефакты 40
3.4.3. Выбор архитектуры процесса 44
3.4.4. Порядок проведения типового проекта 54
3.4.5. Документированные процедуры 71
3.4.6. Выводы 73
Тема 4. Модели команды разработчиков 74
4.1. Коллективный характер разработки 74

4.1.1. Оптимальный размер команды 74
4.1.2. Подчиненность участников проекта 74
4.1.3. Развитие команды и развитие персонала 75
4.1.4. Специализация, кооперация и взаимодействие 75
4.2. Иерархическая модель команды 76
4.3. Метод хирургической бригады 77
4.4. Модель команды равных 79
4.5. Конструирование модели команды 80
4.5.1. Особенности организации и требования к команде 81
4.5.2. Архитектура модели команды 81
4.5.3. Функции, роли и должности 82
4.5.4. Статико-динамическая структура команды 84
4.5.5. Распределение полномочий и ответственности 87
4.5.6. Достоинства и недостатки модели команды 88
Тема 5. Дисциплина программирования 89
5.1. Природа программирования 89
5.1.1. Наука программирования 90
5.1.2. Искусство программирования 91
5.1.3. Ремесло программирования 91
5.2. Парадигмы программирования 92
5.2.1. Структурное программирование 92
5.2.2. Логическое программирование 95
5.2.3. Объектно-ориентированное программирование 97
5.3. Циклы повышения продуктивности 99
5.3.1. Продуктивность программирования 99
5.3.2. Спецификации и модели 101
5.4. Программная архитектура 103
5.4.1. Событийное управление 104
5.4.2. Архитектура клиент/сервер 106
5.4.3. Службы 107
5.4.4. Трехслойная архитектура 109
5.5. Проектирование программ 109
5.5.1. Концептуальное проектирование 110
5.5.2. Логическое проектирование 111
5.5.3. Детальное проектирование 113
5.6. Кодирование 117
5.6.1. Программирование по образцу 118
5.6.2. Образцы проектирования 119
5.6.3. Доказательное программирование 122
5.6.4. Программирование вширь 123
5.6.5. Форматирование кода 124
5.7. Тестирование и отладка 129
5.7.1. Критерии приемлемости 130
5.7.2. Виды тестирования 131
5.7.3. Методы отладки 132
5.8. Инструментальные средства 133
5.9. Выводы 134
Предметный указатель 135
Литература 137

Введение

Совокупность известных технологических подходов к разработке программного обеспечения, или, несколько короче, технология программирования, является инженерной дисциплиной, входящей в обязательный набор знаний и умений всякого инженера, причастного к созданию и эксплуатации программного обеспечения компьютеров. Технология программирования имеет четко выделенный объект изучения – процессы разработки и сопровождения программного обеспечения, но, в настоящее время, не имеет единого метода и общепринятого способа построения. Технология программирования не является строгой математической дисциплиной, которую можно изложить последовательно, начиная с основополагающих понятий и применяя дедуктивные доказательства. Напротив, технология программирования является собранием разнородных и часто несогласованных друг с другом моделей, методик и средств. Детально изложить все используемые в технологии программирования приемы в рамках одного курса невозможно – их разнообразие слишком велико. Кроме того, технологические приемы разработки программного обеспечения очень быстро меняются, почти каждые полгода предлагаются новые подходы, и всякое изложение конкретных технологических решений в рамках учебного курса заведомо отставало бы от практики их применения в реальной жизни.
Вместо малопродуктивной попытки «объять необъятное», в данном курсе излагаются основные понятия, в терминах которых обычно формулируются реально используемые (и непрерывно обновляющиеся!) технологические модели, и приводятся некоторые примеры конструирования таких моделей. Прослушав данный курс, студент будет понимать, *как* строятся различные конкретные технологии программирования и, столкнувшись на практике с конкретной технологией, сможет ее понять, настроить и применить «по месту».

Цели и задачи дисциплины

Цели и задачи дисциплины состоят в подготовке студентов в области инжиниринга разработки программного обеспечения и получении студентами необходимых знаний и навыков применения технологических подходов для анализа, проектирования, разработки и применения программных и программно-аппаратных систем.
Кроме того, данная дисциплина должна содействовать расширению кругозора и развитию системного мышления студентов.

Связь с другими дисциплинами

Изучение дисциплины требует наличия базовых знаний процессов программирования и проектирования программного обеспечения. Опыта практической работы не требуется, хотя он, несомненно, очень полезен.
Многие схемы в методическом пособии выполнены в нотации, близкой к нотации унифицированного языка моделирования UML. Для свободного чтения схем желательно, но не обязательно предварительное знакомство с UML.
Данная дисциплина является вводной в цикле дисциплин, охватывающих различные аспекты инженерии программного обеспечения. Здесь вводится специфическая терминология, во всей общности рассматриваются основные понятия технологии программирования, даются ссылки на главные нормативные документы, приводятся обзоры различных разделов инженерии программного обеспечения, которые более детально рассматриваются в последующих курсах.

Структура и особенности курса

Курс имеет следующую структуру.

В первой теме рассматривается сам предмет дисциплины – технология программирования – с самых общих позиций. Здесь же вводится разделение технологии программирования на три составляющих, в соответствии с которой строится изложение курса.

Во второй теме вводится понятие жизненного цикла программы – ключевой абстракции, вокруг которой строятся все известные в настоящее время технологические подходы к программированию.

Темы с третьей по пятую раскрывают более детально составляющие технологии программирования, введенные в первой теме.

Обзорный характер изложения является важной особенностью курса. В курсе затрагивается множество разнообразных вопросов и используется большое число разнородных понятий. Центральные понятия определяются прямо в курсе, но отнюдь не все. Поэтому очень важной частью курса является изучение студентами дополнительной литературы. Такое «домашнее чтение» является основой самостоятельной и практической работы студентов в этом курсе.

Тема 1. Технология программирования

Процесс создания программ для компьютеров обычно называют программированием. Однако этим же словом часто обозначают и другие виды человеческой деятельности. Например, «математическое программирование» – раздел математики, или «политическое программирование» – словосочетание, которое нетрудно встретить в средствах массовой информации. Если речь идет именно о создании программ для компьютеров, то во избежание неоднозначности используют оборот *разработка программного обеспечения*. В этом курсе слово программирование используется только в смысле «процесс разработки программного обеспечения», а потому для краткости и удобочитаемости везде, где это возможно, используется термин «программирование» как синоним оборота «процесс разработки программного обеспечения».

Программирование (computer programming) — это процесс создания программистом (человеком) программы (информационной структуры), предназначенной для последующего исполнения (компьютером).

Как правило (в большинстве случаев), интерес представляет не только факт исполнения программы компьютером, но и использование результата исполнения человеком. Однако можно указать и несколько исключений, например, программа первоначальной загрузки операционной системы, так что включение ссылки на использование результатов исполнения программы человеком ограничило бы общность приведенного определения без явной необходимости. Мы не включаем указание на использование результатов исполнения программы человеком в явном виде в определение, но неявно подразумеваем, что в типичном случае такое использование имеет место.

Таким образом, в процессе программирования присутствуют явно субъект, объект и цель. В типичном (и привычном) случае субъектом является человек, который ведет процесс осознанно, объектом является текст на формальном языке, а целью является такое выполнение программы, которое в свою очередь имеет явно обозначенную цель. Далее, если явно не оговорено противное, подразумевается именно этот типичный случай.

Замечание

> Спектр вариантов процесса программирования отнюдь не исчерпывается рассматриваемым типичным случаем. Например, программирующим субъектом может быть не человек, а другая программа (автоматический синтез программы по формальным спецификациям или по примерам), процесс программирования может быть не осознан (запись макроса с помощью макрорекордера). Программа может быть выражена на нелинейном языке (нейрокомпьютер) или же не иметь материального носителя (план действий в голове пользователя графического интерфейса). Целью программирования может быть публикация текста программы (а не получение результата ее выполнения) или получение невыполнимой программы (защита от несанкционированного копирования).

В настоящее время эффективность и результативность программирования в целом оставляет желать лучшего. Несмотря повсеместное распространение компьютеров и очевидное улучшение их программного обеспечения, остается весьма значительной доля проектов по разработке программного обеспечения, которые нельзя считать вполне успешными. Наряду с эффектными достижениями имеются и сравнительно многочисленные досадные неудачи. К сожалению, до сих пор слишком часто приходится делать вывод, что программирование рискованно, программы ненадежны, а программисты неуправляемы.

1.1. Назначение технологии программирования

Проблемы в области программирования существуют, и отрицать это невозможно. Лучшие программисты, ведущие предприятия и компьютерное сообщество в целом постоянно тратят значительные и все возрастающие усилия на решение этих проблем. В результате в общем поле компьютерных наук выделились и дифференцировались различные дисцип-

лины. Дисциплина, которая направлена, прежде всего, на решение внутренних проблем программирования, получила название технологии программирования, или инженерии программных систем. Последний термин является точным переводом английского термина software engineering.

Технология программирование (software engineering) – это совокупность методов и средств, позволяющая наладить производственный процесс создания программного обеспечения.

В этом определении особо следует подчеркнуть слово «производственный», которое отражает важнейшую особенность технологии программирования. Например, в официальном определении родственной дисциплины – информатики (computer science) указание на производственный характер дисциплины отсутствует.

Информатика (computer science) – это дисциплина, изучающая общие свойства информации, а также вопросы, связанные с ее сбором, хранением и обработкой.

Поскольку сбор, хранение и обработка информации в настоящее время выполняются, главным образом, с помощью компьютеров, а компьютеры не могут работать без программного обеспечения, информатика оказывается неразрывно связанной с программированием.

Соотношение технологии программирования (software engineering) и информатики (computer science) трудно описать в двух словах. Информатика – более общее понятие, но считать технологию программирования просто частью информатики было бы не совсем верно. В современной практике применения этих отраслей знания те вопросы, которые имеют преимущественно теоретический характер, принято относить к информатике, а сугубо практические приемы принято считать элементами технологии программирования. Например, методы математического доказательства правильности программ обычно относят к теоретической информатике, а методы тестирования – к технологии программирования, хотя это различные методы решения одной и той же задачи. Конечно, это отделение технологии программирования от информатики несколько условно и строгой границы здесь нет, но тенденция очевидна.

1.2. История развития технологии программирования

Программирование – сравнительно молодая область человеческой деятельности. Как массовая профессия программирование существует около полувека, и этот срок намного меньше, чем история развития других инженерных дисциплин. Неудивительно, что программирование еще не избавилось от многих «детских болезней», характерных для всех новых видов человеческой деятельности. Но программирование развивается стремительно, а детские болезни быстро проходят.

В истории технологии программирования происходило множество событий, предлагалась и предлагается масса идей. Чтобы как-то систематизировать эти факты, мы предлагаем следующую классификацию.

Все факты истории технологии программирования делятся на три класса, которые приближенно соответствуют трем периодам.

1.2.1. Дореволюционный период

С момента начала промышленной разработки программного обеспечения и до середины шестидесятых годов XX века вопросы собственно технологии программирования рассматривались, как правило, не отдельно, а в связи и в совокупности с другими вопросами программирования. Разумеется, технологические проблемы существовали, и предлагались методы их решения, но это не было предметом публичных общественных дискуссий.

Компьютеры были дороги, их количество было невелико, и применялись они, в основном, для специальных целей (оборона, космос и т.п.). Следует подчеркнуть, что в это время программирование не являлось массовой профессий, и было, в основном уделом талантливых и высокообразованных одиночек, специалистов в той предметной области, где применялись компьютеры. Можно сказать, что хотя программирование и было высоко профессиональным, оно не было промышленным.

Из основных технологических идей, появившихся в этом период, следует отметить появление языков программирования и компиляторов и явное осознание важности модульного программирования, как основы для накопления библиотек программ и их повторного использования.

1.2.2. «Революция в программировании»

К середине шестидесятых годов ситуация изменилась. Компьютеры стали дешевле, компактнее и производительнее. Сфера применения существенно расширилась, они стали в массовом порядке применяться в промышленности, науке, образовании. Наряду со сложными и ответственными программами (о самом существовании которых не везде можно было говорить вслух) появились, и в гораздо большем количестве, обычные программы для автоматизации производственной и иной повседневной деятельности.[1] Эти обычные программы изготавливались практически в каждой организации, имевшей компьютеры, независимо от основного профиля ее деятельности. Эксплуатация программного обеспечения перестала быть таинством, доступным только посвященным, программирование стало массовой профессией.

Заметим, что общество в целом не сразу осознало социальные последствия происходящего. В это время уже хорошо были известны положительные и отрицательные последствия других видов инженерно-технической деятельности. К проектированию таких изделий, как мосты и самолеты, любители уже не допускались (все понимали, что плохо спроектированный мост может обрушиться, а самолет упасть, и это недопустимо). Проектирование в традиционных инженерных областях велось в соответствии с многочисленными стандартами, правилами, регламентами и инструкциями, в которых число требований к качеству исчисляется сотнями и тысячами. Иное дело программирование: программисты в то время в большинстве своем и не слыхивали о каких-то стандартах качества своей работы. Да и действительно: подумаешь, программа расчета зарплаты «зависла» — это же не самолет упал!

Но низкая надежность — это только одна сторона проблемы. Хорошая технология не только улучшает качество, она еще и увеличивает производительность. А плохая технология — уменьшает. Отсутствие явно выписанной технологии — это самая плохая технология. В начале шестидесятых технология программирования, в современном понимании и как массовое явление, отсутствовала. Реальная средняя производительность труда была низкой, что хорошо видно из отраслевых нормативов производительности программирования тех лет. Еще хуже дело обстояло с результативностью. Именно тогда были проведены первые методически обоснованные исследования и появились отчеты, из которых следовало, что менее половины проектов по разработке программ являются успешными.[2] В средствах массовой информации появились мрачные прогнозы (полученные простой экстраполяцией наблюдаемых значений показателей), что к концу двадцатого века все трудоспособное население будет программировать, и программ будет не хватать. Кризис программирования был налицо.

Очень быстро было предложено множество идей и подходов для выхода из кризиса.

---

[1] Сейчас такие программы называются бизнес-приложениями.

[2] Проект считается успешным, если в плановые сроки в рамках выделенного бюджета удается получить запланированный результат. В противном случае (то есть сроки не выдержаны и/или бюджет перерасходован и/или сделано не все) проект не считается успешным.

Традиционно принято считать, что «первой ласточкой», положившей начало лавинообразному процессу сотворения технологии программирования, было письмо Э. Дейкстры в журнал Communications of the ACM в 1968 году.[3] Очевидно, что письмо Дейкстры подействовало как катализатор, как манифест – за несколько лет были опубликованы, обсуждены и практически внедрены следующие фундаментальные идеи технологии программирования.

- Конструирование программ методом пошагового уточнения.
- Проектирование сверху вниз и снизу вверх.
- Структурное программирование.
- Метод хирургической бригады.
- Водопадная модель процесса разработки.
- Жизненный цикл программного продукта.

По каждому направлению традиция называет основоположников (авторов наиболее замеченных из ранних публикаций), но фактически эти технологические идеи явились плодами совместных усилий, сотни людей внесли весомый вклад в это стремительное развитие. Перечислять все фамилии в учебнике нет возможности, но, по крайней мере три фамилии «отцов-основателей» технологии программирования необходимо упомянуть.

Э́дсгер Ви́бе Де́йкстра (нидерл. Edsger Wybe Dijkstra; 11 мая 1930, Роттердам (Нидерланды) — 6 августа 2002) — выдающийся нидерландский учёный, идеи которого оказали огромное влияние на развитие компьютерной индустрии.

Известность Дейкстре принесли его работы в области применения математической логики при разработке компьютерных программ. Он активно участвовал в разработке языка программирования Алгол. Будучи одним из авторов концепции структурного программирования, он проповедовал отказ от использования оператора GOTO. Также ему принадлежит идея применения семафоров для синхронизации процессов в многозадачных системах и алгоритм нахождения кратчайшего пути на ориентированном графе с неотрицательными весами рёбер. В 1972 году Дейкстра стал лауреатом премии Тьюринга.

Дейкстра Э. Дисциплина программирования. М.: Мир, 1978, 275 с.

Дал У., Дейкстра Э., Хоор К. Структурное программирование. М.: Мир, 1975. 247 с.

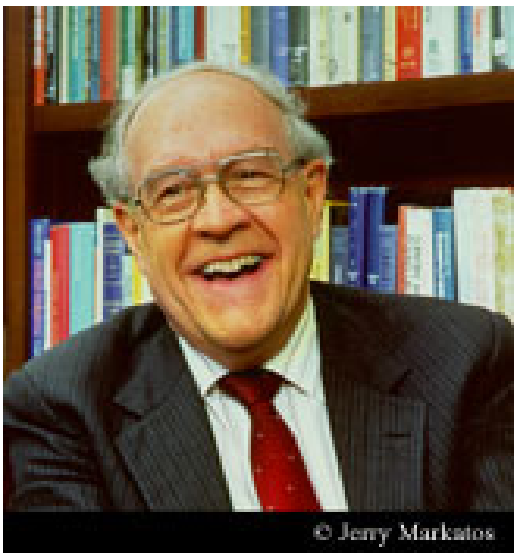
Никлаус Вирт (нем. Niklaus Wirth, род. 15 февраля 1934) — швейцарский учёный, специалист в области информатики. Ведущий разработчик языков программирования Паскаль, Модула-2, Оберон. Обладатель премии Тьюринга 1984 года.

Вирт Н. Систематическое программирование. Введение. М., Мир, 1977. 184 с.

Вирт Н. Алгоритмы + структуры данных = программы. М., Мир, 1978. 410 с.

[3] Оригинальное название письма звучало так «A Case Against the Go To Statement». Редактор журнала, а им был Н. Вирт (!) предложил бессмертное название «Go To Statement Considered Harmful», под которым письмо и было опубликовано.

Фредерик Филипс Брукс мл., род. 19 апреля 1931 – американский менеджер, инженер и ученый, наиболее известен как руководитель разработки операционной системы OS/360. В 1975 году, обобщая опыт этой работы, написал книгу «Мифический человеко-месяц». Повторно книга вышла в виде юбилейного издания в 1995-ом, вместе с комментариями автора и новым эссе «Серебряной пули нет».
Брукс насмешливо называл свою книгу «библией программной инженерии»: «все её читали, но никто ей не следует!»
Фредерик Брукс является лауреатом премии Тьюринга 1999 года.



Брукс-мл. Ф. П. Как проектируются и создаются программные комплексы. М.: Наука, 1975; новое издание перевода: Мифический человеко-месяц. СПб.: СИМВОЛ+, 1999.

1.2.3. Послереволюционный период

Революция была успешной — в исторически кратчайшие сроки технология программирования оформилась как инженерная дисциплина, и некоторые ее положения повсеместно были внедрены в практику. Результаты проявились незамедлительно — средняя производительность труда в программировании увеличилась в несколько раз. Повысилась и надежность, за счет развития практики систематического тестирования. Кризис разрешился.
Развитее технологии программирования продолжалось, но уже не революционным, а эволюционным путем. Взятые за основу, идеи 60-х годов развивались и совершенствовались. Структурное программирование переросло в объектно-ориентированное, на смену водопадным моделям процесса пришли итерационные, метод хирургической бригады дал начало целому спектру моделей команды разработчиков. Но эти подходы не были революционно новыми, они улучшали и совершенствовали уже известное.[4]
Однако время не стоит на месте. Аппаратное обеспечение стремительно прогрессирует. Сфера применения компьютеров расширяется все больше. Уже сейчас без них нигде нельзя обойтись, компьютеры применяются повсеместно. Пока еще слово «компьютер» ассоциируется у среднего жителя нашей планеты с образом персонального компьютера, с экраном и клавиатурой, а программирование ассоциируется с программированием на персональном компьютере и для персонального компьютера. Но это только пока. В самое ближайшее время компьютеры из устройств, которые используются очень часто и во многих местах, превратятся в устройства, которые используются везде и всегда. Речь идет о том, что сейчас называется встроенными системами. Персональные компьютеры выпускаются миллионами штук в год. Встроенные системы выпускаются миллиардами штук в год, а будут выпускаться многими миллиардами! И все эти устройства требуют для своей работы программного обеспечения. Не следует думать, что программировать эти устройства легко. Напротив. Уже сегодня возможности компьютера, скрытого в вашем мобильном телефоне, превосходят возможности того компьютера, которым пользовался Дейкстра, когда писал свое знаменитое письмо. Вполне вероятно, что консервативного усовершенствования старых идей, которого пока хватало для организации программирования персо-

[4] Из всего спектра технологий программирования, появившихся за последнее время, только для аспектно-ориентированного программирования трудно указать предтечу в пионерских работах конца 60-х начала 70-х годов. Все остальные «новинки» последних лет – прямые аналоги либо хорошо забытых, либо малоизвестных работ прошлых лет.

нальных компьютеров, уже не хватит для организации программирования необозримого парка встроенных систем. Тогда технологию программирования ждут новые, действительно революционные изменения. Необходимо быть готовым к этому.

Подчеркнем, что предлагаемая классификация – не более чем мнемоническая схема для удобства запоминания и анализа. Она не раскрывает *причин* изменений в технологии программирования.

1.3. Типы программных проектов

Множество программ неоднородно – спектр реальных и возможных программных систем очень широк. Программы можно классифицировать по множеству различных факторов. Например:

- тип программно-аппаратной платформы, на которой выполняется программа: встроенная систем, настольное приложение, сетевой сервис;
- размер программы (измеренный в байтах, командах, операторах, строчках кода или иных единицах);
- число пользователей: однопользовательская программа «для себя», внутренняя программа организации, программный продукт общего пользования;
- регулярность использования: разовая программа, регулярно используемая программа, постоянно работающая программа;
- источник разработки: заказная разработка, инициативная разработка, конкурсная разработка;
- предметная область: критически важная программа специального применения, приложение для автоматизации бизнес-процессов, демонстрационный прототип.

Этот список можно продолжать и продолжать, причем как в длину, указывая новые факторы, так и в ширину, указывая новые значения факторов. Огласить «весь список» очень трудно.

Важно, что при проведении проектов по разработке программ разных типов целесообразно использовать различные технологии. Программу управления ракетой разрабатывают не так, как информационную систему отдела кадров. Очень часто апологеты конкретных технологий, особенно новых, еще не зарекомендовавших себя широко, смело утверждают, что их технология «лучшая в мире». Очень может быть, что новая рекламируемая технология действительно сейчас лучшая для того типа программных проектов, для которого она была придумана. Но совершенно невероятно, чтобы она оказалась действительно лучшей для всех типов программных проектов, везде и на все времена.

При описании технологий программирования необходимо указывать типы программных проектов, для которых данная технология применима – то есть область применимости.

1.4. Составные части технологии программирования

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. Различные авторы, так же как и различные стандарты, принятые в этой области, используют несколько отличающиеся друг от друга структуры изложения технологии программирования, что дает нам известную свободу в выборе структуры изложения материала.

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 состав-

ляющих. 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.

1.5. Проект, продукт, процесс и персонал

Цель любого программного *проекта* состоит в производстве некоторого программного *продукта*. То, как в рамках проекта производится продукт, представляет собой *процесс*. Поскольку критичным для успеха дела является взаимодействие членов команды, мы включаем в рассмотрение *персонал*. Принцип четырех «П» отражен на рис. 1.

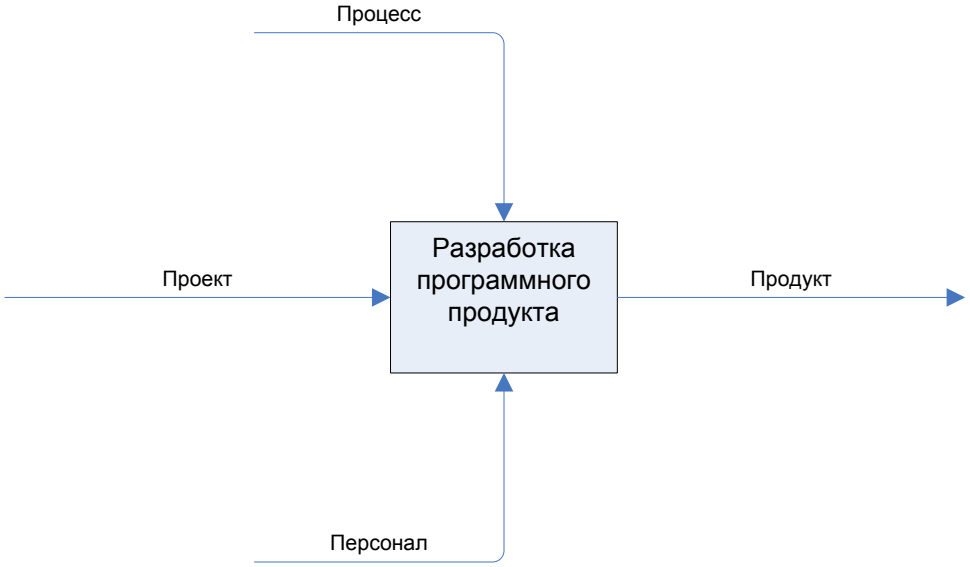


Рис. 1. Четыре «П» технологии программирования: проект, продукт, процесс и персонал

Здесь использована знакомая многим нотация структурного анализа. Входом является проект, выходом – продукт, обеспечивающим механизмом – персонал, а управляющим воздействие – описание процесса.

Тема 2. Жизненный цикл программы

При обсуждении технологии программирования широко используется понятие жизненного цикла программы. Жизненный цикл программы – это некоторая абстрактная модель, однако элементы принимаемого жизненного цикла являются тем материалом, из которого стоятся различные конкретные модели технологии программирования. Поэтому обсуждение понятия жизненного цикла программы необходимо должно предшествовать обсуждению элементов всякой технологии программирования.
В рамках данной темы определяются и обсуждаются наиболее общие идеи и понятия, которые интенсивно используются в последующих разделах курса.

2.1. Циклический характер разработки

Давно замечено, что программа за время жизни претерпевает многочисленные изменения своей формы, зависящие от состояния процесса разработки программы.

Жизненный цикл программы – это совокупность и последовательность изменений формы программы за все время ее существования.

В разных парадигмах и моделях технологии программирования понятие жизненного цикла определяется и трактуется немного по-разному, но, в общем, близко к схеме, представленной на рис. 2. Важно подчеркнуть, что за время своей жизни программа проходит метаморфозы, как правило, несколько раз,[5] т.е. это именно цикл, который повторяется не один раз, а несколько раз.

Замечание

> Программа является преимущественно идеальным (а не материальным) объектом, поэтому в жизненный цикл включаются и те "эмбриональные" формы, в которых программа существует во время разработки. С другой стороны, материально существующие на магнитных носителях программы для больших машин не используются и не сопровождаются, а потому их жизненный цикл считается завершенным. Таким образом, программа жива, пока она жива в голове у программиста и/или пользователя.

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

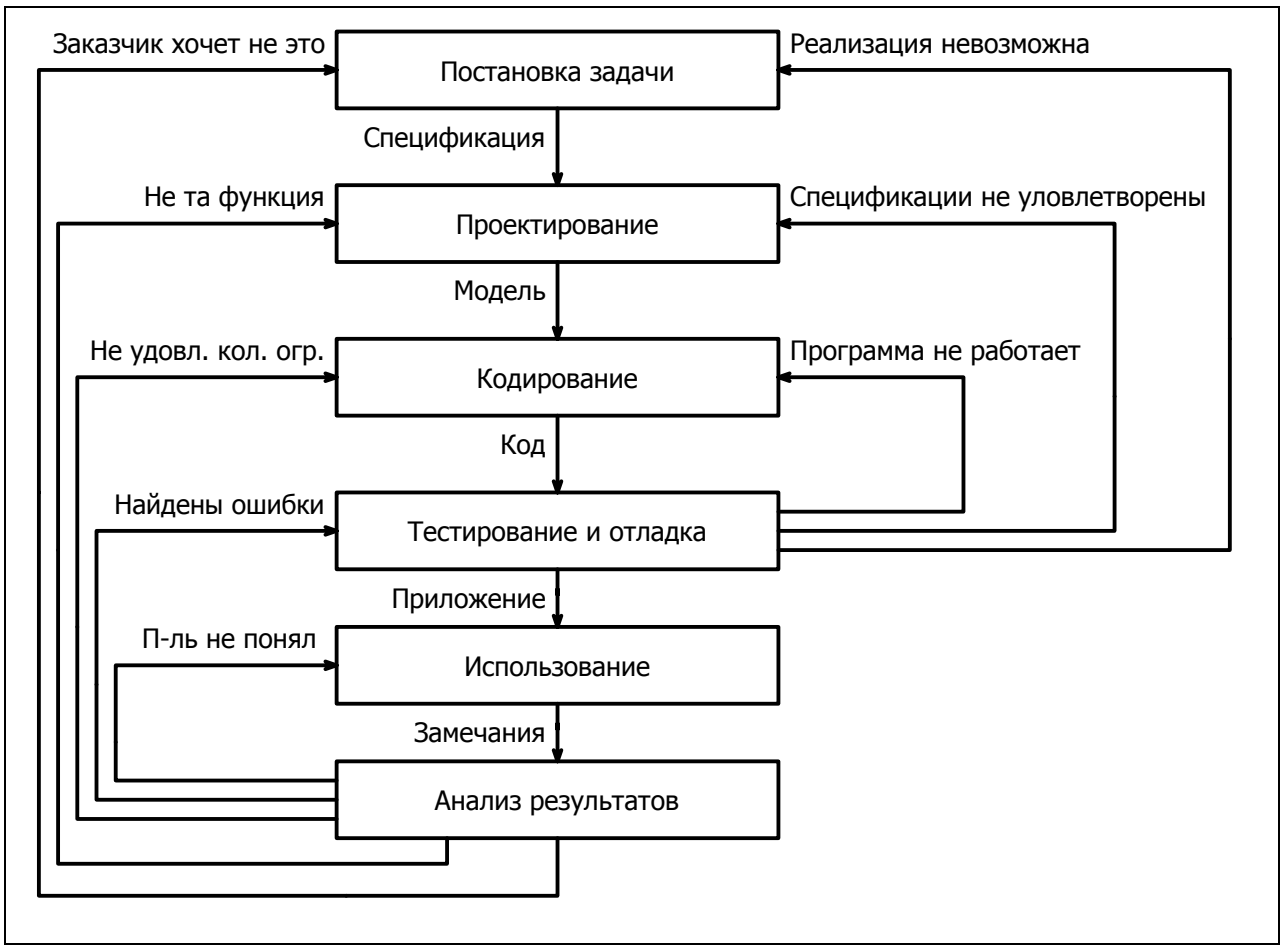


Рис. 2. Жизненный цикл программного обеспечения

На приведенной схеме используются два вида графических обозначений: фигуры символизируют состояния, в которых находится программа, а стрелки – переход из одного состояния в другое. Оба типа элементов нагружены дополнительной информацией: текст внутри фигур обозначает деятельность, проводимую в данном состоянии, а текст над стрелкой – условие или событие, управляющее данным переходом.

Переплетение циклов может быть довольно запутанным, особенно для долго живущих программ. Для упорядочения циклической структуры используется понятие выпуска (release).

Выпуск – характерная точка в жизни программы, отмечающая прохождение одного полного цикла.

Как правило, термином «выпуск» обозначают не только событие (момент в истории жизни программы), но и отчуждаемый артефакт, конкретную форму программы, появление которой отмечает данное событие.

Эта идея четко выделена в дисциплине проектирования Microsoft Solution Framework (MSF), представленной на рис. 3. В результате появляется характерная спиральная структура с периодическими выпусками очередных версий программы.

Замечание

В материалах MSF используется более сложная схема из нескольких вложенных спиралей, по одной для каждой формы существования программы, которая более точно и детально отражает существо методологии программирования MSF. Для понимания основной идеи достаточно приведенной упрощенной схемы.

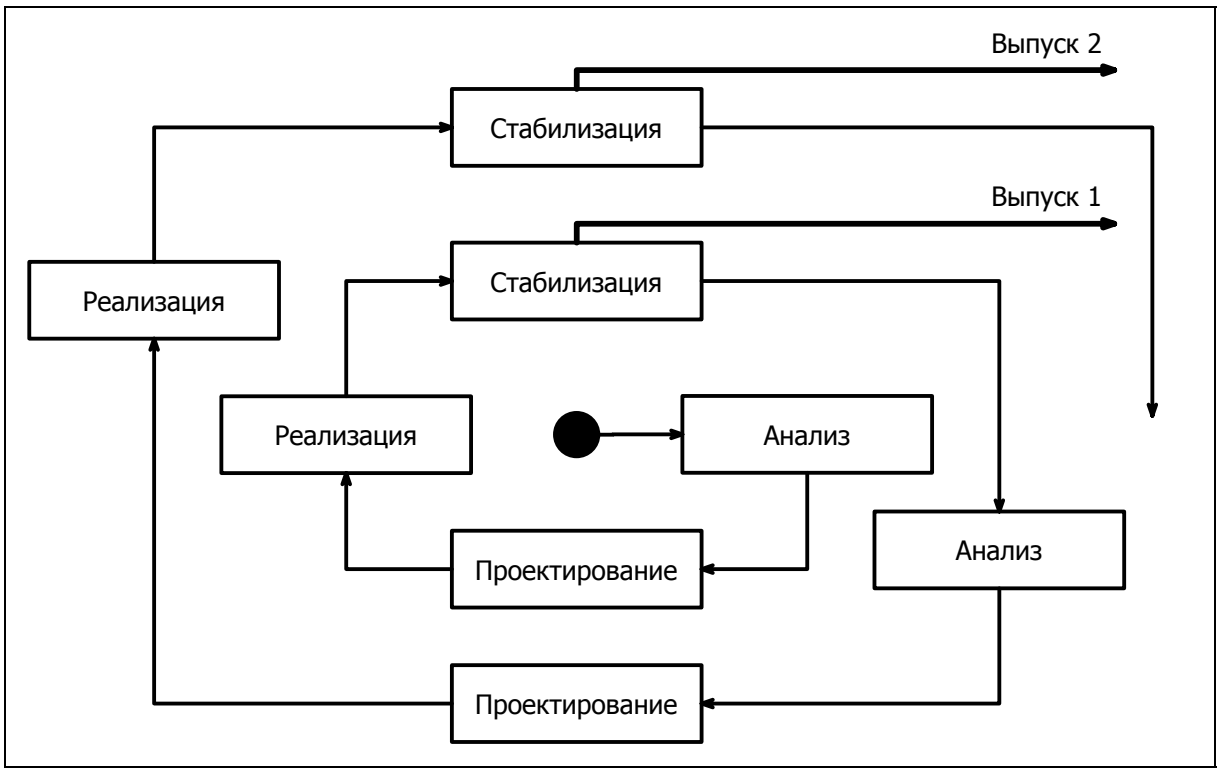


Рис. 3. Спиральная схема жизненного цикла программы

Следует отметить три существенных момента этой схемы:

- подразумевается *одна* долго живущая программа,
- жизненный цикл имеет видимое *начало*,
- жизненный цикл потенциально *бесконечен*.

Программные продукты корпорации Microsoft, опыт разработки которых аккумулирует MSF, обладают именно этими свойствами. Проекты же, которые проводит типичная отечественная программирующая организация в настоящее время, обладают другими характеристиками:

- Типичная программирующая организация проводит параллельно относительно *много* проектов, каждый программист участвует в нескольких проектах одновременно в режиме разделения времени с мелким квантованием (недели, дни или даже часы).
- Круг заказчиков имеет тенденцию к стабилизации в смысле выделения постоянных партнеров, для которых периодически выполняются новые проекты, опирающиеся на результаты прежних, таким образом, многие проекты начинаются *не с нуля*.
- Каждый отдельный проект *конечен* и, как правило, предусматривает один[6] выпуск и краткосрочное сопровождение. Если программный продукт подлежит значительной модификации, то это оформляется отдельным договором, а значит, является независимым проектом.

Чтобы учесть эти особенности, заметим следующее:

- В жизненном цикле программа взаимодействует с двумя категориями действующих лиц: заказчиком (пользователем) и (абстрактным) программистом, причем последний может быть представлен как программно (software), так и умственно (brainware).

[6] В большинстве проектов по разработке приложений для автоматизации бизнеса предусматривается передача заказчику двух версий: прототипа на фазе опытной эксплуатации и окончательной версии на фазе внедрения. С точки зрения организации процесса эти версии необходимо различать, но с точки зрения жизненного цикла программы это один выпуск, потому что реализует один набор функций, отраженный в спецификациях.

- Программ много, они взаимно влияют друг на друга на разных фазах своих циклов через совместно используемых программистов и заказчиков.
- Оба действующих лица (заказчик и программист) имеют двунаправленные связи с программой: заказчик воздействует на программу формулировкой требований при постановке задачи на фазе анализа, а программа воздействует на заказчика своим выпуском после фазы стабилизации; программа воздействует на программиста тем, что после фазы реализации разработанный код (а также другие материалы, например, модели) сохраняется в голове у программиста и/или в репозитории, воздействие программиста на программу очевидно — программист творец программы.

Учет этих наблюдений приводит к модели жизненного цикла множества программ, представленной на рис. 4.

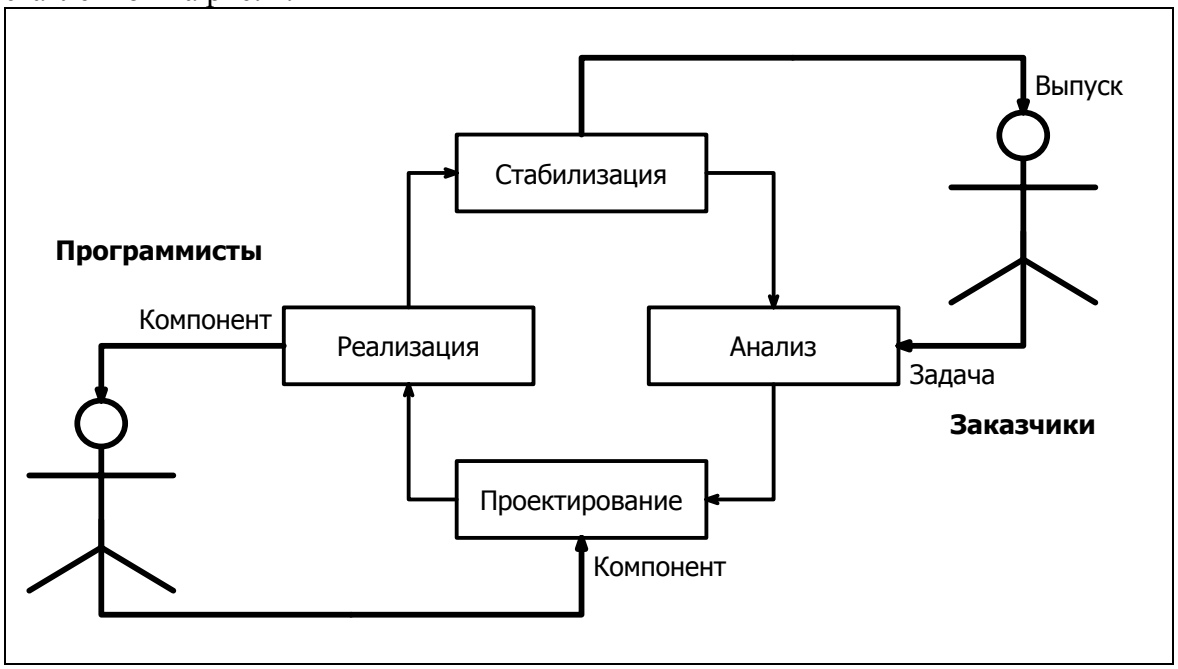


Рис. 4. Жизненный цикл множества программ

Приведем некоторые примеры, иллюстрирующие и подтверждающие предлагаемую нами схему, приведенную на рис. 4. Информационная система масштаба предприятия *обязана* иметь "стандартный" (часто не самый лучший для конкретного случая!) интерфейс и всякие изыски в этой области категорически запрещаются просто потому, что пользователь и раньше работал с другими программами и привык к определенному типу интерфейса. Программист из проекта в проект использует "стандартную" трехслойную архитектуру приложения (часто не самую лучшую для конкретного случая!) просто потому, что он знает эту архитектуру.

Замечание

> Программа может воздействовать через заказчиков и программистов не только на другие программы, но и на самое себя. Хотя на схеме направление стрелок через программиста противоположно направлению смены фаз, а направление стрелок через заказчика проходит через веху выпуска, но, поскольку модель процесса может состоять из нескольких витков, программа может воздействовать на свою более раннюю фазу на более позднем витке!

В следующих разделах этой темы мы кратко охарактеризуем основные понятия, как использованные нами в предыдущих схемах, так и применяемые в различных известных моделях жизненного цикла, с целью уточнить используемую терминологию.

2.2. Основные понятия технологии программирования

В данном разделе рассматриваются общеизвестные термины, которые используются в технологии программирования. Такое рассмотрение необходимо, потому что иначе очень

легко упустить некоторые тонкости, специфические для рассматриваемого предмета, положившись на «общеизвестность» терминов. По нашему мнению, едва ли не самым заметным свидетельством прогресса в технологии программирования является то, что, наконец, удалось договориться о терминах.[7]

2.2.1. Процессы и модели

Так или иначе, предметом технологии программирования является *процесс* разработки программного обеспечения.

Процесс – 1) Последовательность смены состояний в развитии чего-нибудь. 2) Последовательность действий для достижения какого-либо результата.[8]

Заметим, что два процитированных определения не являются разными определениями – это описание одного и того же предмета с разных точек зрения. Можно обратить свое внимание на смену состояний. Но что является причиной смены состояний? Очевидно, выполнение каких-то действий. Можно обратить свое внимание на выполнение действий. Но что является результатом выполнения действий? Очевидно, смена состояний.

Чтобы иметь саму возможность делать какие-то наблюдения или проводить рассуждения относительно процесса, нужно иметь описание процесса, или его модель. Термин «модель» широко используется в различных областях человеческой деятельности, и имеет много значений. В данном случае подразумевается следующее значение.

Модель – любой образ или аналог (мысленный или условный: изображение, описание, схема, чертеж и т.п.) какого-либо объекта, процесса или явления («оригинала» модели), используемый в качестве его «представителя».[9]

В разных областях человеческой деятельности используют совершенно различные средства для построения моделей. Физические процессы описывают математическими моделями, например, системами дифференциальных уравнений. Социальные процессы описывают длинными и невнятными пассажами на естественном языке.

Процесс разработки программ также пытались описывать самыми разными способами, но постепенно к настоящему времени сложилась традиция использовать для описания процесса программирования примерно те же средства, которые используются для самого программирования. Программы же, как всем известно, записывают как алгоритмы, выраженные на том или ином языке программирования.

Обсуждение значения этого факта мы отложим до темы 5, а здесь заметим, что языков программирования очень много (тысячи) и среди них встречаются как такие, которые предназначены для записи программ, исполняемых компьютером, так и такие, которые для этого не предназначены. Например, различные псевдокоды, блок-схемы и тому подобное. Язык программирования, если он предназначен для восприятия людьми, совсем не обязан быть формальным, точным и сугубо бедным, как «настоящий» язык программирования, подобный C++ или Java. Сейчас процессы разработки программного обеспечения принято описывать как алгоритмы, но очень неформально, с использованием понятий высокого уровня, далеких от машинной реализации. Конечно, неформальный характер таких алгоритмов приводит к тому, что у разных авторов встречаются несколько различные толкования основных элементов, из которых строится описание процессов разработки, но, тем не менее, за последние годы фактически устоялся общий набор таких основных понятий, и мы их перечисляем в следующих разделах.

---

[7] Фактическая стабилизация и унификация терминологии стала наблюдаться сравнительно недавно, с начала этого века. Но поскольку переводчики зарубежных книг по технологии программирования редко заглядывают в утвержденные терминологические ГОСТы, в отечественной литературе пока еще наблюдается некоторый разнобой в терминологии.

[8] Большой энциклопедический словарь дает еще одно определение: порядок рассмотрения дел в судопроизводстве. Но в данном случае это значение слова «процесс» не имеет отношения к предмету.

[9] Большой энциклопедический словарь дает еще шесть толкований этого термина.

Замечание

> Следует иметь в виду, что в современных технологиях программирования процесс разработки программного обеспечения рассматривается не как один последовательный процесс, а как несколько параллельных процессов, взаимодействующих друг с другом. Это накладывает дополнительные сложности алгоритмического и терминологического характера.

2.2.2. Фазы и витки

При описании процессов, связанных с разработкой программного обеспечения, основными структурными составляющими являются фазы и витки.

Фаза (phase) – часть процесса разработки. Обычно каждая фаза характеризуется *вехой*, достижение которой знаменует завершение фазы.

На фазу можно смотреть и как на состояние процесса, и как на действие (деятельность) в процессе. Рассмотрим, к примеру, фазу, которая обычно выделяется в процессе разработки: выявление и анализ требований. С одной стороны, это совокупность разнообразных действий: совещания с заказчиками, интервьюирование пользователей, анализ осуществимости и другие. С другой стороны, это состояние процесса, которое характеризуется тем, что требования еще не определены, и значит нецелесообразно или невозможно выполнять другие действия, например, архитектурное проектирование, кодирование, тестирование. «Требования определены» – это веха, достижение которой знаменует завершение фазы определения и анализа требований. Когда эта веха достигнута, процесс переходит в другую фазу, обычно в фазу архитектурного проектирования.

В разных источниках используются также термины *стадия* или *этап*. Мы предпочитаем термин «фаза» по следующей причине. Слова «стадия» и «этап» имеют оттенок линейной упорядоченности, подразумевается, что следующий этап меняет предыдущий, и они не совмещаются во времени. Иногда фазы действительно строго меняют друг друга, но чаще бывает так, что фазы частично перекрывают друг друга и выполняются параллельно – например, требования еще окончательно не утверждены и фаза извлечения требований еще не закончилась, но уже начаты работы по архитектурному проектированию.

Наборы фаз, которые включают в модель процесса разработки, различны в разных технологиях программирования. Чаще всего встречаются следующие фазы.

- Извлечение и анализ требований.
- Архитектурное и детальное проектирование.
- Реализация и кодирование.
- Тестирование и верификация.
- Сопровождение и продолжающаяся разработка.

Замечание

> К сожалению, часто одна и та же фаза в разных моделях называется по-разному. Это связано с устойчивыми традициями и вряд ли может быть изменено. Следует иметь в виду это обстоятельство, и, встречая незнакомое название фазы, тщательно проверять, не является ли название просто новым именем для хорошо знакомой фазы.

В следующих разделах упомянутые фазы рассмотрены подробнее.

Поскольку процесс разработки программного обеспечения имеет циклический характер, многие фазы повторяются несколько раз. Необходимо как-то идентифицировать и различать разные вхождения одной фазы в жизненный цикл. С этой целью, используя метафору спирального развития процесса разработки, вводится термин виток (см. рис. 2).

Виток – последовательность неповторяющихся фаз в жизненном цикле программы.

Вообще говоря, разные витки в одном жизненном цикле могут иметь разный состав входящих в виток фаз. Поэтому витки обычно выделяют не произвольным образом, а так, чтобы каждый виток заканчивался выпуском[10] (см. разд. 2.1).
Довольно часто используют и другой термин – *итерация*.[11]

2.2.3. Вехи и артефакты

Каждая фаза является достаточно крупной составляющей процесса разработки: фаза может продолжать достаточно долго, и требовать выполнения большого объема работы. Более того, фазы могут перекрываться во времени и выполняться параллельно. Таким образом, понятие фазы оказывается недостаточно четким для точного планирования, учета и измерения процесса. В то же время планирование и измерения считаются совершенно необходимыми элементами в современном процессном подходе. Нужно средство установить границы фазы более четко. Таким средством является понятие «веха».

Веха (milestone) – одномоментное идентифицируемое событие, сопровождающееся появлением и фиксацией некоторого *артефакта*.

Обычно при планировании и анализе процессов, в конце каждой фазы выставляется веха, которая служит границей фазы. Такую веху естественно называть главной вехой фазы, а ее артефакт называется результатом вехи.
Например, для фазы «выявление и анализ требований» главная веха – «требования определены», а результатом фазы являются сами требования.
Заметим, что совершенно необязательно считать, что каждая фаза имеет только одну веху: их может быть сколько угодно. Например, начальная веха, которая отмечает начало фазы, промежуточные вехи, которые отмечают важные части фазы или появление важных артефактов необходимых для выполнения других фаз и так далее.
Мы уже много раз использовали слово «артефакт». Вообще говоря, это слово происходит от латинских корней [arte искусственно + factus сделанный], и означает любую искусственно созданную вещь. В последний годы этот термин активно стал применяться в технологии программирования в следующем смысле.

Артефакт (artifact) – документ или иной материал, имеющий материальную форму и отчуждаемый от разработчика.

В этом определении важно обратить внимание на слово «отчуждаемый». Поясним это на примере модельной ситуации. Допустим, что после некоторых разговоров с заказчиком один из разработчиков заявляет: «я понял, что нужно сделать». Можно ли считать, что веха «требования определены» достигнута? В некоторых ситуациях действительно достаточно иметь оракула, умеющего отвечать на любые вопросы, относящиеся к разрабатываемому программного обеспечению, по мере их возникновения. Однако в современных промышленных технологиях программирования так поступать не принято – слишком велик риск. Действительно, что делать, если оракул заболеет или уволится? Считается, что веха «требования определены» должна сопровождаться артефактом, в котором эти требования действительно определены и их можно использовать без непосредственной помощи тех лиц, которые составляли требования.

2.2.4. Заинтересованные лица и работники

Процесс разработки программного обеспечения не является автоматическим – в нем участвуют люди. Но этот процесс, в настоящее время, не является полностью ручным – для разработки программного обеспечения используется программное обеспечение.

[10] Иногда программисты вместо аккуратного термина «выпуск» используют слово «релиз», полученное прямой транслитерацией английского термина «release», который на самом деле в данном случае как раз и означает «выпуск».
[11] Например, термин «итерация» применяют в одной из популярнейших моделей процесса – Rational Unified Process (RUP), название которой обычно переводят на русский язык как Унифицированный процесс.

Программы, которые используется при разработке программ, называются *инструментальными*, или *инструментами*.

У инструментальных средств есть пользователи – те конкретные сотрудники программирующий организации, которые и ведут процесс разработки. При описании (построении) промышленного процесса разработки считается, что сотрудники до некоторой степени взаимозаменяемы,[12] и важны не личностные качества сотрудника, а его профессиональные навыки и умения.

Работник[13] (worker) – набор выполняемых функций и обязанностей, назначенных сотруднику в рамках конкретного проекта.

Но помимо работников, непосредственно занятых в процессе разработки, с каждым проектом по разработке программного обеспечения связаны категории людей и организации, которые так или иначе вовлечены в проект. Примерами могут служить: заказчики, которые влияют на определение требований, инвесторы, которые финансируют проект, пользователи, которые собираются эксплуатировать разрабатываемое программное обеспечение и так далее.

Заинтересованными лицами (stockholders) называют категории людей или организации, которые тем или иным образом связаны с проектом.

Так же как и в случае с работниками, при описании модели процесса принято абстрагироваться от конкретных особенностей реальных заинтересованных лиц, и учитывать только отношения, которые данные заинтересованные лица имеют к проекту.

2.3. Выявление и анализ требований

Фаза с таким или сходным названием (чаще всего применяют термин «разработка требований») присутствует во всех известных моделях жизненного цикла. Более того, современные тенденции в технологии программирования таковы, что данная фаза все чаще рассматривается не только как первая, но и как главная, а иногда и решающая фаза жизненного цикла. Дело в том, что заметные успехи технологии программирования позволяют, при наличии адекватных требований, организовать выполнение других фаз, если и не как автоматический, то, во всяком случае, как управляемый и измеримый процесс. Другими словами, если современные программисты точно знают, что именно нужно сделать, они это, скорее всего, сделают. Верно и обратное: если требования к программному обеспечению не определены, или недостаточно определены, то, скорее всего, успешной разработка не будет. Об этом свидетельствует статистика, собранная при анализе проведения проектов — большая часть причин неуспешного выполнения конкретных проектов была квалифицирована как ошибки или недоработки на фазе анализа требований.

2.3.1. Требования к программному обеспечению

Если мы поймем, что такое требование к программной системе, то сможем определить и то, что делается в фазе «Выявление и анализ требования».
Существует большой диапазон решений этой проблемы.

- **Простое решение**. Понятие «требование» не определяется. Разработчик и заказчик полагаются, в этом случае, на здравый смысл. Риск такого решения определяется тем, что бизнес-цели, опыт и квалификация у заказчика и разработчика различны.

[12] Существует целый ряд областей человеческой деятельности, прежде всего в науке и искусстве, где предположение о взаимозаменяемости сотрудников не всегда верно.
[13] Часто используется также термин "роль". Но поскольку этот термин занят в унифицированном языке моделирования UML, мы, вслед за авторами Унифицированного процесса будем использовать термин "работник".

- **Обычное решение**. Дать какое-либо, возможно не очень точное или не очень понятное определение. Например

Требование – это документированное указание потребности или цели пользователей либо условия и возможности, которым должен обладать продукт, чтобы удовлетворить такие возможности или цели.

Или

Требования – это высокоуровневые обобщенные утверждения о функциональных возможностях и ограничениях системы.

Риск этого решения примерно такой же, как и для предыдущего случая.

- **Стандартное решение**. Использовать стандартное определение понятия «Требование».

  Стандарты IEEE используют следующее определение требований.

1. Функциональность, необходимая пользователю для решения проблемы или достижения цели. 2. Функциональность, которая должна быть получена (достигнута) системой или ее компонентами для соответствия контракту, стандарту, спецификации или другим формальным документам. 3. Документальное представление пп. 1 – 2.

Таким образом, понятие «требование» в этом определении апеллирует либо к пользователям, либо к формальным документам. В любом случае подчеркивается обязательность документального представления.

Российский стандарт ГОСТ 12207 дает другое решение проблемы, в котором понятие «требование» определяется перечислением тех видов требований, которые предъявляются к программному продукту и, практически, не требуют расшифровки.

В соответствии со стандартом ГОСТ 12207 на стадии «Анализ требований» должен быть выполнен анализ требований к программным средствам. Эта работа состоит из следующих задач:

1. Разработчик должен установить и документально оформить следующие требования к программным средствам: • функциональные и технические требования, включая производительность, физические характеристики и окружающие условия, под которые должен быть создан программный объект; • требования к внешним интерфейсам программного объекта; • квалификационные требования; • требования безопасности, включая требования, относящиеся к методам эксплуатации, сопровождения, воздействию окружающей среды и травмобезопасности персонала;

и т.д. Всего расшифровка понятия «требования» в ГОСТ 12207 занимает несколько страниц.

- **Сложное решение**. Проводится формализация (математическое описание) предметной области и требования формулируются как формальные математически строгие утверждения.

Разумеется, такие требования наилучшим образом описывают программную систему. Однако в промышленных технологиях программирования сложное решение применяется не часто. Дело в том, что формальные математические требования оказываются больше по объему, требуют больше трудозатрат на разработку, и требуют более высокой квалифика-

ции разработчиков, чем обычное программирование. Поэтому разработка строгих математических требований к программе оказывается дороже разработки самой программы, причем может быть даже на несколько порядков дороже. В специальных случаях приходится идти на такие затраты, но в типовом программировании для автоматизации бизнес-процессов этого не делают.

2.3.2. Схема разработки требований

Разработка требований – это первая из основных фаз процесса создания программных систем. Этот фаза состоит из следующих основных работ (рис. 5).

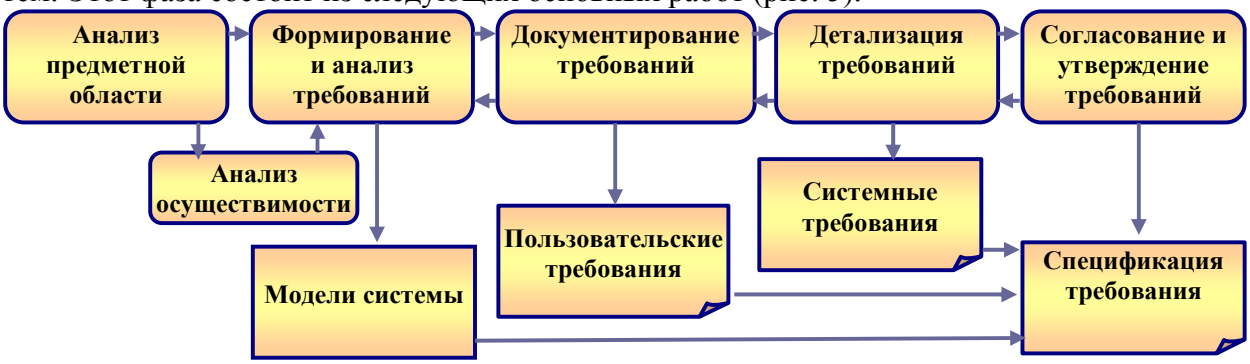


Рис. 5. Схема процесса разработки требований

- **Анализ предметной области**. Позволяет выделить сущности предметной области, определить первоначальные требования к функциональности и определить границы проекта.
- **Анализ осуществимости**. Должен выполняться для новых программных систем. На основании анализа предметной области, общего описания системы и ее назначения принимается решение о продолжении или завершении проекта.
- **Формирование и анализ требований**. Взаимодействуя с пользователями, обсуждая и анализируя с ними задачи, возлагаемые на систему, разрабатывая модели и прототипы, разработчики формулируют пользовательские требования.
- **Документирование требований**. Сформированные на предыдущем этапе пользовательские требования должны быть документированы. При этом нужно учесть, что основными читателями этого документа будут пользователи, поэтому основными требованиями к нему будут ясность и понятность.
- **Детализация требований**. Разработчики детализируют требования пользователей, формируя более точные подробные системные требования.
- **Согласование и утверждение требований**. На этом этапе пользовательские и системные требования должны быть оформлены в виде единого документа, содержащего все функциональные и нефункциональные требования. Такой документ, обычно, называется спецификацией требований. Спецификация требований должна удовлетворять следующим характеристикам качества: корректность, однозначность, завершенность и согласованность.

2.3.3. Управление требованиями

На этапе анализа разрабатываются пользовательские и системные требования к программной системе, которые оформляются в виде единого документа – спецификации требований, – являющегося формальным соглашением заказчика с разработчиком системы.
Практика показывает, что требования к разрабатываемой программной системе часто изменяются. Это обусловлено тем, что разработка программной системы довольно длительный процесс, во время которого:

- понимание пользователями возможностей системы, решаемых ею задач, может измениться;

- происходят изменения в деловой среде, техническом, программном и другом обеспечении системы, которые должны быть учтены;
- понимание разработчиками поставленных перед ними задач меняется.

Под *управлением требованиями* понимают все действия, направленные на поддержание целостности, точности и актуальности спецификации требований в процессе разработки программной системы.

К действиям по управлению требованиями относятся:

- определение основной (базовой) версии спецификации требований для конкретной версии продукта;
- анализ предлагаемых изменений требований, оценка воздействия и стоимости каждого изменения до его принятия;
- включение одобренных изменений при помощи определенной процедуры;
- согласование плана проекта с требованиями;
- отслеживание отдельных требований от проектирования до кода приложения и его тестирования;
- отслеживание статуса требований и действий по изменению на протяжении всего проекта.

В организации (или в проекте) должны быть определены действия по управлению требованиями. Эти действия должны быть документированы и должны выполняться всеми участниками проекта. При разработке процесса нужно определить:

- методы и средства управления версиями спецификации и отдельных требований;
- процесс разработки, согласования, экспертизы и утверждения базовой версии;
- процесс присвоения, контроля и изменения статуса требования;
- процесс передачи новых требований и изменений существующих требований заинтересованным лицам;
- методы анализа влияния и стоимости внесения изменения.

Кроме этого описание процесса должно содержать определение участников проекта, ответственных за выполнение каждой конкретной задачи.

Минимальной единицей управления в спецификации требований является отдельное требование, поэтому вопрос идентификации требования достаточно важен.

Форма представления требования может быть различной (текстовая, графическая и т.д.), поэтому для идентификации требования обычно используют связанный с ним набор атрибутов. Атрибутами могут быть: дата создания требования, номер текущей версии требования, номер версии продукта, для которой предназначено требование, автор требования, ответственный за реализацию требования, состояние требования, происхождение и обоснование требования, подсистема, для которой предназначено требование и т.д. Главное при выборе атрибутов, чтобы они однозначно идентифицировали требование и его состояние.

В процессе выполнения проекта требование, обычно, изменяет свое *состояние* от начального («предложено»), до конечного, например, «реализовано». Состояния требований позволяет оценить степень готовности проекта. Рекомендуются использовать состояния требования, приведенные в табл. 1.

Таблица 1. Состояния требования

Состояние	Определение
Предложено (proposed)	Требование запрошено авторизированным источником
Одобрено (approved)	Требование проанализировано, его влияние на проект просчитано, и оно было размещено в базовой версии определенной версии продукта. Ключевые заинтересованные в проекте лица согласились с этим требованием, а разработчики обязались его реализовать

Реализовано (implemented)	Код, реализующий требование разработан, написан и протестирован. Требование отслежено до соответствующих элементов дизайна и кода
Проверено (verified)	Корректное функционирование реализованного требования подтверждено в соответствующем продукте. Требование отслежено до соответствующих вариантов тестирования. Теперь требование считается выполненным
Удалено (deleted)	Утвержденное требование удалено из базовой версии. Следует зафиксировать причины и лицо, принявшее это решение
Отклонено (rejected)	Требование предложено, но не запланировано для реализации ни в одной из будущих версий. Следует зафиксировать причины и лицо, принявшее это решение

В процессе управления требованиями должны быть определены лица, которые могут изменить состояние требования. Управление статусом позволяет численно определить степень готовности проекта, считая, например, что основная часть работы закончена, если все требования имеют состояние «Проверено» или «Удалено».
После разработки, согласования и утверждения спецификация требований становится основным документом в проектировании системы (версии системы). Изменения в этот документ разрешается вносить только через определенный в организации (или проекте) процесс внесения изменений.
Диаграмма состояний для типового процесса внесения изменений в спецификацию приведена на рис. 6.

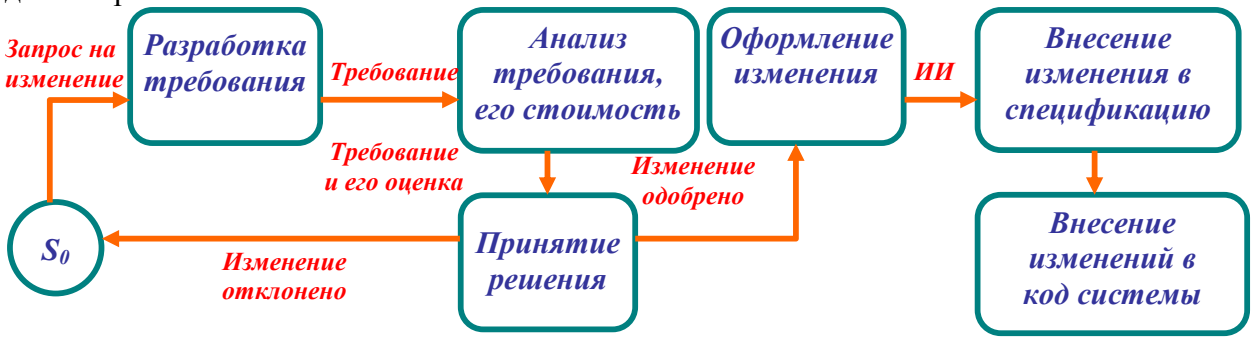


Рис. 6. Процесс внесения изменений в спецификацию требований

2.4. Архитектурное и детальное проектирование

Существуют такие типы проектов по разработке программного обеспечения, которые не нуждаются в фазе проектирования: после того, как требования определены, можно сразу приступать к реализации и кодированию. Однако такая возможность встречается очень и очень редко. Для того, чтобы действительно можно было безнаказанно пропустить фазу проектирования, необходимо наличие целого ряда благоприятных обстоятельств: либо это чрезвычайно простой («учебный») проект, либо имеется высоко квалифицированная команда разработчиков, которая прежде выполняла аналогичные проекты многократно.
Поясним это утверждение аналогией со строительством.[14] В двух случаях *можно* обойтись без разработки строительных чертежей: либо мы строим собачью конуру, либо мы строим типовой дом по готовому типовому проекту, собирая его из готовых стандартных блоков. Во всех остальных случаях профессиональным строителям и в голову не придет начинать строительство, не разработав предварительно детальные строительные чертежи, сметы, планы и графики.
Между тем с сожалением приходится отметить, что очень многие начинающие программисты пытаются «проскочить» или предельно сократить фазу проектирования. Обычно это мотивируют необходимостью сокращения сроков и трудозатрат. Это принципиальная

[14] Эта аналогия является излюбленной у многих авторитетных авторов книг по технологии программирования.

ошибка: сокращение сроков и трудозатрат на проектирование не только не сократит суммарных сроков и трудозатрат на разработку, а, наоборот, в лучшем случае существенно увеличит их, а в худшем случае приведет к краху проекта.
В настоящее время всю фазу проектирования обычно подразделяют на две части: архитектурное проектирование и детальное проектирование.

2.4.1. Архитектурное проектирование

Архитектура программы – это очень и в то же время трудно определимое понятие технологии программирования. Приведем определение из Википедии.[15]

> *Архитектура программного обеспечения* – это представление системы программного обеспечения, дающее информацию о компонентах составляющих систему, о взаимосвязях между этими компонентами и правилах, регламентирующих эти взаимосвязи, которое предназначено для эффективной разработки проекта такой системы.

Проиллюстрируем определение аналогией со строительством. Допустим, нам нужно спроектировать и построить небольшой загородный дом на определенном земельном участке. В таком случае решение таких вопросов, как определение количества жилых комнат и спален, вопрос о подключении к инженерным сетям (электричество, водопровод и так далее), вопрос о сезонности эксплуатации (только летом, разовые визиты зимой, круглый год) – это вопросы, относящиеся к фазе определения требований. В то же время решение таких вопросов, как выбор места на участке, общая площадь, этажность – это архитектурное проектирование.
За последнее время вопросам архитектурного проектирования в технологии программирования было уделено особенно много внимания. Были предложены и реализованы на практике многочисленные типовые архитектуры. Шоу и Гарлан классифицировали архитектуры программного обеспечения с точки зрения практики [2]. Другими словами, они собрали вместе образцы программного обеспечения для различных архитектур. Их классификация, немного адаптированная, показана в табл. 2.

Таблица 2. Типы архитектур (по классификации Гарлана и Шоу)

Категория	Подкатегория	Основная идея
Потоки данных (data flows)	Последовательность пакетов Каналы и фильтры	Независимые процессы обработки данных запускаются, когда им на вход поступают данные
Независимые компоненты (independent components)	Параллельные взаимодействующие процессы Клиент-серверные системы Системы, управляемые событиями	Независимые компоненты взаимодействуют, обмениваясь сообщениями
Виртуальные машины (virtual machines)	Системы на основе правил Интерпретаторы	Определяется предметно-ориентированный внутренний язык и процессор этого языка
Репозиторные архитектуры (repository architectures)	Гипертекстовые системы Базы данных Доски объявлений	Структура приложения определяется структурой хранимых данных

Приведенная в табл. 2 классификация архитектур довольно поверхностна. На практике обычно используют комбинацию нескольких архитектурных идей, настраивая их с учетом особенностей проекта.

[15] http://ru.wikipedia.org/

2.4.2. Детальное проектирование

Продолжим аналогию со строительством. Допустим, архитектура нашего загородного дома выбрана. Теперь нужно решить такие вопросы, как выбор материалов для фундамента, несущих конструкций и кровли, количество, размеры и расположение окон и дверей, способ и место подключения к инженерным сетям и так далее, и так далее – это и есть детальное проектирование.

Фаза детального проектирования обычно предусматривает применение большого числа различных средств, инструментов и приемов. Как правило, здесь наблюдается разнообразия больше, чем в фазе кодирования и реализации. При кодировании и реализации возможностей для выбора и принятия решений не так много – все они предопределены выбранной системой программирования. Фаза детального проектирования, напротив, требует постоянного выбора среди множества возможных альтернатив. Разработанные требования и выбранная архитектура полностью предписывают, *что* нужно сделать, но в очень малой степени являются подсказкой при поиске ответа на вопрос *как* это можно сделать. Другими словами, фаза детального проектирования – самая творческая часть процесса программирования. Именно здесь присутствуют наибольшие возможности для того, чтобы найти новое элегантное решение или совершить грубую дорогостоящую ошибку. Дать обзор всех приемов детального проектирования затруднительно. Мы приведем один пример описания процедуры детального проектирования, основанный на использовании объектно-ориентированного подхода и унифицированного языка моделирования UML (рис. 7). Элементы этой блок схемы указывают, какие артефакты необходимо разработать (или выбрать уже готовые!) на каждом шаге детального проектирования.

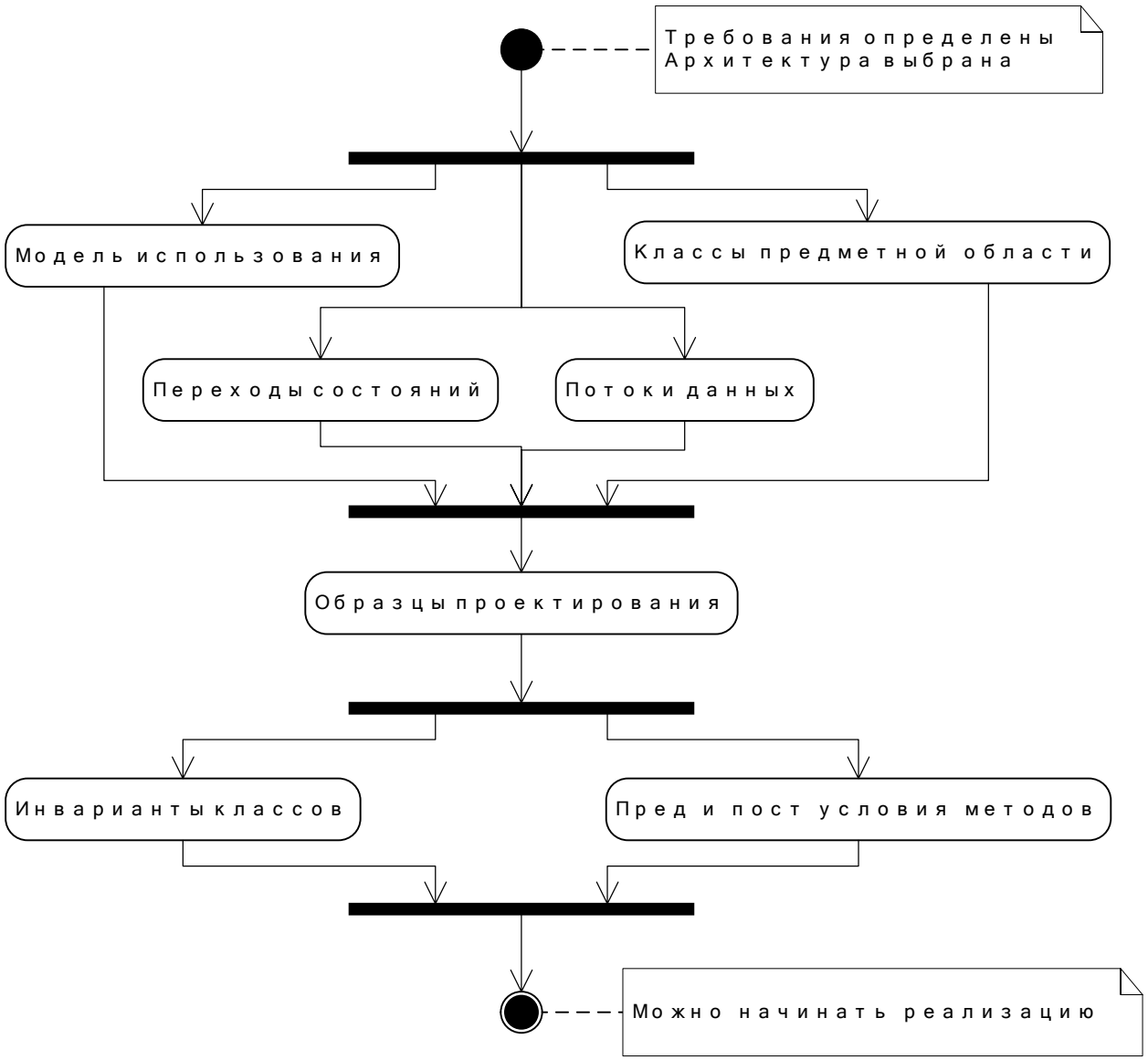


Рис. 7. Процесс детального проектирования

2.5. Реализация и кодирование

Написание программ на скорую руку дает быстрый, но сомнительный результат; дисциплинированный подход, напротив, обеспечивает лучшее качество с меньшими временными затратами.

В этом разделе не предпринимается попытка охватить все аспекты программирования в духе таких книг, как [4]. Вместо этого мы обсудим некоторые принципы кодирования и дадим рекомендации, которые относятся в большей степени к компаниям, разрабатывающим программные приложения, нежели к индивидуальным кодировщикам.

Реализацией (implementation) или *кодированием* (coding) называется составление текста программы на языке программирования в соответствии с детальным проектом, архитектурой и требованиями.

Термин «модуль» относится к самым мелким частям реализации, которые можно поддерживать отдельно: это может быть отдельный метод или класс. Мы будем считать методы модулями самого низкого уровня.

Целью реализации является удовлетворение требований способом, определенным в детальном проектировании. Детального проекта должно быть достаточно, поскольку это документ, на котором основывается программирование. Однако часто программист исследу-

ет параллельно все предшествующие документы (требования, архитектуру), что помогает сгладить несогласованность этих документов.

Инженерным анализом программы[16] (reverse engineering) называется исследование и обработка текста программ с целью восстановления модели этой программы.

Когда модуль реализован, можно использовать инструменты инженерного анализа для повторной генерации аспектов детального проектирования.
Можно посоветовать применять инженерный анализ сразу после начальной реализации, поскольку исходный код часто является более адекватным, чем нижний уровень детального проектирования. Это также может оказаться полезным для унаследованного кода, который плохо документирован.
Вообще говоря, инженерный анализ следует использовать только тогда, когда он действительно необходим. Автор наблюдал много ситуаций, когда инженерный анализ использовался для скрытия недостатков фазы проектирования. Другими словами, программисты преждевременно «ныряют» в написание кода, а затем задним числом создают «проект» из кода. Квалифицированных аудиторов таких «проектов» редко удается обмануть – отсутствие должного проектирования практически невозможно скрыть.
В настоящее время разработка программного обеспечения является интенсивно развивающейся отраслью промышленности. Время, когда программирование было уделом отдельных талантливых одиночек, которые были единственными читателями своих программ, закончилось уже лет 30 назад. Сегодня крупные программные проекты разрабатываются большими коллективами, причем развитие и сопровождение проекта часто длятся годами. За это время успевает обновиться коллектив разработчиков, нередки случаи, когда инициаторы проекта к моменту его завершения могут отсутствовать не только в организации, но и в стране.
В этих условиях естественным является требование унификации стиля программирования. Код, написанный различными программистами, должен читаться как единое целое и быть понятен не только его авторам. Отсюда следует, что программа должна быть рационально написана и хорошо прокомментирована. Отметим, ничто так не раздражает программиста–профессионала, как небрежно написанный и код без комментариев. Так же как невозможно представить инженерный чертеж, оформленный без учета требований стандартов, так и программа должна иметь унифицированный вид. Отсюда следует необходимость составления, постоянной актуализации и строго исполнения стандартов кодирования.

Стандарт кодирования (coding standard) – сборник корпоративных или проектных правил и рекомендаций по составлению и оформлению текстов программ.

2.6. Тестирование и верификация

Тестирование — важнейшая фаза процесса разработки. В общем случае можно дать следующее определение.

Тестирование (testing) — это проверка соответствия требованиям.

В этом смысле тестированию подлежит не только программный код (это само собой разумеется) но и все артефакты процесса разработки.

2.6.1. Процесс контроля качества

Ответственность за качество артефакта в первую очередь несет человек, создающий этот артефакт. Но, как бы то ни было, «один в поле не воин». Всем требуется сторонний взгляд на выполняемую работу. Взгляд со стороны необходим, чтобы избежать недальновидности, нереалистичной самооценки и застоя. Процесс контроля качества является также об-

[16] В литературе часто можно встретить неудачный термин «обратное проектирование».

щественной обязанностью. Каждая часть работы, выполненная инженером, должна быть детально проверена по крайней мере одним человеком, желательно независимым от автора работы.

В дополнение к ответственности индивидуальных разработчиков, многие организации определили процесс раздельной систематической и полной проверки всех артефактов— *контроль качества*. В функции контроля качества входят проверки, инспектирование и тестирование. Контроль качества должен начинаться вместе с запуском каждого проекта (рис. 8). Лучше всего привлекать контроль качества и для проверки правильности используемого процесса и актуальности документации.

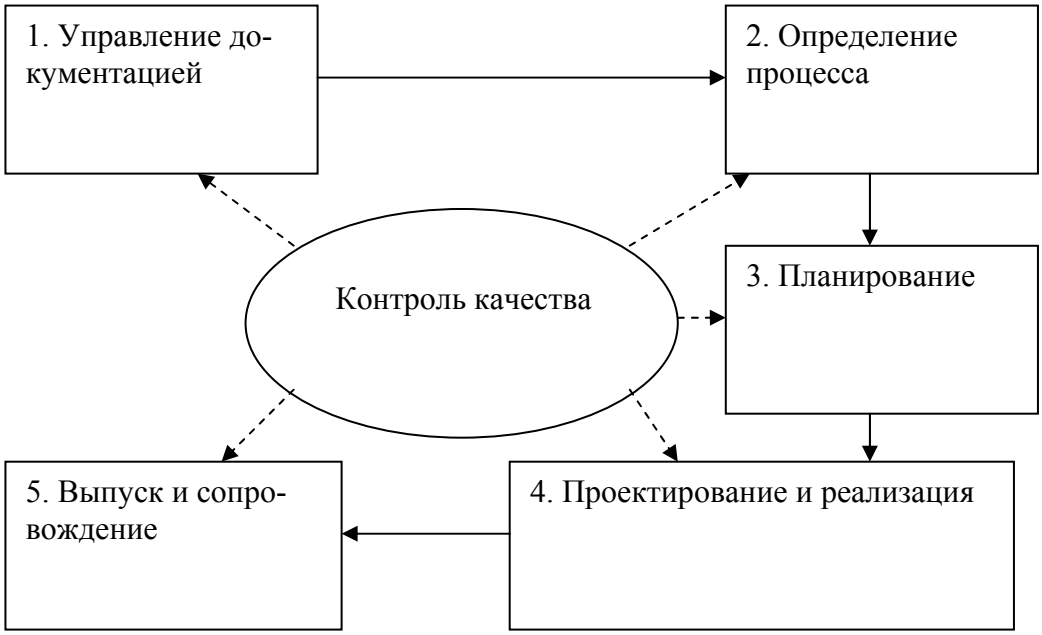


Рис. 8. Осуществление контроля качества

2.6.2. Методы «белого ящика» и «черного ящика»

В случае контроля качества методом *«черного ящика»* приложение (или какая-либо его законченная часть) анализируется как целое. Этот метод используется для проверки того, что приложение (его часть) отвечает предъявляемым требованиям. Контроль качества методом *«белого* (*стеклянного*) *ящика»* осуществляется на уровне компонентов, из которых построено тестируемое приложение (его часть).

Чаще всего о методах «черного» и «белого ящика» говорят в контексте тестирования. Однако эти методы применимы и к другим способам контроля качества. Метод «белого ящика» основан на рассмотрении анализируемого артефакта с точки зрения его структуры, формы и назначения; здесь применяются формальные методы и рассматриваемое в следующем разделе инспектирование. Метод «черного ящика» задается вопросом «Обладает ли построенный объект требуемым поведением?». При этом для тестирования по методу «черного ящика» достаточно иметь набор пар «вход–выход», чтобы можно было проверить правильность поведения.

2.6.3. Инспектирование и обзоры

Инспектирование — это техника «белого ящика» для обеспечения качества. Инспектирование состоит в проверке частей проекта (требований, результатов проектирования, программного кода и т. п.) на наличие дефектов. Инспектирования проводит группа коллег автора работы. Мы советуем начинать инспектирование очень рано, так как оно должно начинаться вместе с появлением первой документации по проекту. Поскольку инспектирование изначально было предложено для улучшения качества программного кода, то его

часто называют *инспектированием кода*, однако было показано, что инспектирование достигает наилучших результатов, если начинается как можно раньше, еще задолго до появления текста программ.

Принцип инспектирования может быть обобщен четырьмя правилами.

- **Вскрытие дефектов**. Из инспектирования намеренно исключается исправление дефектов. Процесс исправления предоставлен автору. Во время инспектирования ни минуты времени не должно быть потрачено на обсуждение способов исправления дефекта. Все обсуждения должны проходить после окончания инспектирования.
- **Участие коллег**. Инспектирование проводится внутри группы разработчиков программного обеспечения и не предполагает вовлечения отношений начальник — подчиненный. Инспектируется текущая работа, а не способности ее автора. Автор несет ответственность только за конечный продукт, тогда как инспектирование проводится до того, как работа сдана. Однако работа, представленная автором для инспектирования, должна быть лучшим вариантом, но ни в коем случае не черновиком. Было бы пустой тратой ресурсов группы искать, находить и описывать дефекты, которые автор, приложив некоторые усилия, в состоянии найти сам.
- **Распределение ролей**. Каждый из участников проекта берет на себя одну из следующих ролей. При нехватке кадров один человек может выполнять сразу две роли. Обычно ведущий может одновременно быть и секретарем. Однако для достижения беспристрастной проверки автор не должен выполнять никаких других ролей.
    - *Ведущий* ответственен за правильное проведение инспектирования. Ведущий также является инспектирующим.
    - *Автор* несет ответственность за свою работу и за исправление всех найденных в ней дефектов. Автор является одновременно и инспектирующим.
    - *Секретарь* отвечает за учет описания и классификацию дефектов, как это принято в команде. Секретарь также участвует в инспектировании.
- **Тщательная подготовка**. Участникам инспектирования необходимо подготовиться к нему так же детально, как и самому автору. Инспектирования не являются просто посиделками, докладами руководству или образовательными семинарами. Инспектирующие должны работать на том же уровне детализации, что и автор.

В различных источниках отмечается, что инспектирование, несмотря на большие затраты времени экспертов и дороговизну проведения, вполне окупает себя. Если начать подсчитывать стоимость инспектирования, то можно схватиться за голову, однако всегда следует сравнивать эту стоимость со стоимостью пропущенных дефектов. В конечном итоге, все дефекты должны быть найдены и исправлены. Если это не делается вскорости после того, как дефект был внесен, то стоимость исправления катастрофически возрастает.

Помимо инспектирования, многие организации применяют обзоры.

Обзор — это собрание, на котором обсуждается как уже завершенная, так и текущая работа.

Примером тому является обзор, на котором может обсуждаться альтернативная архитектура приложения. Хотя обзоры являются неотъемлемой частью проекта, они не требуют такой детальной подготовки, как инспектирование. К тому же участникам обзоров, в отличие от ситуации с инспектированием, не надо играть никаких ролей. Тем не менее, обзоры необходимо проводить, причем регулярно с участием максимального возможно чис-

ла разработчиков. Обзоры задают «пульс» проекта, подтверждают его жизнеспособность и служат фактором положительной мотивации персонала.
Регулярные обзоры — это самое надежное средство добиться того, чтобы все были «в курсе» и могли избегать ошибок, порождаемых непониманием общего контекста проекта.

2.6.4. Цели тестирования

Невозможно протестировать программу абсолютно во всех аспектах, поскольку число вариантов работы нетривиальной компьютерной программы может быть неограниченным.

Тестирование не может доказать *отсутствия* ошибок в программе. Тестирование может только показать *присутствие* ошибок.

Тестирование часто неправильно воспринимается как процесс подтверждения корректности кода, что можно выразить таким высказыванием: «Протестируй это, чтобы убедиться, что тут все правильно». Главная цель тестирования далека от подтверждения корректности. Цель тестирования не в том, чтобы показать удовлетворительную работу программы, а в том, чтобы четко определить, в чем работа программы неудовлетворительна!
Тестирование требует значительных затрат, и нужно стараться получить от этих затрат максимальную отдачу. Для данной тестируемой программы, чем больше дефектов будет найдено на каждый человеко-час тестирования, тем выше выигрыш от вложений в тестирование.

Целью тестирования является обнаружение как можно большего числа дефектов, как можно более серьезных и как можно раньше.

Согласно современным представлениям, тестирование оценивается более чем половиной времени, затрачиваемого на проект. Наградой за нахождение дефекта на ранней стадии процесса является по крайней мере десятикратная экономия по сравнению с обнаружением этого же дефекта на этапе интеграции или, еще хуже, после отправки заказчику. Следовательно, мы должны тестировать *рано* и *часто*.

2.6.5. Верификация, валидация и системное тестирование

Верификация позволяет определить, правильно ли мы создаем приложение.

Другими словами, действительно ли мы на текущей фазе создаем именно те артефакты, что были специфицированы? Это проверяется с помощью инспектирования, обзоров и модульного тестирования.

Валидация позволяет выяснить, правильный ли результат у нас получается.

Другими словами, удовлетворяет ли наш продукт требованиям? Это проверяется с помощью *системного тестирования*.

Системное тестирование – это тестирование все системы в целом.

Существует несколько видов системного тестирования.

- *Контекстное модульное тестирование*. После того как система собрана, становится возможным повторно протестировать модули (например, пакеты) в контексте системы.
- *Тестирование интерфейсов* подразумевает повторную валидацию интерфейсов между модулями.
- Цель *регрессионного тестирования* заключается в проверке того, что добавления к системе не уменьшили ее возможностей. Другими словами, при добавлении в систему нового модуля, следует сначала убедиться, что все старые модули не потеряли работоспособность. Только когда новый модуль

прошел регрессионное тестирование, можно тестировать его работу в контексте системы.

- *Нагрузочное тестирование*. Системное тестирование также валидирует требования, как функциональные, так и нефункциональные. Нефункциональные требования включают в себя требования к рабочим характеристикам, таким как скорость работы и использование ресурсов.
- *Тестирование удобства и простоты использования* валидирует приемлемость программы для ее конечных пользователей.
- *Тестирование инсталляции* выполняется при установке программы на целевых платформах.
- *Приемосдаточные испытания* выполняются клиентом для валидации приемлемости программы.

2.7. Сопровождение и продолжающаяся разработка

Сопровождение программного продукта включает в себя все действия, выполняемые с приложением после поставки продукта заказчику.

Сопровождение программы – это процесс изменения программной системы или компонента после поставки с целью исправления ошибок, повышения производительности или иных параметров, а также для адаптации к изменившимся условиям.

По различным оценкам сопровождение программы составляет от 40% до 90% стоимости всего жизненного цикла приложения. Можно возразить, что в указанное выше определение сопровождения включены усовершенствования, которые лучше было бы считать продолжающейся разработкой.

Продолжающаяся разработка – это разработка новых и переработка имеющихся функций программы в процессе эксплуатации.

В любом случае объем работ по сопровождению программ обычно достаточно велик.
Было подмечено любая программа, адаптацией которой к изменяющимся условиям никто не занимается, с течением времени неизбежно теряет свою ценность. Другими словами, если приложение остается неизменным, польза от него постепенно убывает.
Чтобы представить, какие проблемы возникают при сопровождении программ, вообразите приложение настолько большое, что ни один человек не может знать всех его функций и особенностей. Большую проблему в сопровождении таких программ создает эффект «ряби», возникающей при внесении изменений. Когда множество относительно безвредных изменений производятся одновременно в «удаленных уголках» больших приложений, эти изменения могут приводить к повреждению приложения как целого.
Можно упорядочить проблемы, связанные с сопровождением программ, разделив их на три категории.

- Проблемы управления: трудность выявления прибыли на инвестированный капитал.
- Проблемы обработки: для обслуживания потока запросов на сопровождение требуется жесткая координация.
- Проблемы технического характера: трудность учета всех результатов изменений; высокая стоимость тестирования по сравнению с пользой от отдельных изменений.

Следует различать работы по сопровождению, направленные на *устранение дефектов* (fixing) и на *усовершенствование* (enhancing) приложения. Различные исследования показали, что 60–80% работ обычно относится к усовершенствованию приложения, а не к исправлению его недостатков.
Последовательность обработки запросов на сопровождение состоит из анализа, проектирования, реализации и тестирования, точно так же, как и обычная разработка. Существен-

ным отличием является необходимость анализа влияния изменений на артефакты. Согласно исследованию Вейсса, 19% дефектов в приложениях образуются на этапе определения требований, 52% — на этапе проектирования и лишь 7% — в процессе программирования. Многие другие авторы утверждают, что доля дефектов, вызванных неправильной формулировкой требований, должна быть еще выше и составляет до 30%.

В то же время практика показывает, что стоимости устранения дефектов, допущенных при разработке требований, при проектировании и при реализации относятся как 10 : 3 : 1.

Это эффект нетрудно объяснить: устранение ошибки в требованиях повлечет изменение *всех* артефактов проекта. Устранение ошибки кодирования потребует изменить лишь один программный модуль.

Тема 3. Модели процесса разработки

Трудности конструирования реальных приложений обусловлены их сложностью, и критическую роль в преодолении этой сложности играет сам процесс конструирования.
В настоящее время можно выделить три стратегии конструирования модели процесса [8].

- *Линейная стратегия*. Требования определены в начале разработки, и выпуск программного обеспечения производится один раз в конце разработки.
- *Итерационная стратегия*. Требования также определены в начале разработки, но выпуск версий программного обеспечения производится многократно по ходу разработки.
- *Эволюционная стратегия*. Выпуск версий программного обеспечения производится многократно по ходу разработки, но требования не определены в начале разработки, они определяются по ходу разработки.

Свойства указанных стратегий можно свести в табл. 3.

Таблица 3. Характеристики стратегий конструирования

Стратегия	Требования определены в начале?	Множество циклов конструирования?	Промежуточные результаты распространяются?
Линейная	Да	Нет	Нет
Итерационная	Да	Да	Да
Эволюционная	Нет	Да	Да

Далее рассматриваются некоторые известные модели, представляющие каждую из упомянутых стратегий.

3.1. Водопадные и конвейерные модели

В модели *водопада* (иногда называемой моделью *конвейера*) процесс разработки (жизненный цикл программного продукта) делится на фазы, которые последовательно сменяют друг друга (рис. 9).

- Анализ требований состоит в сборе требований к продукту. Результатом анализа, как правило, является некоторый текст или модель использования.
- Проектирование описывает внутреннюю структуру продукта. Обычно такое описание дается в форме диаграмм и текстов.
- Реализация — это программирование. Результатом реализации является программный код всех уровней, будь то код, генерируемый высокоуровневой системой программирования, компилятором языка четвертого поколения или какой-либо другой.
- Тестирование — это процесс сборки всего продукта из отдельных частей и проверки того, что требования удовлетворены.

Замечание

> Модель водопада (конвейера) является исторически первой осознанно использованной моделью процесса разработки программного обеспечения. Многие очень крупные проекты были успешно проведены с использованием этой модели.

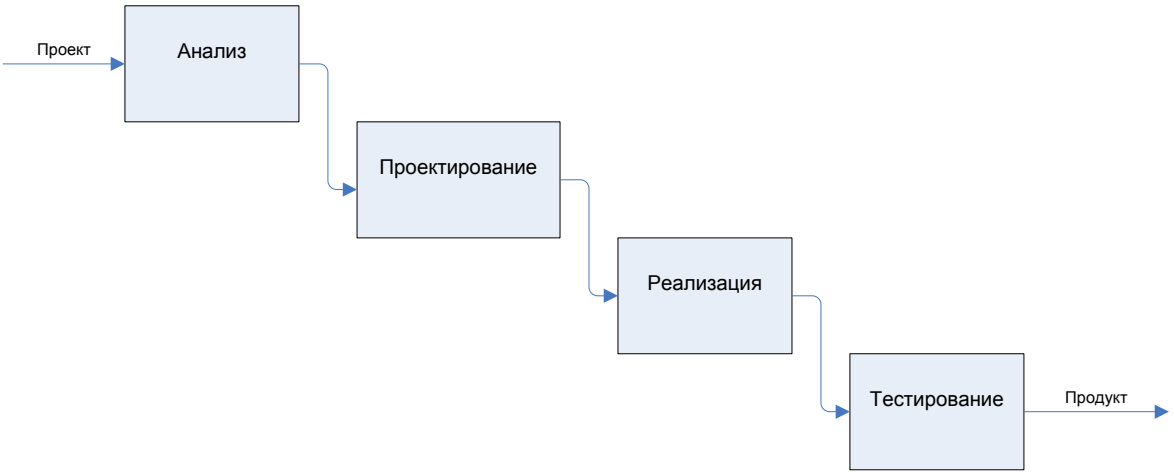


Рис. 9. Водопадная модель процесса разработки

В действительности перечисленные фазы не следуют строго последовательно друг за другом, а частично перекрываются. На практике любую из фаз можно начинать до того, как будет полностью завершена предыдущая. Но важнейшей особенностью водопадной модели является то обстоятельство, что завершив фазу, мы уже больше к ней не возвращаемся. Таким образом, водопадная модель является типичным представителем линейной стратегии.

3.2. Спиральные и инкрементные модели

В *инкрементных* моделях процесс делится на витки, или итерации, каждая из которых в свою очередь делиться делится на фазы (например, анализ, планирование, разработка, стабилизация). Каждая фаза кончается вехой (концепция, спецификации, код, выпуск). После выпуска (release) раскручивается очередной виток *спирали* (см. рис. 10).
Таким образом, на каждой итерации происходит выпуск продукта. Последовательные выпуски отличаются некоторым приращением – реализованными функциями или изменением других свойств. Такое приращение иногда называют *инкрементом* (от английского слова increment), что и дало название этой группе моделей.
Наиболее известной конкретной спиральной моделью процесса является модель Microsoft Solution Framework (MSF), в адаптированном виде приведенная на рис. 10.

Замечание

> Модель конвейера, также как и спиральная модель, может использовать вехи. Характерным для модели конвейера (водопада) является безвозвратный (не итеративный) характер процесса, в то время как в спиральных и инкрементных моделях мы вновь и вновь повторяем последовательность фаз. Таким образом, рассматриваемые модели относятся к итерационной стратегии.

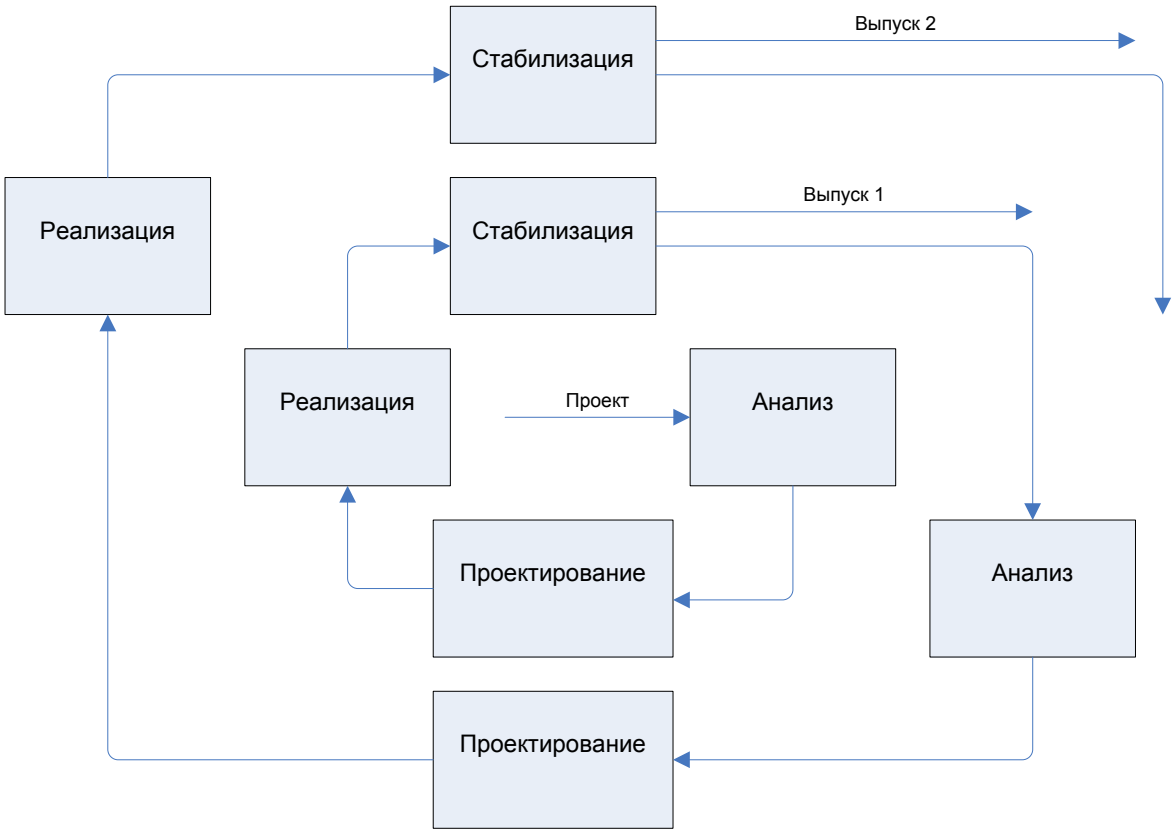


Рис. 10. Модель спирального процесса

Иногда представляется возможным понемногу продвигать проект вперед при практически непрерывном процессе. Такая модель процесса особенно полезна на поздних стадиях проекта, когда продукт находится на сопровождении или когда разрабатываемый продукт очень схож с созданным ранее. Например, процесс, используемый в некоторых отделениях корпорации Microsoft, предусматривает обновление программного кода и документации ежедневно к конкретному времени для интеграции и ночного тестирования. Другие организации используют для этого недельные циклы. Для поддержания соответствующего уровня инкрементальной разработки необходимо иметь четко установленную архитектуру проекта и исключительно синхронизированную систему документации. Для организации инкрементальной разработки обычно выбирается характерный временной интервал, например неделя. Затем в течение этого интервала происходит обновление исходного проекта (документации, набора тестов, программного кода и т. д.).

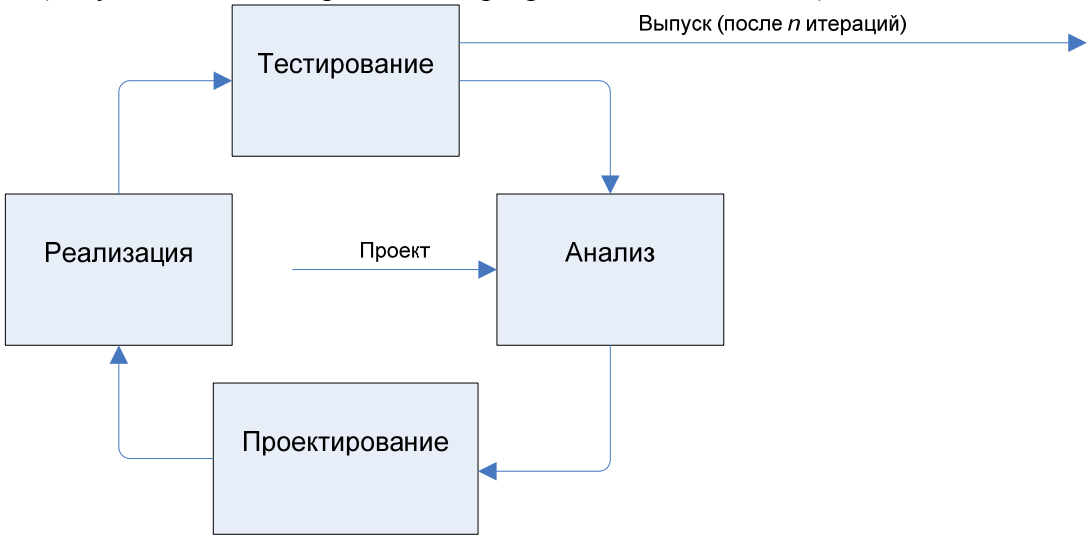


Рис. 11. Модель инкрементного процесса

Наиболее известной инкрементной моделью является Унифицированный процесс (Rational Unified Process).

3.3. Гибкие модели процесса разработки

Самой яркой моделью гибкой разработки сейчас, видимо является *экстремальное программирование*, предложенное Кентом Беком.

Кент Бек (Kent Beck) – пионер экстремального программирования. Практикующий разработчик, менеджер и автор нескольких книг по экстремальному программированию, применению образцов проектирования и гибким процессам разработки.

В основу этой модели, так же как и других эволюционных моделей положено следующее наблюдение: в момент постановки задачи заказчик очень часто не имеет четкого представления о функциональности заказанного программного обеспечения, в результате чего образ программного проекта в процессе разработки постоянно меняется в его сознании. Часто из-за длительных циклов разработки требования, предъявляемые к программному продукту к моменту его готовности, уже не совпадают с заданными изначально (рис. 12).

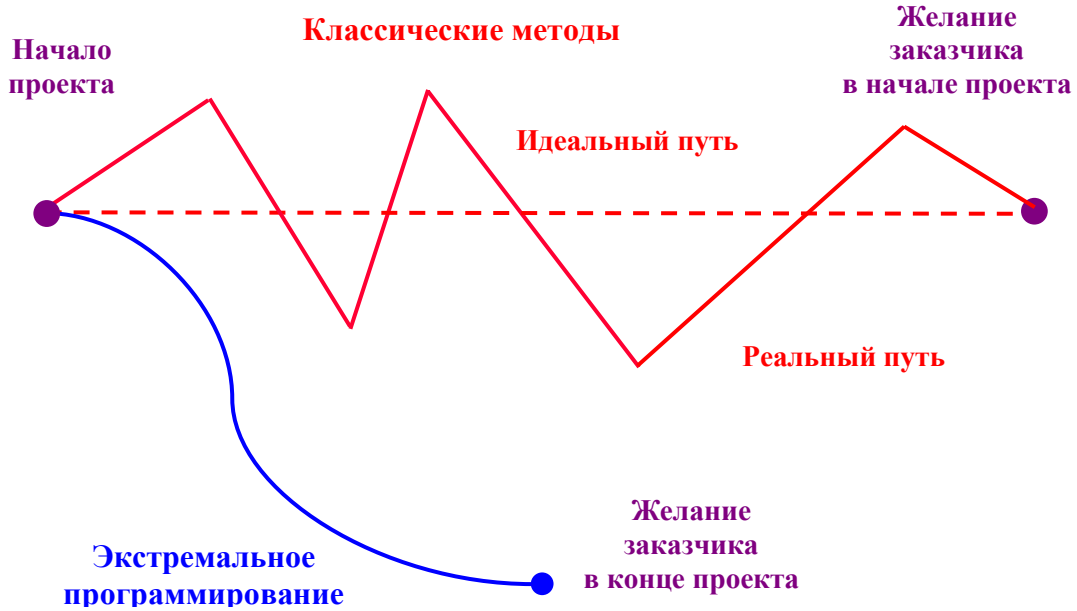


Рис. 12. Изменение требований в процессе разработки программного продукта

Экстремальное программирование не является жестко фиксированным процессом разработки. Это лишь набор простых методик (практик), каждая из которых, будучи использована отдельно, вряд ли даст ощутимый эффект, однако, используемые вместе, они позволяют значительно повысить эффективность процесса разработки программных изделий.
Вообще говоря, экстремальное программирование использует многие известные ранее методики, но применяет из максимально интенсивным, экстремальным образом.[17] Ключевыми методиками экстремального программирования принято считать следующие.

- *Динамическое планирование* (planning game) – быстро сформировать приблизительный план работы и постоянно обновлять его по мере того, как условия задачи становятся все более четкими. Если со временем план перестает быть актуальным, он должен быть обновлен.
- *Частые выпуски* (small releases) – самая первая упрощенная версия быстро сдается заказчику, после чего через относительно короткие промежутки времени (неделя–две недели) происходит выпуск новых версий.
- *Архитектурная метафора* (system metaphor) –простая и понятная заказчику концепция системы.
- *Простое проектирование* (simple design) – в каждый момент времени система должна быть спроектирована так просто, как это возможно, так как заказчик может изменить функциональность.
- *Опережающее тестирование* (testing) – для любой программы должны существовать автоматические тесты. Любая функция системы, для которой нет тестов, считается несуществующей. Не должно существовать кода без тестов.
- *Рефакторинг* (refactoring) – постоянное улучшение качества кода с сохранением функциональности.
- *Парное программирование* (pair programming) – весь код пишется двумя программистами, работающими на одном компьютере.
- *Коллективное владение кодом* (collective ownership) – любой разработчик может улучшать любой код системы в любое время.
- *Непрерывная интеграция* (continuous integration) – как только код написан и протестирован, он интегрируется в общую систему. После интеграции код обновляется у всех разработчиков, т.е. в любой момент времени все разработчики работают с актуальной версией кода.
- *40-часовая рабочая неделя* (no overtime) – внеурочная работа способствует ухудшению психологического климата в команде. Невозможность планирования рабочего времени отрицательно сказывается на результатах работы.
- *Доступный заказчик* (on-site customer) – в команде все время должен находиться представитель заказчика, действительно готовый отвечать на вопросы.
- *Стандарты кодирования* (coding standards) – весь код должен быть написан в едином стиле.

В экстремальном программировании применяются и многие другие методики, но по крайней мере перечисленные должны применяться совместно и интенсивно, иначе эффект не будет достигнут.

3.4. Конструирование модели процесса

На практике программирующие организации редко применяют приведенные абстрактные модели процессов в «чистом виде». Как правило, используя известные модели и учитывая конкретные особенности и условия, организация создает специфические модели «под себя», которые наилучшим образом учитывают особенности организации.

[17] Откуда и происходит название модели.

Основными «методами» конструирования конкретных процессов разработки являются, конечно, здравый смысл и опыт. Невозможно дать в учебнике короткий набор рекомендаций, заменяющих эти важнейшие качества. Но можно привести *пример* конструирования конкретного процесса.

Вообще говоря, есть много общего в процессе разработки и в процессе конструирования описания процесса разработки.[18] Действительно, ведь само описание процесса разработки, как мы показали ранее, фактически является алгоритмом, только исполнителем этого алгоритма является не компьютер, а программирующая организация. Следовательно, в процессе конструирования описания процесса разработки наблюдаются примерно те же фазы, что и в процессе разработки: выявление требований, выбор архитектуры, реализация и валидация. Применительно к конструированию процесса реализация подразумевает определение конкретных процедур (алгоритмов) по которым должен выполняться сконструированный процесс разработки.

Оставшаяся часть данного раздела до конца темы представляет собой пример сконструированного конкретного процесса разработки для некоторой гипотетической программирующей организации.

Замечание по конструированию

> Замечания, которые присутствуют далее в тексте описания сконструированного процесса, не являются частью сконструированного процесса, т.е. не являются частью примера. Эти замечания содержат объяснения и мотивацию того, почему было принято то или иное решение при конструировании.

3.4.1. Выявление требований к процессу

Рассмотрим гипотетическую организацию, разрабатывающую программное обеспечение на заказ. Допустим, организация выявила у себя следующие особенности.

- **Договорные отношения с заказчиками**. Организация проводит проекты по разработке программного обеспечения для сторонних заказчиков на договорных началах. Каждый проект регулируется отдельным договором. Комплект договорных документов соответствует сложившейся отечественной практике.
- **Фиксированные временные рамки проектов**. Основным типом проектов являются вертикальные приложения масштаба предприятия или компоненты таких приложений, для которых время окончания разработки и внедрения предопределено условиями договора. Продолжающаяся разработка не характерна, если она имеет место, то по новому договору, а значит в рамках другого проекта.
- **Новизна предметной области**. Проекты выполняются в различных предметных областях (разные типы предприятий, разные бизнес-процессы), поэтому часто содержат элементы существенной новизны для разработчиков. В этих условиях весьма велик технический риск получения неполной или неадекватной первой версии каждой системы.
- **Гарантированный конечный успех**. Для расширения и стабилизации круга заказчиков требуется гарантированный конечный успех каждого отдельного проекта, даже если это требует дополнительных внутренних расходов на проведение проекта.

Для такой организации целесообразно построить комбинированную модель процесса, и дифференцировать ее по типам проектов.

Сформулируем требования к конструируемой модели процесса.

- **Учет особенностей организации**. Процесс должен учитывать выявленные особенности организации и не должен им противоречить.

[18] Выражаясь терминами, принятыми в унифицированном языке моделирования UML, можно сказать, что процесс конструирования описания процесса разработки является мета процессом по отношению к процессу разработки.

- **Использование стандартных элементов**. Процесс должен быть описан в терминах, приведенных в предыдущих разделах и опираться на известные абстрактные модели процессов. Использование принципиально новой модели нежелательно.
- **Минимальное описание**. Процесс должен быть по возможности легким, он не должен содержать процедур, артефактов и регламентирующих документов, без которых можно обойтись.
- **Область применения**. Конструируемое описание должно описывать только процесс разработки программного обеспечения. Другие процессы, например, договорные отношения с заказчиком, описывать нежелательно.

Замечание по конструированию

> В число требований включены как свойства, которыми конструируемый процесс должен обладать, так и свойства, которыми он *не* должен обладать.

3.4.2. Используемые фазы, вехи и артефакты

Замечание по конструированию

> Целесообразно начать описание процесса с определения используемых понятий, подобно тому, как саму разработку целесообразно начинать с составления словаря предметной области.

В разрабатываемом процессе используются фазы, которые называются

- Анализ
- Проектирование
- Реализация
- Стабилизация
- Внедрение

Замечание по конструированию

> Названия первых четырех фаз совпадают с фазами процесса MSF (см. рис. 10) В модели процесса MSF имеются только четыре первых фазы по следующей причине. Фаза внедрения (deployment) не включается в процесс разработки и рассматривается в MSF как отдельный процесс. В случае рассматриваемой организации мы предположили, что разработка и внедрение регулируются, как правило, одним и тем же договором, поэтому внедрение считается частью процесса разработки.

Последовательность пяти фаз называется «виток». Фазы одного витка всегда выполняются в указанном порядке.

Замечание по конструированию

> Последовательность фаз в витке является *логической*, в том смысле, что последующие фазы зависят от предшествующих фаз. Это не означает, что фазы выполняются во времени строго одна за другой и следующая фаза может начаться только после окончания предыдущей. Напротив, фазы могут выполняться параллельно, и быть частично или полностью совмещенными по времени.

Каждая фаза заканчивается главной *вехой*, которая является характеристическим признаком фазы.

Кроме главных вех, которые являются видимыми извне процесса событиями, фазы могут иметь внутренние вехи, которые могут быть видимы и не видимы извне. Состав и порядок внутренних вех зависит от типа проекта и устанавливается при планировании процесса для конкретного проекта.

Замечание по конструированию

> Видимость извне означает, что отчуждаемые материалы вехи согласуются с заказчиком. Таким образом, все главные вехи и видимые извне внутренние вехи являются контрольными точками, привязанными ко времени планом-графиком, прилагаемым к Техническому заданию на проект. Далее в качестве синонима оборота "согласуемый с заказчиком…" используется термин "внешний".

3.4.2.1. Фаза «Анализ»

Фаза анализа является первой в каждом витке. Главная веха фазы анализа называется «утверждение концепции проекта» (или просто «концепция») и состоит в фиксации документа (серии документов) с общим названием "Концепция…". Концепция включает в себя следующие обязательные разделы:

1. Введение. Основание для разработки и статус проекта.
2. Постановка задачи. Общее описание предметной области проекта; формулировка проблемы, на решение которой направлен проект; краткий обзор альтернативных решений, если они есть; основные отличительные особенности проекта.
3. Цели проекта. Явное, лаконичное и эффективно проверяемое утверждение о целях проекта. Обычно проект имеет одну главную цель и несколько вспомогательных или промежуточных подцелей.
4. Категории пользователей. Классификация пользователей результатов проекта; перечисление их ожиданий по отношению к результатам проекта.
5. Используемые подходы. Перечисление и краткое описание методов, средств, приемов и мероприятий, которые предполагается использовать для достижения целей проекта.
6. Критерии успеха. Количественные эффективно вычислимые критерии, позволяющие оценить степень успешности проекта.
7. Типовые примеры. Несколько (1–3) неформальных примеров, иллюстрирующих использование ожидаемого решения.
8. Отчуждаемые материалы. Перечисление и краткое описание отчуждаемых материалов, которые составляют результат проекта.

Основное назначение Концепции состоит в том, чтобы явно сформулировать общее и согласованное понимание проблемы и путей ее решения, которое бы разделялось всеми участниками проекта: заказчиком, исполнителями (командой) и руководством организации. Промежуточными вехами на фазе анализа могут быть:

- Обследование. Отчет о проведенном обследовании бизнес-процессов, на которые направлен проект.
- Техническое задание. Формализованный документ, отражающий в стандартизованной форме содержание Концепции.

Замечание по реализации

> При описании артефакта (документа) на этапе проектирования процесса достаточно указать название, назначение и структуру документа. При практическом применении процесса потребуется подготовить соответствующие шаблоны, места хранения документов и т.д.

3.4.2.2. Фаза «Проектирование»

Проектирование является решающей фазой, определяющей общий успех проекта. Главной вехой фазы проектирования является утверждение спецификаций проекта. Спецификации представляют собой набор разнородных документов и других материалов, среди которых обязательными и важнейшими являются:

- **Функциональные спецификации**. Исчерпывающее и подробное описание функций решения. Для описания функциональных спецификаций могут использоваться разнообразные средства: естественный язык; формальные языки; компьютерные средства моделирования и др. Во всех случаях, когда это возможно, настоятельно рекомендуется применение средств моделирования, поддерживающих унифицированный язык моделирования (UML).

Замечание по конструированию

> Различаются внешние и внутренние спецификации проекта. Внешние спецификации согласуются с заказчиком, внутренние спецификации являются внутренними документами проек-

та. Все внешние спецификации обязательно создаются на фазе проектирования, некоторые внутренние спецификации могут быть перенесены на последующие фазы.

- **Календарный план-график**. Список фаз и частей фаз с указанием главных и промежуточных вех, привязанных к конкретным датам, исполнителей и отчуждаемых материалов каждой вехи. Настоятельно рекомендуется использовать для этой цели инструментальное средство Microsoft Project. Другие средства календарного планирования используются только в том случае, если заказчик возражает против использования Microsoft Project.

Замечание

Документ с аналогичным названием подготавливается также на первом этапе подготовки к проекту, прилагается к Договору и согласуется с заказчиком. План-график фазы проектирования может совпадать с календарным планом, который видит заказчик, если все вехи являются внешними и совпадают с этапами ТЗ. Но гораздо чаще календарный план-график является внутренним документом, содержащим детальную расшифровку фаз и вех с персоналиями. В этом случае внешний (менее детальный) план-график оформляется в соответствии с пожеланиями заказчика, а внутренний план-график ведется с помощью Microsoft Project.

Кроме указанных основных компонентов, в спецификации проекта могут входить:

- **Прототип**. Компьютерная модель приложения, если проект подразумевает разработку приложения. Прототип может быть полнофункциональным или только прототипом интерфейса, если приложение имеет интерфейс. Полнофункциональный прототип подменяет функциональные спецификации.
- **Внутренний язык**. Описание внутреннего языка командного типа рекомендуется для всех многофункциональных приложений, допускающих варьируемые сценарии использования. Для приложений, имеющих программный интерфейс (API), такое описание крайне желательно.
- **Дополнительные требования**. Перечень и описание количественных ограничений и/или специальных требований (например, по устойчивости системы защиты информации), которым должно удовлетворять решение. Дополнительные требования могут быть оформлены как дополнение к Концепции и/или Техническому заданию.
- **План тестирования**. План тестирования содержит набор тестов и описания порядка их применения с целью измерения таких важнейших характеристик разрабатываемого в рамках проекта приложения, как надежность, функциональная полнота и применимость. Допускается план тестирования не включать в Спецификации, а объявить первой промежуточной вехой фазы стабилизации.
- **План внедрения**. Описание процедур развертывания и демонстрации работоспособности разрабатываемого приложения. Кроме того, план внедрения может включать план обучения пользователей и технических специалистов заказчика и план-проспект пользовательской документации, если это предусматривается общим планом проекта. Допускается план внедрения не включать в Спецификации, а объявить первой промежуточной вехой фазы внедрения.

Каждый из компонентов спецификаций проекта может быть отдельной промежуточной вехой фазы проектирования. Функциональные спецификации, Календарный план-график и План внедрения являются внешними материалами, План тестирования, как правило, является внутренним материалом.

3.4.2.3. Фаза «Реализация»

Фаза реализации (кодирования) имеет основную веху, которая называется «код готов». Хотя на этой фазе создается основной отчуждаемый результат проекта (код приложения), доля фазы реализации составляет не более 20% всех ресурсов проекта. Веха «код готов» означает, что все специфицированные функции запрограммированы, и работоспособность приложения в целом продемонстрирована на нескольких примерах.

Замечание по конструированию

> Многочисленные исследования показывают, что чем крупнее проект, тем меньшую долю в нем занимает собственно кодирование. Указанная величина 20% является ориентировочным показателем для типичного проекта.

Промежуточные вехи фазы реализации зависят от типа проекта и используемого инструментального программного обеспечения. Они включают в себя:

- **Логический проект**. Модель объектов (служб) приложения. Спецификация интерфейсов услуг (методов и свойств), предоставляемых службами (объектами).
- **Схема базы данных**. Фиксация представления данных проекта. В случае использования стандартной реляционной СУБД — определение структуры таблиц и связей между ними.
- **Дизайн интерфейса**. Фиксация внешнего вида статической части графического интерфейса (форм и диалоговых окон). Фиксация состава команд меню и языка пользователя, если он предусматривается проектом.
- **Физический проект**. Упаковка служб в компоненты, описание протоколов взаимодействия компонент для распределенных приложений.

Если какие-то промежуточные вехи фазы реализации являются внешними (например, логический проект), то они относятся к внешним спецификациям и выполняются на стадии планирования.

Рекомендуется использование средств CASE, ориентированных на визуальное проектирование и автоматическую генерацию кода. Наиболее перспективным является использование средств, поддерживающих UML. В этом случае результатом для целого ряда вех (Функциональные спецификации, Логический проект, Схема базы данных, Физический проект и даже Код готов) является модель приложения (отдельные части этой модели), построенная с помощью используемого средства CASE.

3.4.2.4. Фаза «Стабилизация»

Фаза стабилизации (тестирования) заканчивается главной вехой, которая называется выпуск (release). Достижение этой вехи характеризуется наличием полного и подготовленного к тиражированию (копированию, передаче заказчику) комплекта отчуждаемых материалов проекта, которые были специфицированы на фазе планирования.

Основной деятельностью на фазе стабилизации являются тестирование и отладка, то есть выявление и устранение ошибок. Различаются следующие виды комплексного тестирования:

- **Тестирование устойчивости** (reliability). Целью этого вида тестирование является выявление не перехватываемых ошибок времени выполнения, т.е. тесты устойчивости нацелены на то, чтобы "сломать" приложение. Типичными приемами, применяемыми при тестировании устойчивости, являются ввод данных, выходящих за пределы области допустимых значений, нарушение порядка действий, предусмотренных сценарием, создание ситуаций, нарушающих количественные ограничения.
- **Тестирование функциональности** (functionality). Целью этого вида тестирования является проверка того, что для допустимых (правильных) входных данных получаются допустимые (правильные, корректные, удовлетворительные, соответствующие спецификациям) результаты. Ожидаемые правильные результаты для каждого теста функциональности должны быть получены заранее независимо от тестируемого приложения. Типичным приемом является ввод предельных (находящихся на границе области допустимых значений) данных.
- **Тестирование применимости** (usability). Целью этого вида тестирования является проверка того, что предусмотренные способы использования приложения являются удовлетворительными для пользователя при выполнении типичных сценариев работы. При тестировании применимости проверяются, например, следующие пара-

метры: реактивность, защищенность данных, способность к восстановлению после ошибки, понятность интерфейса и др.

Фаза стабилизации имеет промежуточные вехи, которые являются обязательными, если выпуск соответствующих материалов предусмотрен проектом.

- **Нет известных ошибок**. Все тесты, предусмотренные планом тестирования, не выявляют ошибок. Все найденные и зафиксированные в базе данных ошибки помечены как исправленные.
- **Исходные тексты готовы**. Все исходные тексты программ проверены на соответствие принятой дисциплине программирования (комментарии, имена объектов, структура текста). Некоторые советы, касающиеся хорошего стиля программирования, приведены ниже в разделе Дисциплина программирования.
- **Документация**. Эксплуатационная (пользовательская) документация (электронная и печатная) готова к тиражированию, т.е. проверена ответственным редактором, техническим редактором и корректором, корректура внесена и проведена сверка. Если заказчик согласен, то выпуск пользовательской документации может быть перенесен на фазу опытной эксплуатации.

Замечание по конструированию

> Техническая подготовка документации (так называемая предпечатная подготовка) и само тиражирование (если оно предусмотрено планом) в ответственных проектах может выполняться специалистами, не входящими в состав команды проекта, поэтому оно не упоминается в описании процесса.

3.4.2.5. Фаза «Внедрение»

Фаза внедрения является заключительной, ее главная веха называется «проект закончен». Эта веха характеризуется следующими основными признаками: разработанное решение используется заказчиком (пользователями) на практике; взаимоотношения с заказчиком урегулированы, результаты проекта документированы, измерены, архивированы и подготовлены для повторного использования.

Фаза опытной эксплуатации подобна фазе внедрения, но не является заключительной. Эти фазы различаются в части документов, описывающих удовлетворенность заказчика результатами проекта. В конце фазы внедрения все зафиксированные претензии и замечания заказчика удовлетворены. В конце фазы опытной эксплуатации претензии и замечания заказчика зафиксированы в форме протокола замечаний/разногласий.

На фазе внедрения промежуточные вехи могут устанавливаться заказчиком. Типичными вехами фазы внедрения являются следующие:

- **Приложение (решение) развернуто**. Разработанное приложение (решение) установлено у заказчика и проверена его работоспособность согласно программе и методике испытаний.
- **База данных загружена**. Эксплуатация решения ведется на реальных данных заказчика.
- **Пользователи обучены**. Пользователи работают с разработанным приложением без постоянного наблюдения разработчиков. Вмешательство и помощь со стороны разработчиков происходит только по запросу пользователей.

3.4.3. Выбор архитектуры процесса

В предлагаемой модели процесса в типичном случае полный процесс состоит ровно из двух витков, каждый из которых состоит из пяти фаз. Таким образом, каждая фаза выполняется дважды.

Замечание по конструированию

> Это обеспечивает итеративность разработки, необходимую для гарантии конечного успеха и, одновременно, предсказуемую завершаемость проекта, требуемую фиксированными временными рамками, указанными в требованиях к процессу.

Витки выполняются не последовательно, а частично накладываются друг на друга.

Замечание по конструированию

> Это обеспечивает сокращение общих сроков выполнения проекта (при некотором увеличении трудозатрат, что допускается требованиями к процессу).

Конкретный способ наложения витков в модели называется «спиральной структурой». Допускаются различные спиральные структуры, которые могут отличаться количеством витков, наложением фаз в витках и сдвигом витков по времени.

3.4.3.1. Типы проектов

Конкретный вид спиральной структуры допускает различные вариации в зависимости от типа проекта. Рассматриваются два типа проектов, которые условно называются "легкий" и "тяжелый", а также два подтипа, которые называются, соответственно, "сверх легкий" и "сверх тяжелый".

Замечание по конструированию

> Используя объектно-ориентированную терминологию, можно сказать, что предлагаемая модель процесса в целом описывает *класс* процессов разработки. Описание этого класса на мета уровне представляется затруднительным для формулирования и понимания. Поэтому описание устроено следующим образом. Приводятся примеры объектов – экземпляров этого класса (конкретные схемы проектов, которые называются типами и подтипами проектов). Описание каждого типа и подтипа проекта дано в графической форме и сопровождается неформальными текстовыми пояснениями, что именно является типизируемым.

Легкий проект характеризуется тем, что технический риск допущения грубых ошибок на фазах анализа и проектирования невелик. Несоответствие первой версии требованиям и ожиданиям пользователей может быть порождено такими причинами: неполнота или неточность функциональных спецификаций, недовольство пользователей дизайном интерфейса, наличие мелких ошибок, пропущенных на фазе стабилизации. В легком проекте повторное выполнение фаз носит консервативный характер, то есть заключается в исправлениях и доделках.

Сверх легкий проект характеризуется тем, что технический риск допущения грубых ошибок на фазах анализа и проектирования отсутствует, т.е. в начале второго этапа работы над проектом точно известно, что и как нужно сделать. Поэтому схема сверх легкого проекта имеет один неполный виток. Сверх легкий проект является частным предельным случаем легкого проекта.

Тяжелый проект характеризуется тем, что технический риск работы по неверным функциональным спецификациям велик. Несоответствие первой версии ожиданиям пользователя в тяжелом проекте может быть порождено такими причинами: невозможность удовлетворить количественным ограничениями на выбранной платформе или инструментальном средстве, несоответствие специфицированной функциональности реальным целям бизнеса, отторжение реальными пользователями выбранной парадигмы интерфейса и другие ошибки проектирования, которые не могут быть исправлены консервативными улучшениями и требуют повторного проектирования. Поэтому в тяжелом проекте фазы проекта расположены таким образом, чтобы на фазах второго витка можно было использовать результаты фаз первого витка.

Сверх тяжелый проект характеризуется тем, что технический риск работы по неверным функциональным спецификациям близок к единице или не может быть оценен, т.е. в начале работы над проектом не известно, что получится в конце. В схеме сверх тяжелого проекта планируется неудача, проявляющаяся в том, что с первой попытки, возможно, не

удастся достичь вехи "код готов". Таким образом, схема сверх тяжелого проекта предусматривает два с половиной витка.

Типизация проекта, т.е. отнесение проекта к определенному типу и подтипу, требует выбора исходя из нескольких независимых критериев (многокритериальная задача). В предлагаемой модели процесса рассматриваются следующие три критерия:

- критерий новизны;
- критерий сложности;
- критерий времени.

Критерий новизны подразумевает проверку следующих условий:

- инструментальное программное обеспечение является новым (первый раз используемым) для всех разработчиков;
- предметная область проекта является новой (не имеет прямых аналогов среди выполненных проектов);
- тип и/или масштаб приложения являются новыми (не имеют прямых аналогов среди выполненных проектов).

Если не выполнено ни одно из этих условий, то проект относится к сверхлегким, если выполнено только одно из условий, то проект легкий, если выполнены любые два, то тяжелый, а если выполнены все три условия, то проект следует классифицировать как сверх тяжелый.

Критерий сложности основан на подсчете количества принципиально различных специальных (инструментальных) средств, которые предполагается использовать в проекте. Что именно следует считать специальным средством, зависит от специфики проекта. Например, подготовку ТЗ в Word не следует считать случаем использования специального средства, а написание сотни хранимых процедур для SQL сервера – следует.

Если проект требует не более одного специального средства, то это сверх легкий проект, если 2–3, то легкий, если в проект вовлечено больше трех различных средств, то это тяжелый или сверх тяжелый проект.

Критерий времени использует календарные сроки проекта для типизации проекта.

Замечание по конструированию

> Критерий времени применим, только если сроки проекта определяются извне. Обычно сроки проекта определяются по типу проекта, а не наоборот.

Сверх легкий проект имеет продолжительность до двух месяцев, легкий проект – до 6 месяцев, тяжелый проект – до 9 месяцев и сверх тяжелый проект продолжается год или более.

Замечание по конструированию

> Формальные методы оценки технического риска работы по неверным функциональным спецификациям предложить затруднительно. Поэтому типизация проекта по многим критериям является экспертной оценкой (критерии могут противоречить друг другу) и ее правильность во многом зависит от опыта и технического чутья эксперта, которым в большинстве случаев является руководитель проекта.

3.4.3.2. Модель процесса сверх легкого проекта

Модель процесса сверх легкого проекта имеет вырожденную спиральную структуру, состоящую из одного неполного витка, как показано на рис. 13.

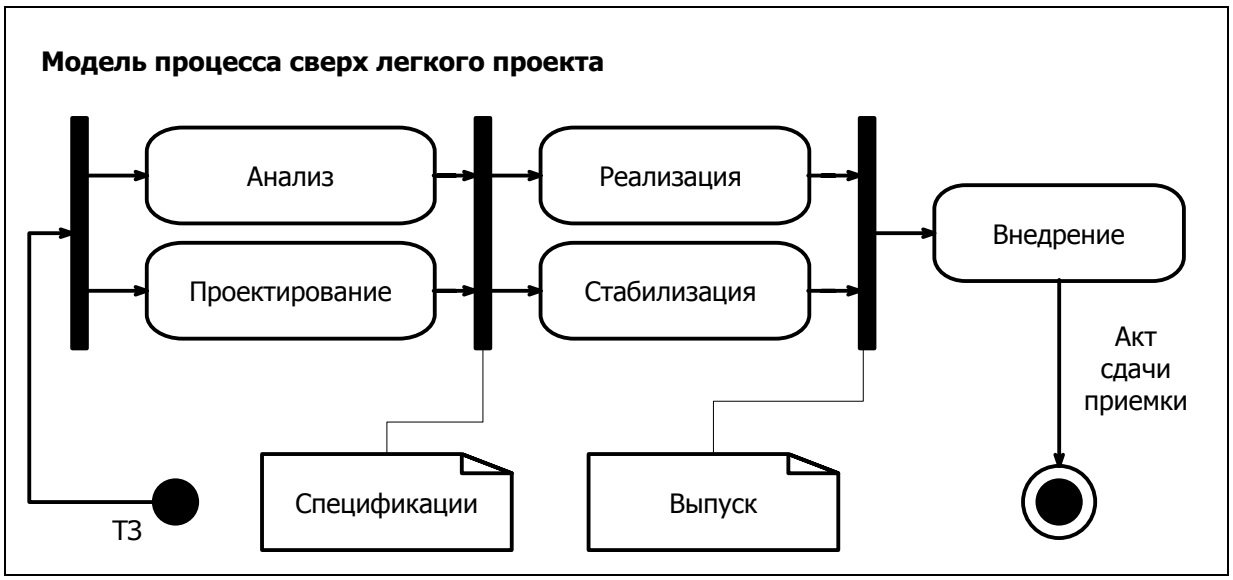


Рис. 13. Модель процесса сверх легкого проекта

В модели процесса сверх легкого проекта предполагается, что фазы анализа и проектирования, а также реализации и стабилизации выполняются параллельно. Это обусловлено тем, что

- ТЗ совпадает с Концепцией и фаза анализа не имеет главной вехи (что не отменяет необходимости самого анализа задачи),
- используется ровно одно инструментальное средство и автономная отладка эквивалентна комплексной отладке.

Замечание по конструированию

> Для целей сравнения продолжительности фаз далее используется некоторая единица измерения времени, которая условно названа "месяц" (практика показывает, что единичная фаза действительно выполняется около месяца, чем и обусловлен выбор названия условной единицы).

Для сверх легкого проекта фазы анализа / проектирования и внедрения требуют примерно вполовину меньше ресурсов, чем для проектов других типов, поэтому общая продолжительность сверх легкого проекта составляет 0,5+1+0,5=2 месяца.

3.4.3.3. Модель процесса легкого проекта

В спиральной структуре легкого проекта второй виток максимально наложен на первый (рис. 14), поскольку на фазе реализации первого витка не ожидается возникновения информации, которая может повлиять на фазу анализа второго витка. Таким образом, для легкого проекта расчет общей продолжительности проводится по формуле 6+1. В типичном случае, считая продолжительность одной фазы равной месяцу, заказчик получает результат через 6 месяцев.

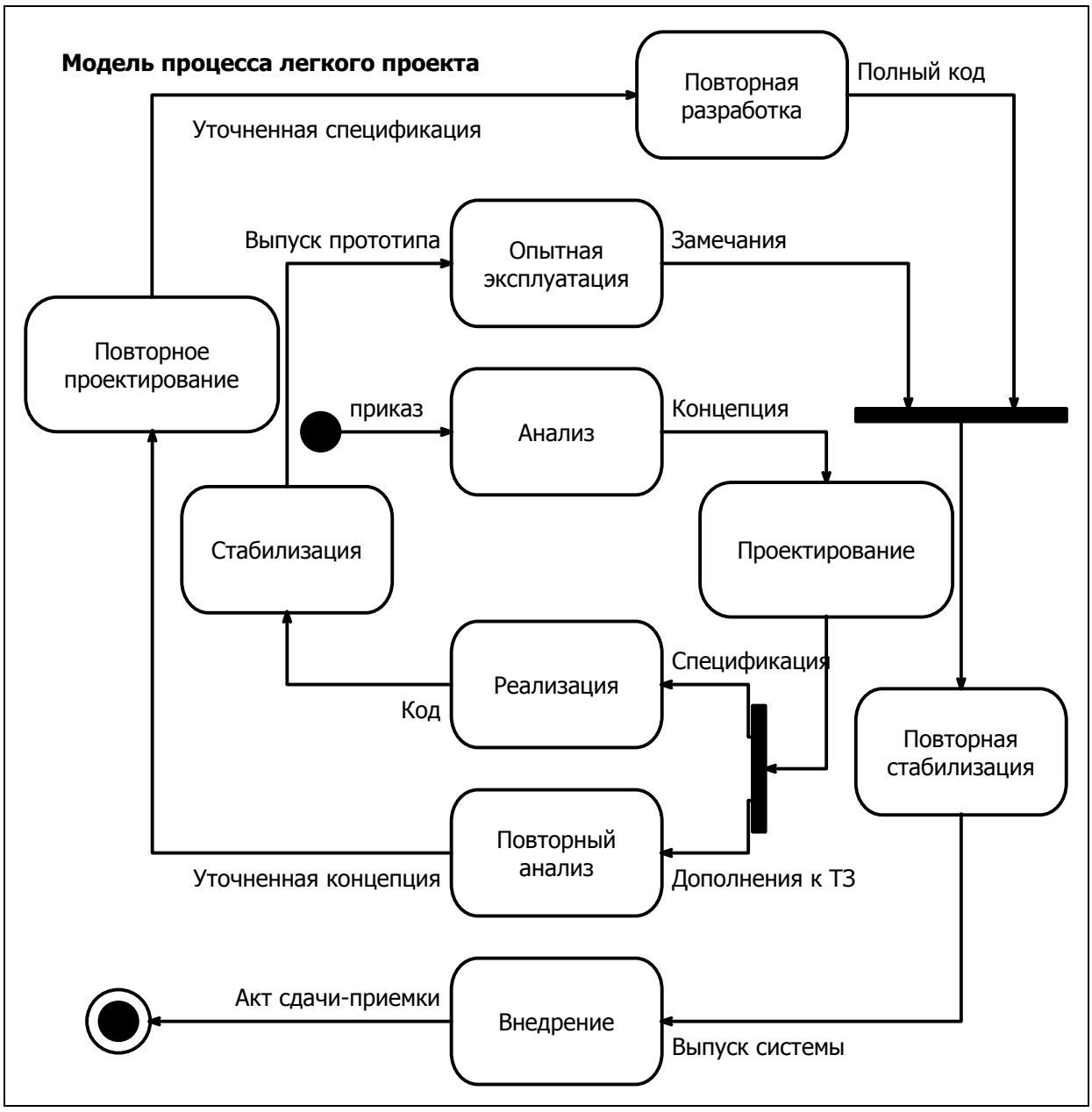


Рис. 14. Модель процесса легкого проекта

Замечание по конструированию

В легком проекте заказчик получает результат через 6 месяцев (а не 7) в худшем случае, так как на фазе внедрения приложение (решение) не подвергается радикальным переделкам, а только косметическим улучшениям.

Схематический план-график модели процесса легкого проекта:

П.	Фаза	Веха	Роль	Примечания
1	Анализ	Концепция	Аналитик	
2	Планирование	Спецификации	Аналитик	
3	Реализация	Код готов	Программист	
	Анализ	Концепция	Аналитик	
4	Стабилизация	Выпуск	Программист / Тестировщик	
	Планирование	Спецификация	Аналитик	Исправление предыдущей спецификации

П.	Фаза	Веха	Роль	Примечания
5	Опытная эксплуатация		Тестировщик / Эксплуатационник	
	Реализация	Код готов	Программист	
6	Стабилизация	Выпуск	Программист / Тестировщик	Учет ошибок, найденных при опытной эксплуатации
7	Внедрение	Проект закончен	Эксплуатационник	

В легких проектах могут совмещаться фазы повторного анализа и повторного проектирования, или же повторной реализации и повторной стабилизации, как что фаза опытной эксплуатации плавно переходит в фазу внедрения и проект заканчивается через 6 месяцев, например, как показано на рис. 15.

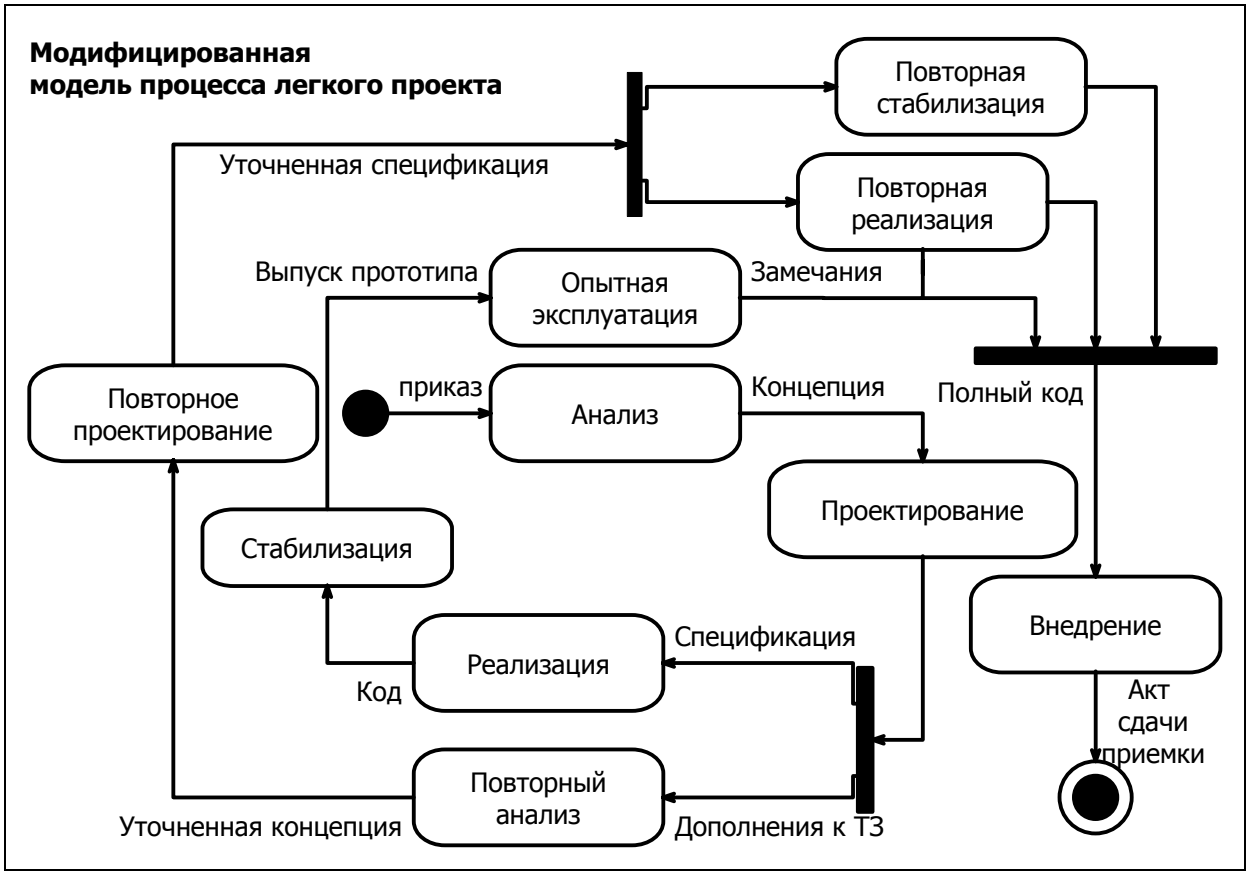


Рис. 15. Модифицированная модель процесса легкого проекта

3.4.3.4. Модель процесса тяжелого проекта

В спиральной структуре тяжелого проекта (рис. 16) второй виток начинается в тот момент, когда достигнута главная веха фазы реализации первого витка и, тем самым, доказана принципиальная возможность реализации при выбранной концепции. Для тяжелого проекта расчет общей продолжительности проводится по формуле 7,5+1,5. В типичном случае, считая продолжительность фаз проектирования и внедрения равной полутора месяцам, а продолжительность остальных фаз равной месяцу, заказчик получает результат через 9 месяцев.

Замечание по конструированию

> В тяжелом проекте фазы проектирования (и повторного анализа), а также внедрения следует немного удлинить.

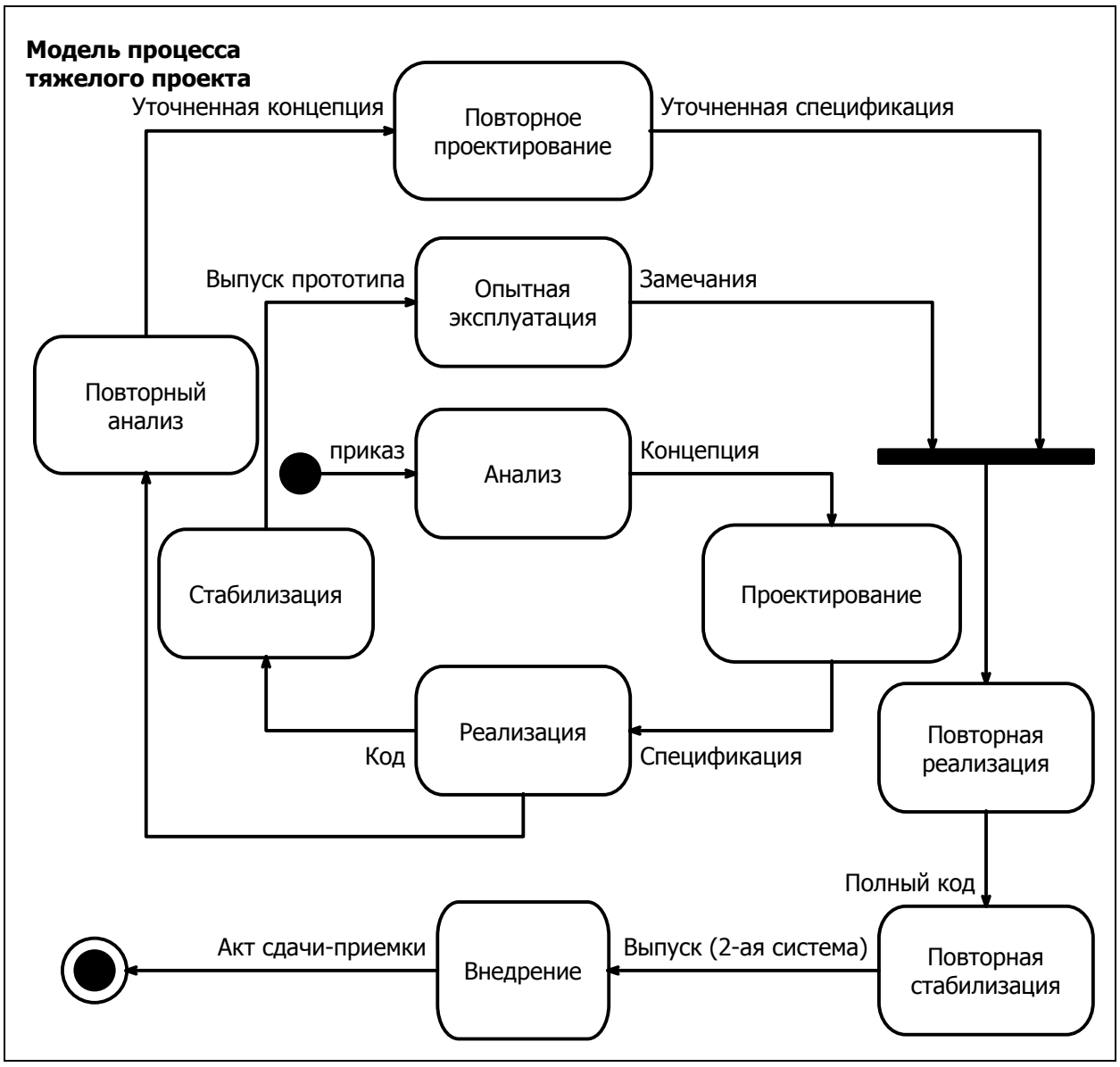


Рис. 16. Модель процесса тяжелого проекта

Замечание по конструированию

На схемах спиральной структуры первый выпуск заканчивается выпуском прототипа[19], который является законченным приложением (решением), но не является тем экземпляром приложения, который будет реально эксплуатироваться заказчиком после окончания проекта. Это не означает, что результаты пилотного проекта полностью выбрасываются. Напротив, они в максимальной степени используются при построении второго "боевого" варианта приложения (решения).

Схематический план-график модели процесса тяжелого проекта:

П.	Фаза	Веха	Роль	Примечания
1	Анализ	Концепция	Аналитик	
2	Планирование	Спецификации	Аналитик	
3	Реализация	Код готов	Программист	
4	Анализ	Концепция	Аналитик	
	Стабилизация	Выпуск	Программист / Тестировщик	

[19] Иногда употребляется термин "пилотный проект". Не следует путать "прототип" на этих схемах с "прототипом", о котором речь шла при перечислении вех фазы планирования.

5	Планирование	Спецификация	Аналитик	Переработка предыдущей спецификации
	Опытная эксплуатация		Тестировщик / Эксплуатационник	
6	Реализация	Код готов	Программист	
7	Стабилизация	Выпуск	Программист / Тестировщик	
8	Внедрение	Проект закончен	Эксплуатационник	

3.4.3.5. Модель процесса сверх тяжелого проекта

Если фаза реализации тяжелого проекта заканчивается неудачей, т.е. вехи "код готов" на первом витке не удается достичь, то проект переходит из категории тяжелых в категорию сверх тяжелых. Такая ситуация означает необходимость увеличить срок выполнения проекта на время, затраченное до момента получения видимой неудачи первоначальной концепции. Если увеличить сроки выполнения проекта невозможно, то проект следует прекратить, чтобы уменьшить неизбежные убытки. В противном случае, т.е. при попытке выполнить проект в намеченные сроки за счет вложения дополнительных ресурсов, возникает очень большой финансовый риск. Если же проект можно удлинить, то он возвращается в категорию тяжелых и проходит согласно модели процесса тяжелого проекта. Таким образом, в типичном случае, считая продолжительность одной фазы равной месяцу, заказчик получает результат сверх тяжелого проекта через 12 месяцев (рис. 17).

Замечание по конструированию

> Согласно авторитетным зарубежным источникам, сокращение сроков требует не менее чем квадратичного увеличения ресурсов. Например, чтобы сократить сроки вдвое, потребуется вложить вчетверо больше ресурсов.

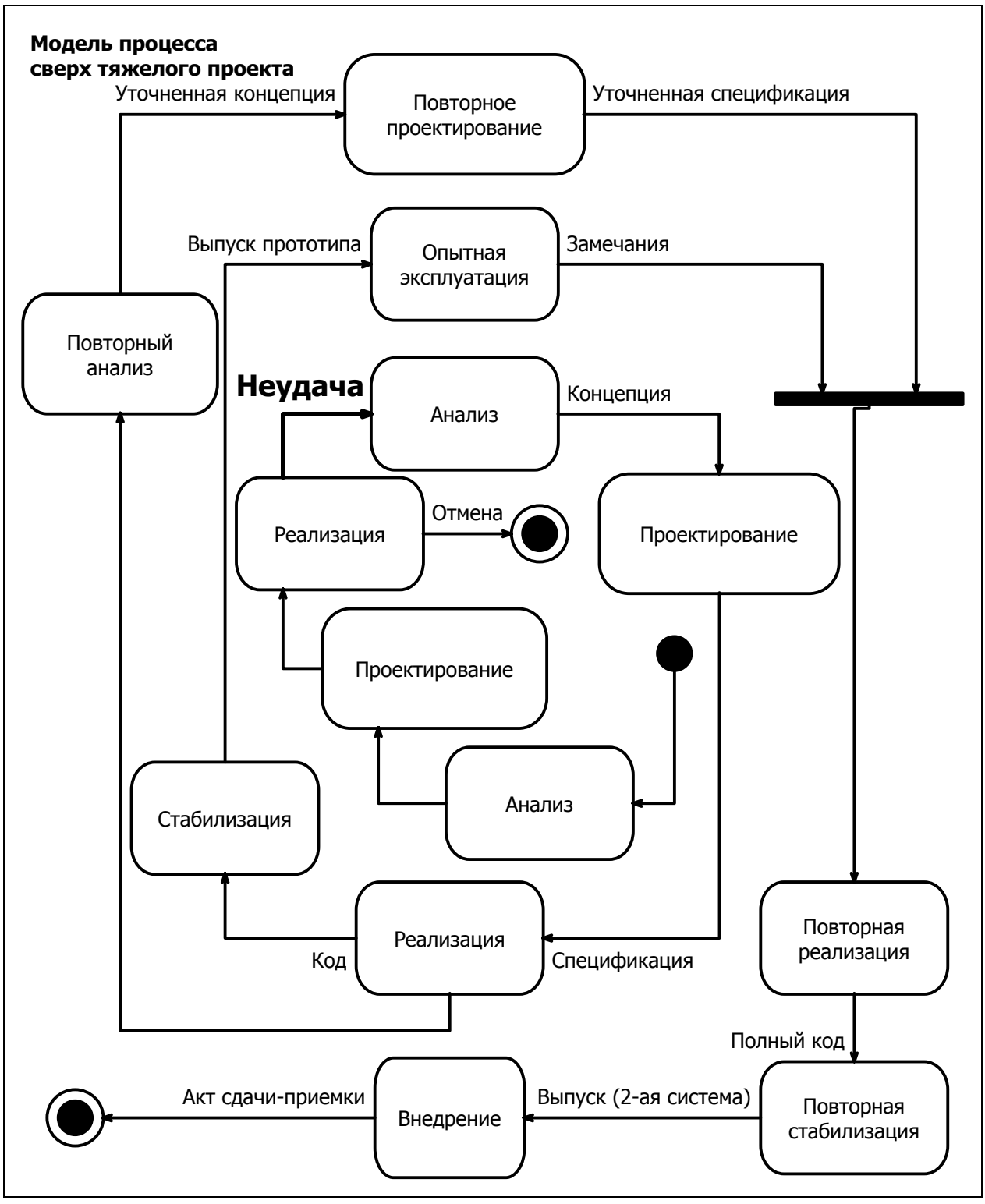


Рис. 17. Модель процесса сверх тяжелого проекта

Замечание по конструированию

Формально в процессе сверх тяжелого проекта имеется 11 периодов, но продолжительность фаз анализа, повторного анализа и окончательного внедрения следует положить равной 1,5 месяца.

3.4.3.6. Занятость исполнителей

Модель процесса предполагает параллельное выполнение нескольких проектов. Для динамического формирования команд исполнителей и составления индивидуальных планов-

графиков необходимо учитывать схему занятости исполнителей, которая определяется моделью процесса.

В случае выполнения сверх легких проектов, в которых на каждый проект задействован один исполнитель, играющий все роли и являющийся руководителем проекта, особых проблем с занятостью исполнителей нет. После окончания одного проекта исполнитель начинает следующий, чем обеспечивается 100% занятости при равномерной нагрузке.

Замечание

> Индивидуальный характер работы над сверх легкими проектами позволяет просто планировать отвлечение исполнителей (отпуск, обучение) на периоды между проектами.

В случае выполнения только легких проектов может быть обеспечена равномерная полная занятость без смены ролевых функций при следующем соотношении численности групп:
Аналитики : Программисты : Эксплуатационники = 2 : 2 : 1

На рис. 18 показана плотная укладка четырех проектов. Фазы первого проекта обведены сплошной линией, фазы второго проекта заштрихованы горизонтально, фазы третьего проекта заштрихованы вертикально, фазы четвертого проекта обведены волнистой линией. Буква А обозначает фазу анализа, П – планирование, Р – реализация, С – стабилизация, О – опытная эксплуатация, В – внедрение. Цифры обозначают номер витка.

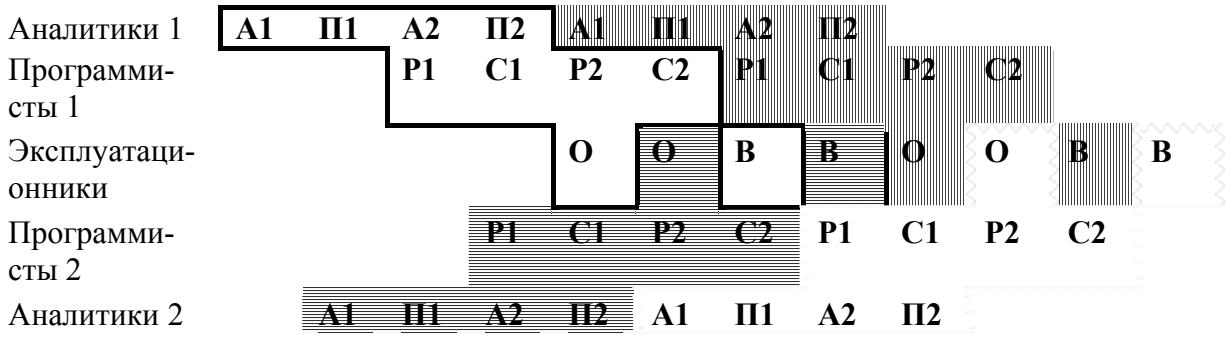


Рис. 18 Схема укладки фаз легких проектов на временной шкале

Замечание по конструированию

> Схему укладки реализовать значительно проще, если исполнители владеют смежными специализациями и могут играть различные роли.

В случае наличия тяжелых и сверх тяжелых проектов плотной укладки без смены ролевых функций не существует. Для проектов этих типов неизбежно динамическое переназначение ролей, и даже в этом случае не всегда возможно распределить роли так, чтобы обеспечить полную занятость и равномерную нагрузку.

Замечание по конструированию

> В предлагаемой модели процесса нет абсолютной жесткой схемы назначения ответственности за достижение вехи на определенную роль. Конкретное назначение ответственности за веху определяется планом фазы, и может быть различным для разных типов проектов.

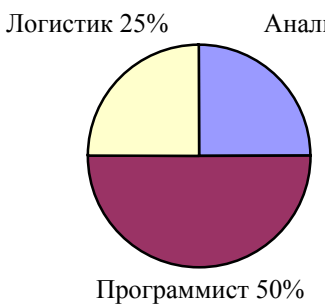
Абсолютные величины трудозатрат по ролям для проектов разных типов (в условных человеко-месяцах):

	Сверх легкий	Легкий	Тяжелый	Сверх тяжелый
Аналитик	0,5	4	4	6
Программист	1	4	3	4
Эксплуатационник	0,5	2	3	3
Всего	2	10	10	13

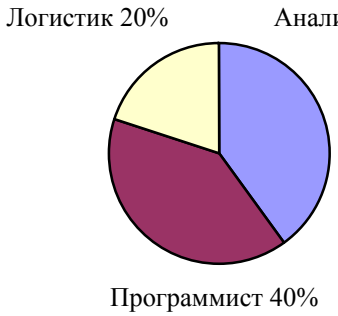
На рис. 20 показано относительное соотношение между трудозатратами для проектов разных типов.

Замечание по конструированию

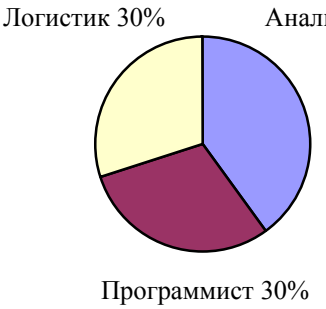
Следует обратить внимание на то, что чем тяжелее проект, тем большую долю составляет работа аналитиков и тем меньшую долю составляет работа программистов.



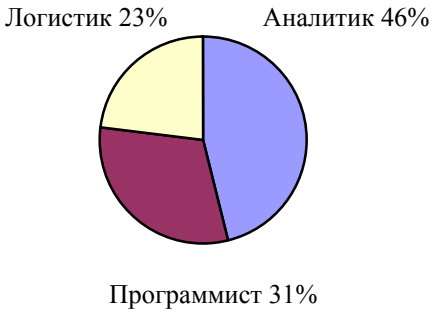
Сверх легкий проект



Легкий проект



Тяжелый проект



Сверх тяжелый проект

Рис. 19. Относительное соотношение между трудозатратами для проектов разных типов

3.4.4. Порядок проведения типового проекта

Замечание по конструированию

Сконструированная в соответствии с потребностями программирующей организации абстрактная модель процесса должна быть переведена в конкретную форму, допускающую ее практическое применение. Такой формой в настоящее время чаще всего является документированная процедура. Документированные процедуры описывают различными средствами, в соответствии с корпоративной культурой и иными особенностями организации. В качестве примера приведем описание документированной процедуры «порядок проведения типового проекта» с использованием унифицированного языка моделирования UML и естественного языка.

В качестве типового проекта, на примере которого описывается порядок проведения, выбран легкий проект, содержащий разработку вертикального приложения по внешнему заказу.

Вопросы *появления* проектов (маркетинг, формирование портфеля заказов, работа с потенциальными заказчиками) находятся вне рамок основной процедуры. Далее считается, что рассматриваемый проект имеет место, т.е. известен потенциальный заказчик, известна предметная область и тип потенциального проекта.

Выполнение проекта другого типа может отличаться в деталях выполняемых действий и в составе документов проекта, но основная процедура является неизменной и состоит в последовательном выполнении трех этапов:

1. подготовка к проекту;
2. работа над проектом;

3. завершение проекта.

Далее рассматриваются основные действия и документы, специфические для каждого из этапов. Выполнение каждого этапа подразумевает выполнение соответствующей процедуры этапа. Общая процедура прохождения проекта представлена на рис. 21.

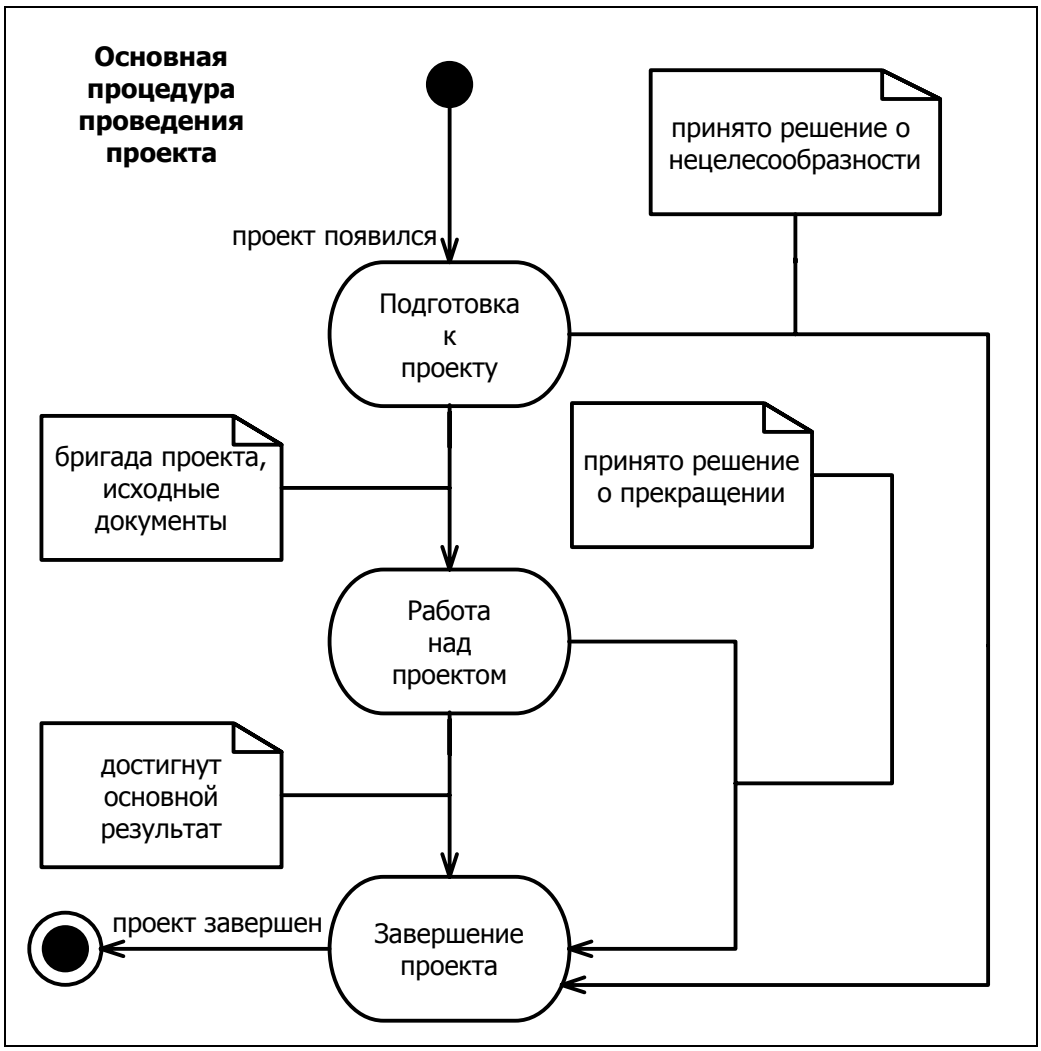


Рис. 20. Основная процедура проведения проекта

3.4.4.1. Этап 1. Подготовка к проекту

На первом этапе подготовки к проекту решаются три основные задачи:

- сбор и анализ предварительной информации;
- формирование бригады проекта;
- подготовка исходных документов.

Сбор информации, формирование бригады и подготовка исходных документов выполняются параллельно, как показано на рис. 22.

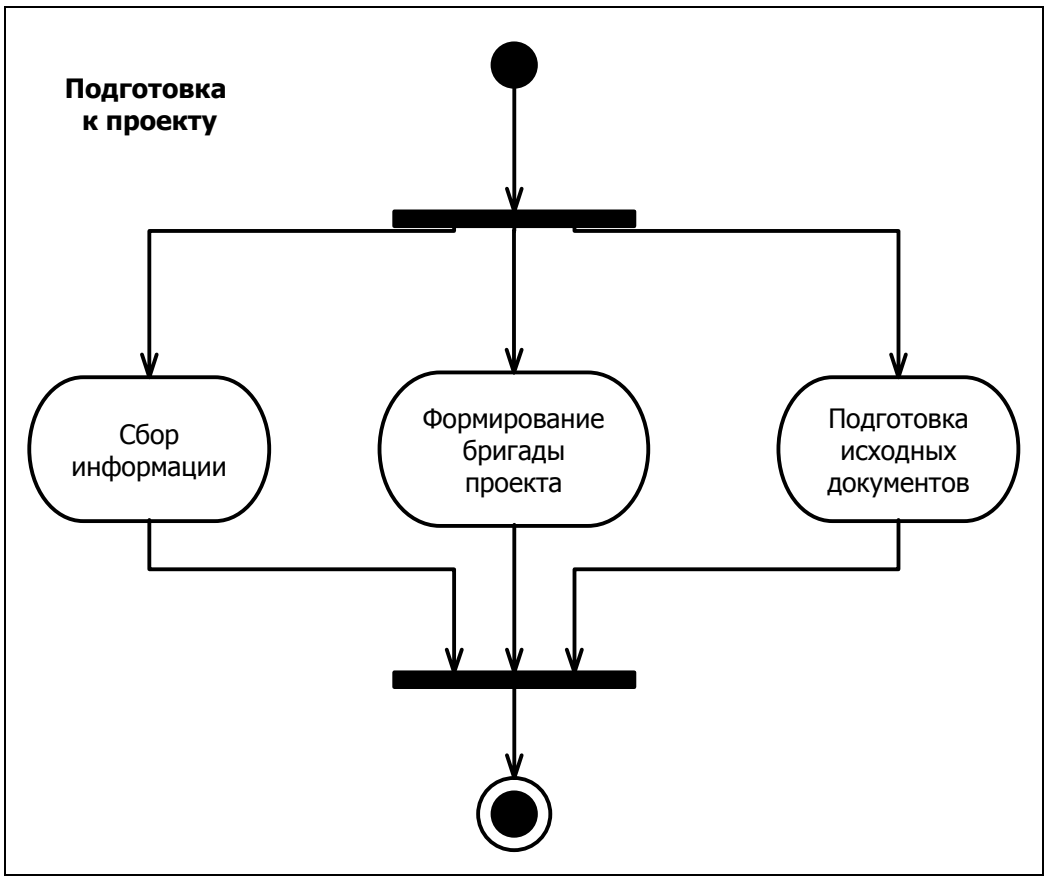


Рис. 21. Процедура «Подготовка к проекту»

3.4.4.2. Сбор и анализ предварительной информации

На первом этапе собирается (актуализируется) различная информация, имеющая отношение к проекту: архивные материалы по законченным аналогичным проектам; научно-технические статьи, мнения экспертов и пр. Для легких и сверх легких проектов собранная информация оформляется в виде аннотированного списка найденных источников информации. Для крупных проектов первый этап называется «Обследованием» и, как правило, оформляется в виде самостоятельного проекта или отдельного этапа проекта в соответствии с общей процедурой. При проведении обследования применяются различные специальные приемы, такие как анкетирование, протоколирование действий пользователей, технические совещания с представителями заказчика.

Замечание по конструированию

> Сбор информации на этапе подготовки к проекту носит *предварительный* характер. В зависимости от типа проекта на фазе анализа второго этапа работы над проектом может быть предусмотрен сбор и анализ более детальной информации о заказчике и предметной области, необходимой для составления концепции («одностраничного» описания) проекта.

Главной задачей сбора информации на этапе подготовки к проекту является получение оценки экономической целесообразности проекта.

3.4.4.3. Формирование бригады проекта

Формирование бригады проекта осуществляется по следующему алгоритму:

1. предварительный анализ;
2. назначение руководителя проекта;
3. подготовка проекта исходных документов;
4. утверждение бригады и запуск проекта.

На рис. 23 приведена процедура формирования бригады проекта и подготовки исходных документов.

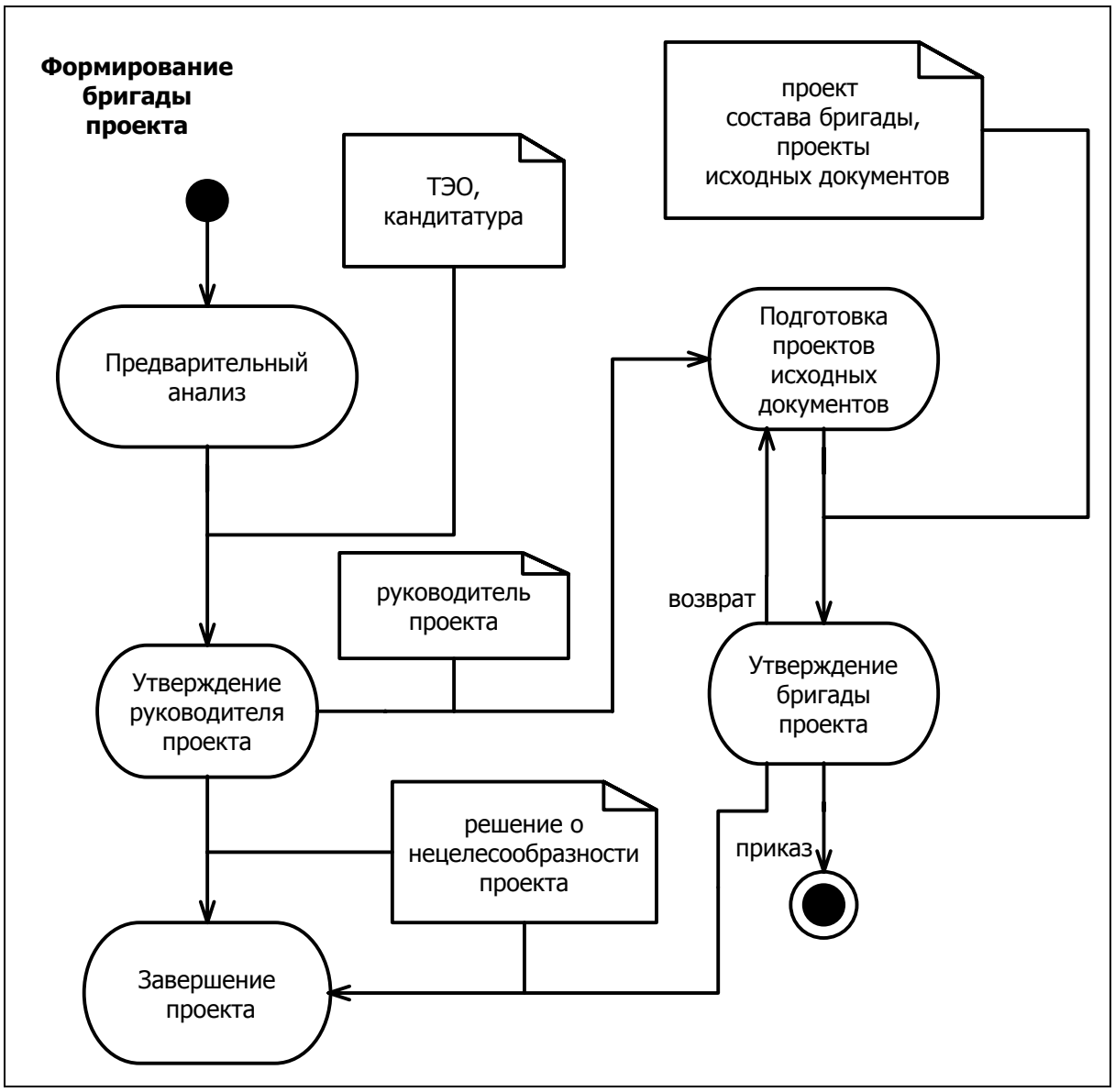


Рис. 22. Процедура «Формирование бригады проекта и подготовка проектов исходных документов»

Замечание по конструированию

> На рис. 23 описана *логическая* взаимосвязь различных действий по формированию бригады проекта и подготовке проектов исходных документов. Эти действия не обязательно выполняются последовательно во времени; как правило, эти действия выполняются параллельно (одновременно). Процедуру на рис. 23 можно было бы определить, как показано на рис. 22.

Шаг 1. Предварительный анализ

Руководитель подразделения разработки проектов самостоятельно или посредством консультаций производит первичный анализ задачи:

- на соответствие ресурсам;
- на квалификацию исполнителей;
- на экономическую целесообразность;
- на предмет развития структуры подразделения и предприятия в целом.

Замечание по конструированию

> В описании процедур используются названия подразделений и должностей. Таким образом, считается, что в организации определены организационная структура и штатное расписание.

Итогом первого шага является оценка экономической целесообразности проекта или технико-экономическое обоснование (ТЭО) проекта и кандидатура руководителя проекта.

Замечание по конструированию

Технико-экономическое обоснование является более широким документом, чем оценка экономической целесообразности. Как правило, ТЭО предусматривает рассмотрение нескольких вариантов решения и обоснование выбора варианта с точки зрения заказчика. Оценка же экономической целесообразности содержит просто предварительный расчет рентабельности проекта с точки зрения организации-разработчика.

Шаг 2. Назначение руководителя проекта

Руководитель подразделения разработки проектов на производственном совещании выносит на обсуждение ТЭО проекта (или оценку экономической целесообразности) и кандидатуру руководителя проекта.

Состав производственного совещания: руководитель предприятия, руководитель подразделения разработки проектов и другие специалисты по усмотрению руководителя предприятия.

Итогом второго шага является решение руководителя предприятия о целесообразности продолжения подготовки к проекту и утверждение кандидатуры руководителя проекта.

Замечание по конструированию

В случае принятия решения о нецелесообразности продолжения подготовки к проекту сразу выполняется последний шаг общей процедуры (архивирование результатов проекта) и на этом выполнение проекта заканчивается.

Замечание по конструированию

Рекомендуется принимать решение о назначение руководителя проекта как можно раньше, чтобы сразу вовлечь его в подготовку исходных документов, в особенности договорных материалов.

Шаг 3. Подготовка проекта исходных документов

Руководитель подразделения разработки проектов знакомит утвержденного руководителя проекта с принятым решением и со всей собранной информацией о проекте. Предлагает детализировать задачу (путем контактов с техническими специалистами заказчика, анализа документов заказчика и т. д.), а также подготовить предложения по составу бригады проекта, распределение ролей в бригаде и оценить необходимые ресурсы на каждом из предполагаемых этапов проекта.

Итогом третьего шага является проект (предварительный вариант для обсуждения) исходных документов, в состав которых могут входить: техническое задание, технические (коммерческие) предложения или «одностраничное» описание проекта; предложения по формированию бригады проекта; предварительная оценка требуемых ресурсов, и, в случае необходимости, предложения по привлечению ресурсов извне.

Замечание по конструированию

Состав и форма предварительных вариантов исходных документов не фиксированы в описании процедуры. Эти документы используются только для предварительного обсуждения на совещании шага 4 и подлежат дальнейшей доработке.

Шаг 4. Утверждение бригады и запуск проекта

Руководитель проекта на производственно-техническом совещании выносит на обсуждение свои предложения.

В состав производственно-технического совещания кроме перечисленных выше руководителя предприятия и руководителя подразделения разработки проектов включаются: руководитель проекта, руководители групп (все или только имеющие административное отношение к составу бригады) и ведущие специалисты по усмотрению руководителя предприятия.

Замечание по конструированию

Сверх легкие проекты может выполнять один человек, который последовательно (или параллельно) играет все роли. Для проектов других типов рекомендуется назначение бригады из нескольких человек, возможно занятых только частично. Процедура динамического назначения команды из состава бригады для выполнения конкретных фаз проекта описана в теме 4 *Модели команды разработчиков*.

Итогом четвертого шага является распоряжение (приказ) руководителя организации о проведении проекта, составе бригады, распределении ролей в бригаде и выделяемым ресурсам. Проект приказа готовит руководитель проекта.

Замечание по конструированию

Появление приказа означает, что этап подготовки к проекту завершен и начался этап работы над проектом. Определяющим является принятие решения о проведении проекта и утверждение состава бригады. Наличие полного комплекта исходных документов не является обязательным для начала работы над проектом. В этом случае доработка исходных документов выполняется на втором этапе. Например, для сверх легких проектов техническое задание и договорные материалы (см. следующий раздел) готовятся и утверждаются до начала второго этапа работы над проектом, а для тяжелых проектов, наоборот, ТЗ утверждается во время работы над проектом на фазе анализа, а Дополнения к ТЗ составляются и утверждаются на фазе повторного анализа. Решение о составе исходных документов принимает руководитель проекта в зависимости от типа проекта.

3.4.4.4. Подготовка исходных документов

К исходным документам относятся:

- Технические (коммерческие) предложения;
- Техническое задание;
- договорные документы;
- другие типы документов, которые создаются или собираются на первом этапе подготовки к проекту, например, письма потенциальных заказчиков, запросы на участие в конкурсе (тендере), внутренние приказы и распоряжения руководства.

На рис. 24 показана общая процедура подготовки исходных материалов. Предполагается, что заказчик участвует в процедуре согласования и не прекращает подготовку к проекту на данном этапе. Детали подготовки конкретных документов описаны в последующих разделах.

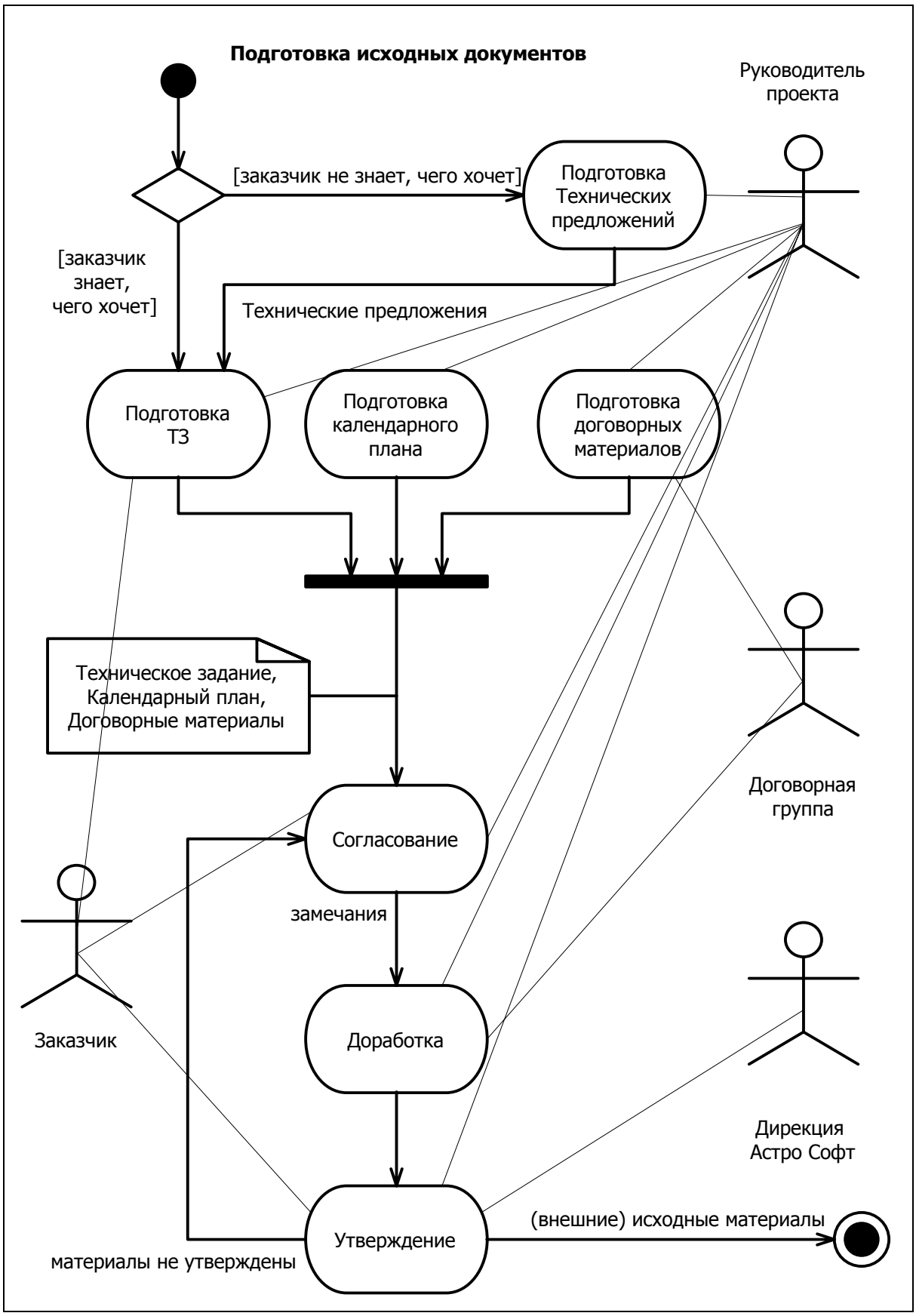


Рис. 23. Процедура "Подготовка исходных документов"

Технические (Коммерческие) предложения

В случае если руководитель проекта более детально представляет предмет разработки, чем представитель заказчика, он инициирует создание Технических (Коммерческих)

предложений, которые служат основой для разработки последующих технических документов, в частности, Технического задания.

Замечание по конструированию

> В MSF роль "представитель заказчика" называется "менеджер продукта".

Технические (Коммерческие) предложения должны содержать по крайней мере следующие основные части: описание целей проекта, укрупненное содержание работы, (в Коммерческих предложениях – экономический эффект от внедрения), краткие сведения о предприятии-исполнителе.

Замечание по конструированию

> Технические (Коммерческие) предложения не являются обязательным документом. Составление этого документа целесообразно в следующих случаях: заказчик – это очень крупная организация; заказчик – совершенно новый для организации разработчика; есть информация о том, что заказчик рассматривает предложения конкурентов; заказчик технически не готов к составлению ТЗ. Очень полезно иметь заранее заготовленные образцы Технических предложений для типовых проектов.

Техническое задание

Как правило, Техническое задание (ТЗ) создается руководителем проекта совместно с представителем заказчика. В проекте, регулируемом договорными документами, ТЗ является обязательным документом, поскольку является приложением к договору.

ТЗ является исходным техническим документом, определяющим назначение, конкретные цели и задачи разработки, технические требования, конечные результаты, и сроки выполнения работ на всех этапах проекта.

ТЗ является основанием для расчета трудоемкости работ и стоимости проекта.

Коррекция ТЗ производится путем оформления Дополнения, которое после согласования становится неотъемлемой частью ТЗ.

Замечание по конструированию

> Согласованное ТЗ обычно включает в себя перечень этапов и является основой для подготовки договорных материалов.

Договорные материалы

В различных заказывающих предприятиях существуют сложившиеся традиции оформления договорных материалов. При оформлении и определении состава договорных документов рекомендуется придерживаться явно выраженных пожеланий заказчика. Таким образом, состав и оформление договорных материалов могут варьироваться в различных проектах. Стандартный набор договорных материалов содержит следующие документы:

- договор;
- техническое задание;
- календарный план со стоимостью каждого этапа и работы в целом;
- протокол согласования договорной цены;
- структура цены.

Договор, Техническое задание, Календарный план, а также Протокол согласования договорной цены являются обязательными договорными документами, прочие документы составляются при необходимости. Порядок составления Технического задания, как наиболее специфического для проекта договорного документа, описан в предыдущем разделе.

Алгоритм процесса согласования договорных материалов (см. рис. 24):

1. проект договорных материалов (включая ТЗ) в одном экземпляре (или в электронном виде) передается заказчику на предварительное согласование;
2. заказчик возвращает проект с замечаниями технического подразделения, планового и юридического отделов;
3. производится коррекция и оформление договорных материалов;

4. заказчику передаются оригиналы договорных материалов в двух экземплярах в виде твердых копий на окончательное согласование;
5. если согласование не получено, то переход на шаг 3.

3.4.4.5. Этап 2. Работа над проектом

При работе над проектом используется
управляемая вехами,
ориентированная на заказчика,
итеративная,
параллельная,
модель процесса, описанная в разделе 3.4.3.
Для типичного случая работы над легким проектом разработки вертикального приложения **процедура работы над проектом** представлена на рис. 24. На этой диаграмме состояния – это фазы работы над проектом, а переходы – это главные вехи проекта.

Замечание по конструированию

Диаграмма на рис. 5 является *примером* схемы организации процесса работы над проектом, отвечающей принципам организации процесса по модели. Вариации процедуры работы над проектом для случая неполного, сверх легкого, тяжелого и сверх тяжелого проекта описаны в разделе 3.4.3. Другие примеры различных схем организации процесса могут быть приведены в других нормативных документах организации. В зависимости от типа проекта и других конкретных обстоятельств, типовая схема может быть изменена таким образом, чтобы соответствовать нуждам конкретного проекта. При этом схема должна соответствовать основным принципам, изложенным в разделе 3.4.3, а отклонения от типовых схем должны быть документированы и обоснованы.

Замечание по конструированию

Диаграмма на рис. 24 отражает *логическую* взаимосвязь фаз и главных вех. Фактически, фазы даже одного витка могут перекрываться во времени. Например, фаза стабилизации может быть начата до того, как код полностью готов. Это зависит от типа проекта, используемых инструментальных средств и принятой дисциплины программирования.

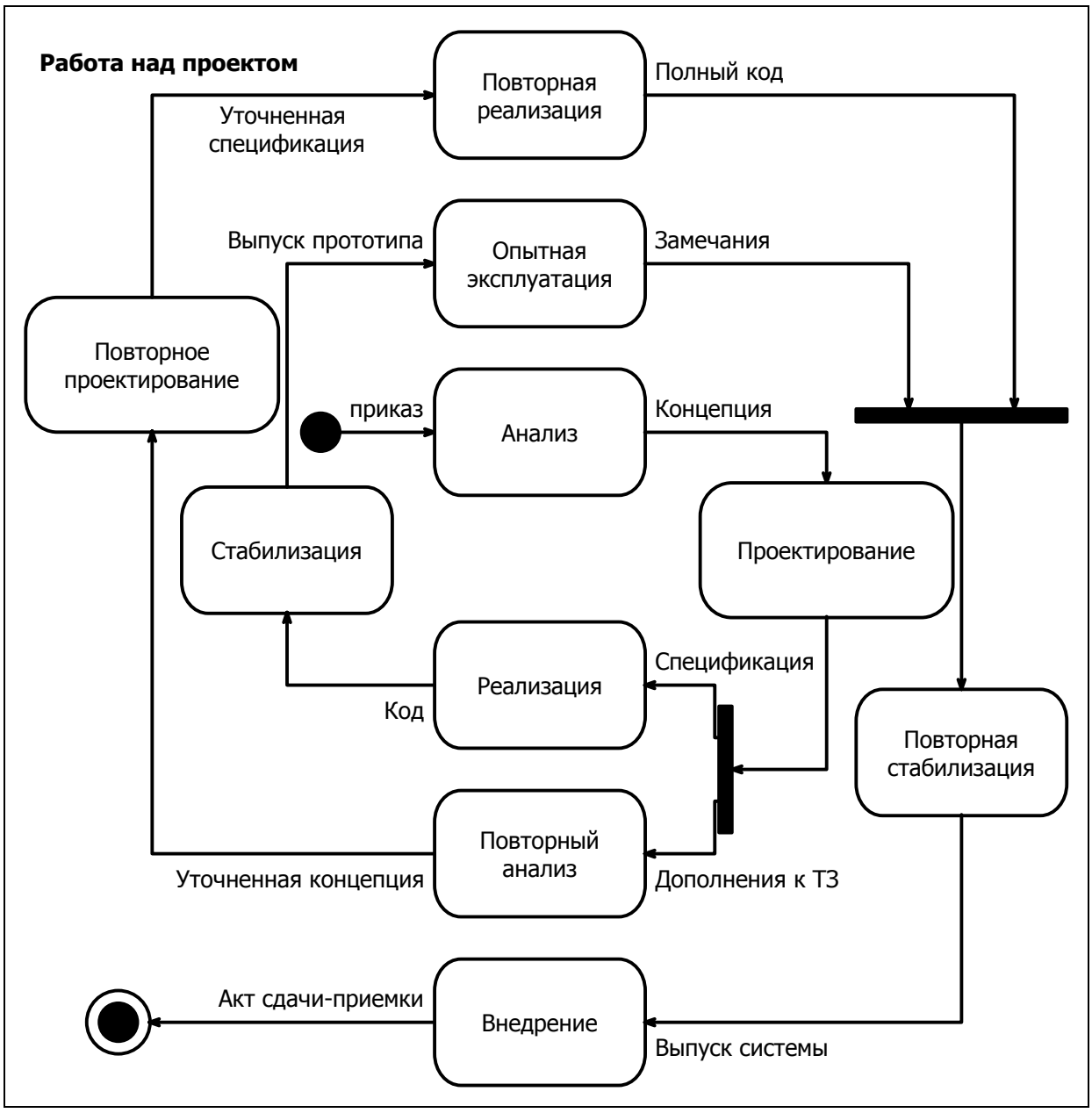


Рис. 24. Процедура "Работа над проектом"

3.4.4.6. Процедура выполнения фазы проекта

Состав промежуточных вех (получаемых результатов) для каждой фазы проекта не является фиксированным и зависит от многих субъективных и объективных факторов:
типа проекта;
индивидуальных пожеланий заказчика;
используемых инструментальных средств;
личных предпочтений руководителя проекта.
Возможные промежуточные вехи для различных фаз проекта указаны в разделе 3.4.3. Абсолютно обязательным для любой фазы является наличие плана в начале и отчуждаемого материала главной вехи в конце.
Назначение ролей для выполнения функций, связанных с достижением промежуточных вех на данной фазе проекта, называется командой фазы проекта. Состав команды также не является фиксированным и зависит от различных факторов:
утвержденного состава бригады проекта;
индивидуальных возможностей и способностей членов бригады;
общего состояния хода работы над проектом;

состояния других проектов, в которых задействованы сотрудники, входящие в бригаду проекта;
внешних обстоятельств (например, возникновения незапланированных работ вне проекта).
Таким образом, процедура выполнения конкретной фазы проекта является предметом динамического (оперативного) планирования, которое выполняется руководителем проекта. Диаграмма процедуры выполнения фазы приведена на рис. 25.

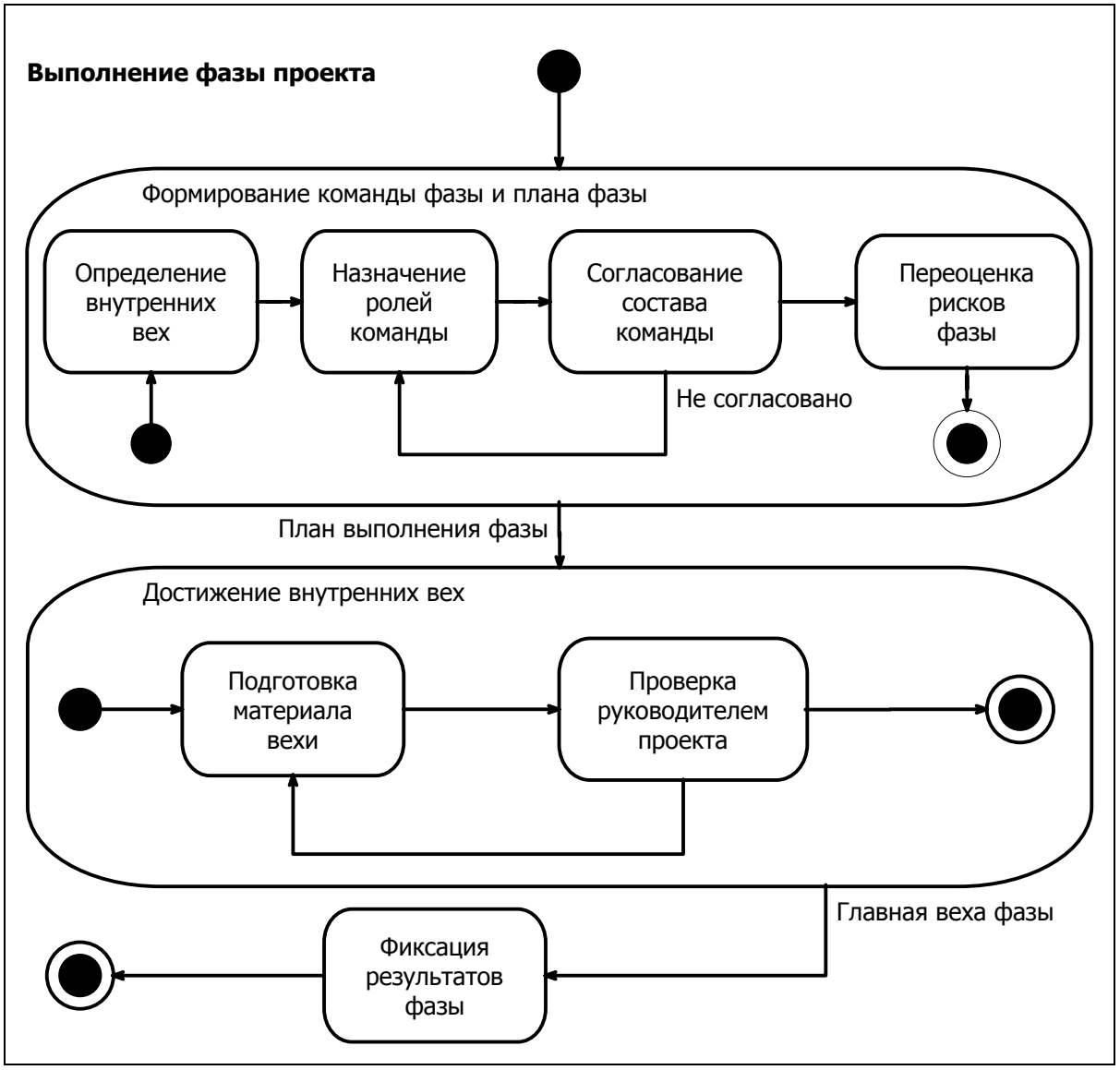


Рис. 25. Процедура "Выполнение фазы проекта"

На этой диаграмме подразумевается, что "Достижение внутренних вех" – это множество однотипных составных активностей, которые могут выполняться параллельно для всех вех фазы (в том числе для главной вехи).
Составление плана выполнения фазы, проверка и фиксация результатов выполнения фазы, а также отчет о выполнении фазы (см. ниже раздел *Периодическая отчетность*) являются прерогативой и *обязанностью* Руководителя проекта (кроме того, Руководитель проекта может играть на данной фазе какую-либо конкретную роль и готовить материалы вех).

Замечание по конструированию

> Руководитель проекта может играть или не играть определенные роли на различных фазах в зависимости от типа и других особенностей проекта. Загрузка всех ролей может быть полной или частичной. Например, для сверх легкого проекта Руководитель проекта играет все роли и загружен на 100%. Для сверх тяжелого проекта с большой бригадой Руководитель

проекта играет, в основном, роль администратора ("освобожденный руководитель") и только эпизодически выполняет иные функции, при этом его загрузка может быть менее 100%.

План выполнения фазы

План выполнения фазы должен содержать следующую информацию:
список вех фазы с указанием для каждой вехи формы отчуждаемых материалов вехи и календарного срока достижения вехи;
список ролей команды фазы с указанием для каждой роли сотрудника из состава бригады проекта, назначенного на роль, и процента использования рабочего времени сотрудника, которое отводится на выполнение функций роли;
оценка (переоценка, если риск был оценен для проекта в целом) рисков фазы;
список связей между вехами фазы и ролями команды с указанием для каждой связи оценки ресурсов, необходимых для выполнения функций, обеспечивающих достижение данной вехи.
План выполнения фазы, как правило, составляется с помощью программы Microsoft Project. Для небольших проектов план фазы может быть частью общего плана проекта, для более крупных рекомендуется составлять планы фаз отдельно, чтобы они были обозримы. На рис. 26 приведен пример: фрагмент плана фазы проектирования для некоторого условного проекта. На рис. 28 тот же пример плана представлен в форме таблицы.

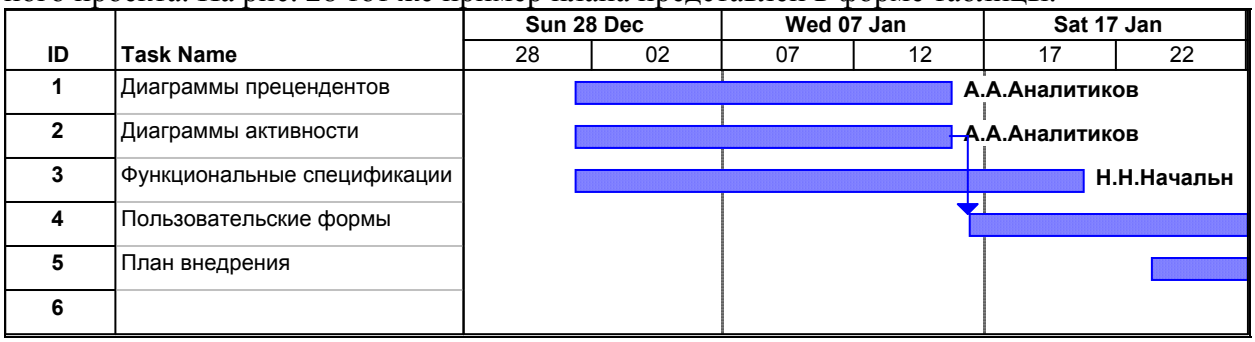


Рис. 26. План фазы проектирования, построенный с помощью Microsoft Project

Веха	Форма	Срок	Ресурсы	Роль	Исполнитель	Загрузка
Функциональные спецификации	Диаграммы прецедентов	01.01-15.01	1 неделя	Аналитик	А.А.Аналитиков	100%
	Диаграммы активности		1 неделя			
	Текст	01.01-20.01	2 недели	Аналитик	Н.Н.Начальников	45%
Прототип	Пользовательские формы	15.01-30.01	2 недели	Программист	П.П.Программистов	100%
План внедрения	Текст	01.01-30.01	1 день	Аналитик	Н.Н.Начальников	5%

Рис. 27. Календарный план фазы проектирования в форме таблицы

Замечание по конструированию

Использование Project позволяет не только представить план в наглядной форме, но и анализировать план, например, определять критические задачи, выявлять перегруженные ресурсы и т.п. Поэтому в проектах, выполняемых по стандарту, рекомендуется использование Project и не рекомендуется использование упрощенных форм планирования типа таблиц.

Замечание по конструированию

Веха может включать несколько результирующих материалов (например, Спецификация может состоять из нескольких документов), на одну роль может быть назначено несколько сотрудников (например, код могут писать несколько Программистов). План выполнения фазы должен быть составлен с такой степенью подробности, чтобы давать однозначный ответ на во-

прос, кто из сотрудников является ответственным за каждый из результирующих материалов фазы.

Замечание по конструированию

Сумма процентов использования рабочего времени сотрудников не должна превышать 100% с учетом возможного участия сотрудника в бригадах нескольких проектов. Ответственным за выполнение этого условия является административный начальник сотрудника.

3.4.4.7. Подготовка результирующих материалов вех

Состав материалов, которые готовятся на втором этапе в ходе работы над проектом и составляют отчуждаемый результат проекта, существенно зависит от типа и объема проекта. Для типичного проекта разработки вертикального приложения на заказ возможный состав материалов кратко описан в разд. 3.4.3. Различаются внешние материалы проекта, которые согласуются с заказчиком и/или входят в состав результатов проекта, предусмотренных Техническим заданием, и внутренние материалы, которые заказчику не предоставляются.

Замечание по конструированию

Из сказанного не следует, что внешние материалы готовятся только для заказчика, а внутренние материалы исчезают при завершении проекта. Напротив, внешние материалы интенсивно используются бригадой проекта, а внутренние материалы архивируются после завершения проекта для повторного использования в последующих проектах. В связи с этим к качеству оформления внешних и внутренних материалов предъявляются одинаковые требования.

Основное различие между внешними и внутренними материалами состоит в том, что внешние материалы планируются на первом этапе подготовки к проекту при составлении Технического задания и Календарного плана, а внутренние материалы планируются на втором этапе работы над проектом при планировании каждой очередной фазы.

Далее рассматриваются наиболее важные и типичные виды материалов проекта. По инициативе заказчика может быть разработан план-проспект (краткое содержание) внешних документов с целью предварительного согласования формы и содержания итоговых документов проекта.

Концепция

Концепция – это документ (или серия документов), который является основным результатом фазы анализа. Основное назначение Концепции состоит в том, чтобы явно сформулировать общее и согласованное понимание проблемы и путей ее решения, которое бы разделялось всеми участниками проекта: заказчиком (включая будущих пользователей), бригадой проекта (включая руководителя проекта) и руководством. Как правило, это внешний документ, во многом опирающийся на Технические предложения и Техническое задание. Отличие Концепции состоит в том, что это менее формальный документ, ориентированный на описание технической сути проблемы, а не на организационные и юридические аспекты, связанные с ее решением.

Замечание по конструированию

В ЕСПД Концепции соответствует Эскизный проект. Если организация склонна к использованию терминологии ЕСПД, то документ "Концепция" можно называть "Эскизный проект", от этого суть и назначение документа не меняются.

Оценка риска

На этапе подготовки к проекту производится начальная оценка риска.

Оценка риска – это список различных факторов и обстоятельств, которые могут отрицательно повлиять на выполнение проекта, а также оценка вероятности возникновения таких обстоятельств вместе с перечнем мероприятий, направленных на уменьшение этой вероятности.

Замечание

Оценка риска является внутренней информацией, которая используется Руководителем проекта и руководством при планирования процесса, выделении и перераспределении ресурсов. Как правило, оценка риска не предоставляется заказчику. Однако, в некоторых специальных случаях особых отношений с заказчиком оценка риска может включаться в ТЗ. Методика оценки риска является отдельной достаточно сложной процедурой, не включенной в данной описание процесса.

Симптомы проявления и нарастания факторов риска должны быть документированы, чтобы их можно было как можно раньше обнаружить и провести соответствующие мероприятия. Различают следующие типы риска:

технический риск, который связан с неосуществимостью выбранного технического подхода или возникновением проблем при реализации запланированного технического решения;

календарный риск, который ставит под угрозу срок выполнения этапа или проекта в целом;

управленческий риск, который связан с выходом за рамки бюджета, негативной реакцией заказчика или плохими контактами с пользователями;

бизнес-риск, как опасность того, что система не будет удовлетворять (частично или полностью) экономическим требованиям и ожиданиям заказчика.

Оценка риска может быть самостоятельным документом, а может входить в качестве раздела в ТЭО.

Замечание по конструированию

Переоценка риска производится на каждой фазе для всех типов проектов, кроме сверх легкого.

Спецификации

Спецификации проекта являются основным результатом фазы анализа. Как правило, спецификации проекта состоят из нескольких материалов различного типа.

Функциональные спецификации. Исчерпывающее и подробное описание функций решения. Для описания функциональных спецификаций могут использоваться разнообразные средства: естественный язык; специальные таблицы; формальные языки; компьютерные средства моделирования и др. Наиболее перспективным является применение средств моделирования, поддерживающих Unified Modeling Language (UML).

Замечание по конструированию

Следует различать внутренние и внешние спецификации. Во внешних спецификациях не должно быть деталей реализации, которые не интересны заказчику. Если они появляются – это путает и пугает заказчика. В то же время только описания сценариев и пользовательских форм не достаточно для программистов, пишущих код. Унифицированный язык моделирования UML является тем средством специфицирования, которое позволяет перейти от внешних спецификаций к внутренним путем последовательного уточнения, и поэтому должно использоваться как основное средство.

Прототип. Компьютерная модель приложения, если проект подразумевает разработку приложения. Прототип может быть полнофункциональным или только прототипом интерфейса, если приложение имеет интерфейс пользователя. Полнофункциональный прототип в некоторых случаях подменяет функциональные спецификации.

Внутренний язык. Описание формального языка (внутреннего языка командного типа) рекомендуется для всех многофункциональных приложений, допускающих варьируемые сценарии использования. Для приложений, имеющих программный интерфейс (API), такое описание крайне желательно.

Замечание по конструированию

Например, для проекта, связанного с разработкой транслятора, описание входного и выходного языков является главной частью спецификации. Практически единственным общепризнан-

> ным средством спецификации является в этом случае описание порождающих грамматик. Эту же технику можно с успехом применять и в случае вертикальных приложений для бизнеса.

Дополнительные требования. Перечень и описание количественных ограничений и/или специальных требований (например, по устойчивости системы защиты информации), которым должно удовлетворять решение. Дополнительные требования часто оформляются в виде Дополнения к Техническому заданию.

План тестирования. План тестирования содержит набор тестов и описания порядка их применения с целью измерения таких важнейших характеристик разрабатываемого в рамках проекта приложения, как надежность, функциональная полнота и применимость.

Замечание по конструированию

> В некоторых случаях План тестирования не включается в Спецификации, а является первой промежуточной вехой фазы стабилизации.

План внедрения. Описание процедур развертывания и демонстрации работоспособности разрабатываемого приложения. Кроме того, план внедрения может включать план обучения пользователей и технических специалистов заказчика и план-проспект пользовательской документации, если это предусматривается общим планом проекта.

Замечание по конструированию

> В некоторых случаях План внедрения не включается в Спецификации, а является первой промежуточной вехой фазы внедрения (опытной эксплуатации).[20]

Как правило, спецификации проекта (полностью или частично) являются внешними материалами и в общем случае контролируются заказчиком.

Модель приложения

Для подготовки различных материалов фаз анализа, планирования и реализации целесообразно использовать средства визуального моделирования, в особенности средства, поддерживающие UML. Средства моделирования UML ориентированы на представление различных аспектов моделируемого приложения с помощью диаграмм различного типа. Именно поэтому диаграммы UML могут служить в качестве результатов таких вех, как Спецификации, Логический проект, Физический проект и даже Код готов (если средства автоматической генерации кода это позволяют).

Замечание по конструированию

> В идеале полная модель (набор диаграмм UML) может служить в качестве комплекта материалов проекта, за исключением договорных и финансовых документов.

База данных тестирования

Наличие и ведение базы данных тестирования является обязательным для любых проектов, связанных с разработкой программного кода.

Пользовательская документация

В большинстве проектов на фазе внедрения составляется документ (или несколько документов) с названием «Руководство пользователя».

3.4.4.8. Этап 3. Завершение проекта

Назначение третьего завершающего этапа состоит в том, чтобы подготовить результаты проекта к повторному использованию. Таким образом, завершающий этап выполняется в интересах программирующей организации, а не заказчика. Признаком перехода на завершающий этап является подписание акта сдачи-приемки работы. На рис. 28 приведена диаграмма, иллюстрирующая порядок завершения проекта.

---

[20] В ЕСПД План внедрения и План тестирования оформляются как один документ с названием Программа и методика испытаний. Если организация склонна к использованию ЕСПД, то "План внедрения" можно называть "Программа и методика испытаний". План тестирования лучше иметь в виде отдельного документа.

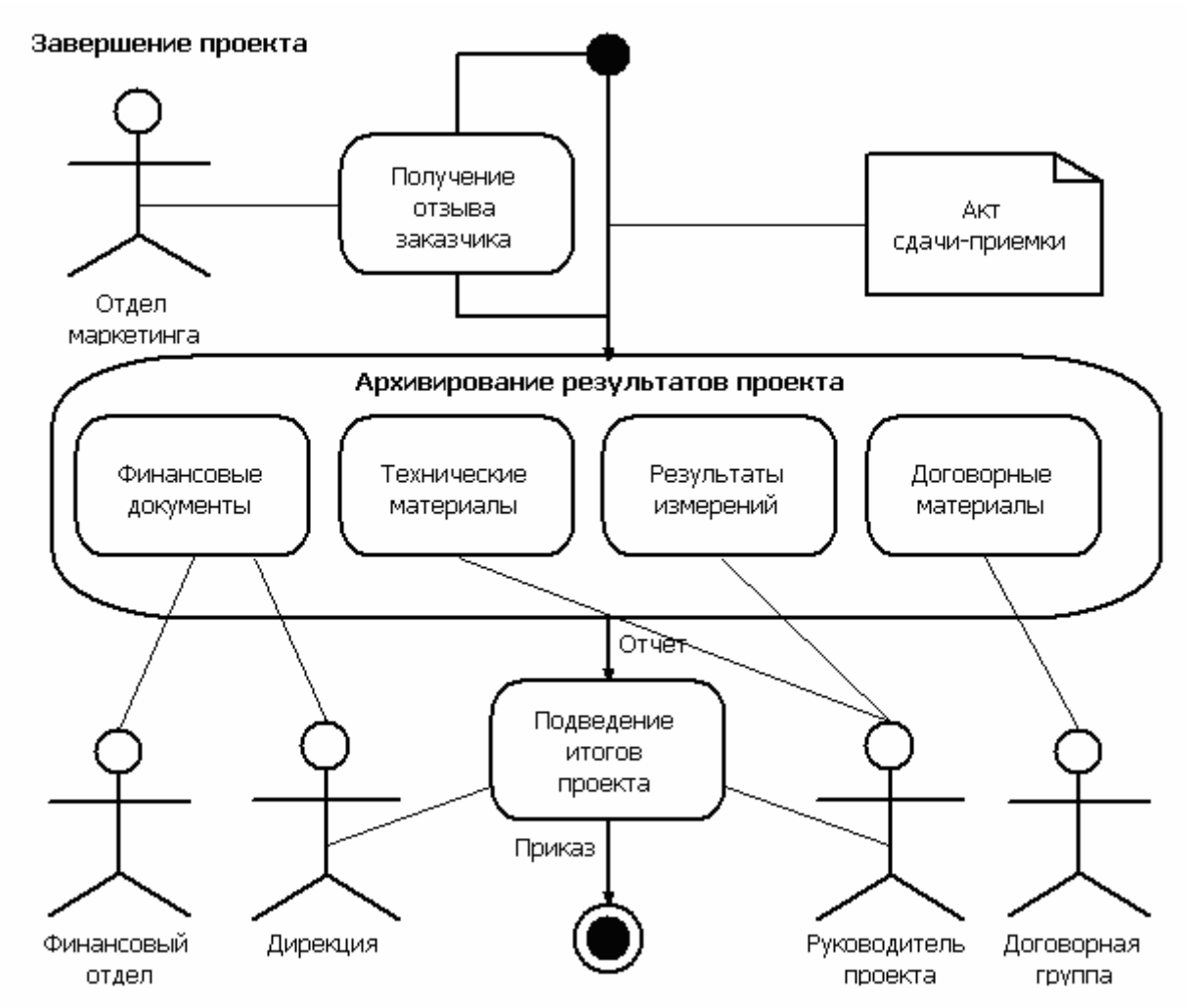


Рис. 28. Процедура "Завершение проекта"

Акт сдачи-приемки работы

Акт сдачи-приемки работы (этапов работы) является одновременно техническим и финансовым актом, содержащим сведения о том, что работа выполнена в соответствии с договором и ТЗ. Акт оформляется (согласуется) в том же порядке, в котором оформляется договор. Типовой бланк акта сдачи-приемки имеется в договорной группе (см. документ *Образцы договорных документов*).

Отзыв заказчика

Отзыв заказчика – это обобщенное название документально зафиксированного факта удовлетворенности заказчика результатами проекта. Отзыв может иметь различные формы, например:
формальный отзыв в виде письма на бланке заказчика;
письменное разрешение заказчика на публикацию результатов проекта в открытой печати;
письменное разрешение заказчика на использования названия фирмы заказчика и других данных в маркетинговых материалах .
Целесообразность получения и предпочтительная форма отзыва заказчика определяются с помощью отдела маркетинга.

3.4.4.9. Архивирование результатов работы

Исходные, промежуточные и отчетные документы проекта существуют в виде файлов, а также в виде твердых копий; копии либо передаются заказчику, либо являются рабочими

документами для организации технических семинаров проектной группы. Для архивирования результатов работы:
Технические материалы проекта помещаются в папку коллективного доступа, которую определяет администратор сети. Доступ к этой папке ограничен (для технических специалистов). По мере накопления материалов проектов, по усмотрению администратора сети, материалы проектов целиком переносятся на компакт-диски для постоянного хранения.
Оригиналы финансовых и договорных материалов в виде твердых копий хранятся в финансовом отделе. Кроме того, договорные материалы архивируются в специальной папке, которую ведет договорная группа. Доступ к этим папкам ограничен (для технических специалистов).
Исходная (кроме договорной), промежуточная и отчетная документация хранятся у руководителя подразделения разработки в виде твердых копий в отдельных папках с названиями (шифрами) проектов.

3.4.4.10. Подведение итогов проекта

После архивации результатов проекта Руководитель проекта составляет отчет о результатах проекта, в котором содержится следующая информация:
общие сведения о выполненном проекте: название, шифр, календарные сроки выполнения, краткие сведения о заказчике, состав бригады проекта, краткие сведения о назначении проекта, тип проекта, перечень использованных технологий и инструментов;
точные ссылки на все архивированные результаты проекта;
выводы Руководителя проекта об успешности или не успешности проекта и другие замечания, которые он посчитает нужным включить в отчет.
Отчет о результатах проекта – это короткий документ (около одной страницы), который хранится в оперативной общедоступной базе данных отчетов о проектах. Посредством этого документа осуществляется доступ к материалам проекта для их повторного использования.
Кроме отчета, Руководитель проекта составляет три приложения к отчету:
Анализ экономической эффективности проекта. Анализ производится путем сравнения плановых показателей экономической эффективности в ТЭО и фактических показателей, измеренных в ходе проекта.
Предложения по премированию бригады проекта. Предложения по премированию по результатам успешного проекта подготавливаются Руководителем проекта в соответствии с нормативными документами организации.
Выводы и предложения по итогам проекта. Руководитель проекта сравнивает фактические средние показатели производительности в проекте с такими же показателями для аналогичных предыдущих проектов с целью выявления тенденции их изменения. По результатам анализа делаются выводы и формулируются предложения, например, о внесении изменений и дополнений в порядок прохождения проекта, об использовании технологий и инструментальных средств, о повышении квалификации персонала и т.д.
На производственном совещании, посвященном подведению итогов проекта,[21] рассматриваются подготовленные Руководителем проекта отчетные документы и принимаются соответствующие административные и организационные решения. Проект завершается изданием приказа (рис. 28).

Замечание

> По результатам законченного проекта целесообразно проводить неформальный информационный семинар, с презентацией Руководителем проекта основных результатов, а также выводов и предложений. Аудиторией семинара может быть группа разработчиков, подразделение разработки или вся организация.

[21] На бюрократическом жаргоне такое совещание называют "разбор полетов".

3.4.5. Документированные процедуры

Выше описаны типовые процедуры, используемые при проведении проекта по модели , которые применяются к самому проекту, т.е. к бригаде проекта, процессу проведения проекта, материалам проекта. Ниже приведены процедуры, которые не имеют отношения к конкретным проектам и описывают порядок применения модели безотносительно к проектам. Основных процедур четыре:

- учет рабочего времени (трудозатрат и производительности);
- ведение регулярной отчетности (в проекте и вне проекта);
- проверка качества материалов (нормоконтроль);
- управление документами.

Замечание по конструированию

> В данном случае используется минимальный набор документированных процедур для удовлетворения требования легкости процесса. Конкретные обстоятельства организации, например, сертификация системы менеджмента качества, может потребовать существенного расширения этого списка.

3.4.5.1. Учет рабочего времени

Учет рабочего времени – это стандартная измерительная процедура, обеспечивающая сбор исходной информации о фактических затратах трудовых ресурсов на проведение проектов.

Назначение этой процедуры состоит в регулярном сборе и документировании данных о фактических затратах рабочего времени сотрудников на выполнение конкретных функций на различных фазах прохождения проектов. Собранные данные используется для ретроспективного статистического анализа с целью выявления и устранения узких мест в технологии организации проведения проектов, определения нормативных показателей производительности для конкретных операций, выработки усредненных нормативов для более точного планирования фаз проектов

Замечание по конструированию

> Следует подчеркнуть, что данные, собираемые в процессе учета рабочего времени, не могут являться основанием для административного воздействия на сотрудников. Основанием для административных воздействий (поощрений и наказаний) являются только результаты прохождения проектов.

3.4.5.2. Периодическая отчетность

Периодическая отчетность является одной из основополагающих процедур, обеспечивающих обратную связь в контуре управления порядком прохождения проектов. В той или иной форме эта процедура присутствует в любой системе управления. В модели процесса предусматривается отчетность нескольких видов:
внутренняя отчетность в рамках проекта;
внешняя отчетность в рамках проекта (перед заказчиком);
внутренняя отчетность в рамках организации.
В рамках работы бригады отчетность обеспечивается основной процедурой проведения проектов, которая управляется вехами. Результирующие материалы вех и есть необходимая и достаточная отчетность в рамках проекта. Внешняя отчетность перед заказчиком также регламентирована вехами (внешними) и предопределяется планом проекта и планами выполнения фаз. Необходимость в других формах отчетности возникает, только если планы не выполняются. В этих случаях используются различные методы сбора отчетной информации и динамического изменения планов: обмен сообщениями электронной почты, тематические доклады исполнителей руководителю проекта, технические совещания, структурированная экспертиза материалов и др.

Внутренняя отчетность в рамках организации производится в соответствии со статической организационной структурой и привязана не к вехам, а к календарю и событиям организации. Внутренняя отчетность организована по уровням – подчиненный отчитывается перед своим непосредственным начальником. Для целей проведения проектов периодическая отчетность должна включать информацию, необходимую для оперативного учета и планирования материально-технических и трудовых ресурсов, требующихся для выполнения проектов, поскольку суммарной занятостью сотрудников, участвующих в нескольких проектах, управляют непосредственные начальники сотрудников, а материально-техническими ресурсами управляет начальник руководителя проекта.

В типичном случае для проведения проектов достаточно еженедельного[22] отчета руководителя проекта,[23] адресованного руководителю подразделения, и включающего:

1. список вех, достигнутых за прошедшую неделю;
2. персональный состав команды фазы с указанием процента занятости за прошедшую неделю;
3. список вех, планируемых на следующую неделю;
4. персональный состав команды фазы с указанием процента занятости на следующую неделю;
5. список особых обстоятельств, например, факт невыполнения плана, проблемы с материально-техническим обеспечением, переоценка факторов риска и т.п.

Замечание по конструированию

> ISO 9000 рекомендует коэффициент 3 для периодичности отчетности разных уровней, т.е. на каждые три периода получения отчетов от подчиненных должен приходиться один период представления отчета начальнику.

3.4.5.3. Проверка качества материалов

Для обеспечения повышения качества процесса проведения проектов определенные виды отчуждаемых материалов (документов) проверяются и визируются Комитетом Качества. Порядок проверки и визирования определяется Комитетом Качества. Например, Комитет Качества может назначить своего представителя для проверки конкретного документа или всех документов определенного типа. Проверке и визированию подлежат все внешние (т.е. предоставляемые заказчику) материалы и планы фаз. По усмотрению Комитета Качества для некоторых или всех проектов могут проверяться все или некоторые внутренние материалы. Проверка имеет целью установить степень соответствия материала принятым стандартам.

Проверяющий не должен вмешиваться в структуру и содержание ревизуемого материала, вносить исправления или добавления в материал. В обязанности проверяющего входит следующее.

Проверка соответствия материала требованиям и рекомендациям и указание на несоответствие.

Указание на очевидные (например, орфографические) ошибки.

Указание на несоответствие хорошему тону, здравому смыслу и деловому стилю.

Если проверяющий не завизировал материал, то руководитель проекта обязан обеспечить доработку материала в соответствии с полученными указаниями и представить материал на повторное визирование.

Замечание по конструированию

> По мере накопления опыта использования процедур данное положение может быть изменено или отменено.

[22] Периодичность отчетности может быть изменена руководителем подразделения разработки.

[23] Периодическая отчетность руководителей подразделений здесь не рассматривается.

3.4.6. Выводы

Несмотря на гибкость модели процесса, в ней имеется неизменяемое жесткое ядро Далее следует список основных положений, которые являются характеристическими для модели и которые должны неукоснительно выполняться при проведении любого проекта.

1. Общая процедура проведения проекта состоит из трех этапов: подготовка к проекту, работа над проектом, завершение проекта. Этапы не могут быть опущены.
2. Работа над проектом организована как процесс, состоящий из фаз, заканчивающихся вехами, причем каждая веха имеет видимые результаты, отчуждаемые от разработчика.
3. Схема проекта имеет нелинейный (итеративный и параллельный) характер и не противоречит модели процесса. Отклонения схемы проведения проекта от типовых схем документированы и обоснованы.
4. Процесс работы над проектом ориентирован на заказчика: результаты вех согласуются с заказчиком, заказчик вовлекается в практическое использование результатов проекта на ранней фазе опытной эксплуатации.
5. Назначенный Руководитель проекта участвует в проекте от начала до конца, несет ответственность за все результаты проекта и имеет все ресурсы и полномочия, необходимые для выполнения проекта.
6. Назначение сотрудников на выполнение роли происходит динамически на каждой фазе проекта.
7. Руководитель проекта осуществляет и документирует статическое планирование всего проекта в целом и динамическое планирование каждой фазы в отдельности.
8. Измеряются и учитываются значения количественных показателей индивидуальной производительности при выполнении каждой функции и определяются интегральные и средние значения показателей для проекта в целом.
9. Результаты проектов архивируются и подготавливаются для анализа и повторного использования.

Замечание по конструированию

> Принятые модель жизненного цикла и модель процесса разработки взаимно определяют друг друга и являются согласованными. Некоторая разница в терминологии объясняется разными точками зрения на предмет: с точки зрения программиста и с точки зрения программы. Однако фазы и вехи строго соответствуют другу, хотя и по-разному названы. В таблице 4 указано это соответствие для фаз.

Таблица 4. Соответствие терминов жизненного цикла и модели процесса

Жизненный цикл	Модель процесса	Форма существования программы
Постановка задачи, Анализ результатов	Анализ	Отлична от исполнимого кода (документация, модель, идеи в голове)
Проектирование (логическое)[24]	Проектирование	Модель, схемы, диаграммы
Кодирование	Реализация	Введенный, но не исполнимый код
Тестирование и отладка	Стабилизация	Исполнимый, но не отчужденный код
Использование	Опытная эксплуатация, Внедрение	Отчужденный код и документация

[24] В дисциплине программирования проектирование подразделяется на концептуальное, логическое и детальное. Логическое проектирование выполняется на фазе проектирования, концептуальное – на фазе анализа, а детальное – на фазе реализации (см. раздел *Проектирование*).

Тема 4. Модели команды разработчиков

Если программа разрабатывается индивидуально, одним человеком, то проблем взаимодействия с самим собой обычно не возникает.[25]
В современных условиях программное обеспечение разрабатывается коллективами, иногда очень большими коллективами. В таких коллективах проблемы взаимодействия являются постоянными и иногда очень сложными. Сами проблемы, несомненно, относятся к процессу разработки программного обеспечения, а способы решения этих проблем относятся к технологии программирования.

Модель команды – описание производственных отношений между людьми, вовлеченными в процесс разработки программного обеспечения.

В этой теме рассматриваются различные абстрактные модели команды, и приводится пример конструирования конкретной модели для гипотетической организации.

4.1. Коллективный характер разработки

В рамках этой темы мы принимаем как постулат предположение, что разработка программного обеспечения ведется некоторой группой людей.

Команда проекта – группа людей, как правило, сотрудников одной программирующей организации, осуществляющая процесс разработки программного обеспечения в рамках одного проекта.

Команда проекта может иметь различные свойства. Команда может быть большой или маленькой, может иметь постоянный или переменный состав по ходу проекта, может быть постоянной или временной, созданной для одного проекта.

4.1.1. Оптимальный размер команды

Психологи утверждают, а практика подтверждает, что оптимальное количество сотрудников, с которыми приходится регулярно общаться каждому из команды проекта, составляет от трех до семи человек.

Замечание

> Регулярное общение означает разговор с кем-либо в течение примерно двух часов в неделю.

В принципе программист-разработчик может работать почти без регулярного индивидуального общения с кем бы то ни было. Однако такая изоляция обычно приводит к непониманию разработчиком предъявляемых к нему требований и, как следствие, к относительно низкому уровню эффективности. С другой стороны, если участник проекта работает в слишком большой команде, то из-за постоянного общения с каждым из коллег у него не останется времени на выполнение своих основных обязанностей по проекту. Это снова приводит к относительно низкому уровню эффективности. Например, если сотрудник регулярно общается с десятью своими коллегами, то половина его рабочего времени (20 из 40 часов в неделю) проходит в разговорах. Менеджеры проектов, планирующие проекты с большими командами, должны это учитывать.

4.1.2. Подчиненность участников проекта

В современных условиях используются три основных вида организации подчиненности (субординации):

- Линейно-функциональная;
- Проектно-ориентированная;

[25] А если возникают, то такие проблемы относятся к области психологии или психиатрии, но не к технологии программирования.

- Матричная.

В *линейно-функциональных организациях* весь персонал разбит на непересекающиеся функциональные подразделения, каждое из которых выполняет определенный специфический вид работы. Персонал отчитывается перед руководителем подразделения. Такой способ организации производства хорош для организаций, выполняющих не отдельные проекты, а постоянный, непрерывный и отлаженный производственный процесс. Например, конвейерное производство автомобилей или розничная торговля. Никаких специальных команд проекта не создается. При производстве программного обеспечения линейно-функциональная структура практически не применяется и поэтому здесь не рассматривается.

В *проектно-ориентированных организациях* персонал проекта организационно отчитывается перед руководителем проекта. Руководитель проекта является административным начальником команды проекта. Разработчик всегда прикреплен к какому-либо конкретному проекту и организационно не связан с другими разработчиками в других проектах внутри компании. Преимуществом является упрощение вертикали власти, однако основной недостаток заключается в профессиональной изоляции разработчиков. Например, очень маленькие компании почти всегда организованы по принципу проектной ориентации, однако даже огромные компании иногда организуются так же, особенно когда хотят произвести впечатление на заказчика.

Вообще говоря, системы отчетности на практике имеют более сложную и комплексную структуру, чем в проектно-ориентированных организациях. Это происходит из-за использования *матричной структуры организации*. Такая структура предусматривает, что служащие относятся к функциональным подразделениям (например, отдел разработки, отдел продаж и т. п.) и выделяются этими подразделениями для участия в том или ином проекте. Таким образом, административный начальник программиста, отвечающий за своего подчиненного, является руководителем соответствующего функционального подразделения (например, отдел разработки). Однако в каждом проекте, в котором разработчик принимает участие, он одновременно подчиняется руководителю проекта. Обычно разработчиков на постоянной основе привлекают не более чем к двум-трем проектам. Основным достоинством матричных организаций является профессионализм и возможность улучшения навыков. К недостаткам относится нечеткость в линиях субординации. Довольно часто компании стараются найти золотую середину между матричной и проектно-ориентированной структурами.

Мы будем считать, что так или иначе, здесь рассматриваются такие организации, в которых существуют команды проекта.

4.1.3. Развитие команды и развитие персонала

Каждый проект, кроме достижения основной своей цели, должен использоваться руководителем проекта для развития команды. А именно, в процессе выполнения проекта:

- происходит приобретение участниками новых технических знаний, освоение новых технологий, т.е. техническое обучение и совершенствование профессионального мастерства,
- вырабатываются и укрепляются навыки командной работы,
- приобретается опыт выполнения проектов в соответствии с принятой в компании схемой организации процесса разработки программного обеспечения.

Поэтому косвенным результатом любого проекта является развитие персонала, что приводит к более успешному выполнению последующих проектов.

4.1.4. Специализация, кооперация и взаимодействие

Программирование отнюдь не однородно. В модели процесса мы выделили несколько фаз. Успешное выполнение каждая из фаз требует наличия определенных навыков у работников. Частично эти навыки для разных фаз совпадают, но есть и специальные навыки,

которые специфичны для некоторой фазы. Например, если в проекте применяются формальные методы, то на фазах анализа и проектирования может потребоваться владение математическим аппаратом в таких объемах, которые не предусматриваются программами учебных заведений, готовящих программистов.[26] Другой пример: на фазе внедрения требуется написать пользовательскую документацию. К сожалению, практика показывает, что большинство программистов просто не в состоянии написать хорошее руководство для пользователя – нет навыка, нет опыта. Поэтому специализация «технический писатель» является сейчас даже более дефицитной, чем «системный архитектор».

Короче говоря, люди (в том числе и программисты) разные. Хороший руководитель проекта старается так распределить работу, что каждый работник делал то, в чем он силен и не делал того, в чем он слаб. Это называется *специализацией*.

Набор хороших специалистов – это условие является необходимым, но не является достаточным для организации хорошей команды. Необходимо добиться их согласованной работы, то есть *кооперации*.

Кооперация достигается через *взаимодействие*. Вот почему коммуникативные навыки считаются столь важными для современных программистов.

4.2. Иерархическая модель команды

Иерархическая модель (или модель *дерева субординации*, см. рис. 29) является самой распространенной и известной моделью. Эта модель обладает целым рядом достоинств.

Принцип единоначалия. Принцип единоначалия обеспечивает очень высокую степень надежности, устойчивости и управляемости команды. В критических ситуациях всегда используется именно эта модель.[27]

Иерархическая модель привычна и известна абсолютно всем. Ее использование не требует дополнительных мероприятий по внедрению и обучению персонала.

Иерархическая модель обладает высокой степенью масштабируемости. Она с успехом применяется в масштабах от десятков до десятков миллионов исполнителей.

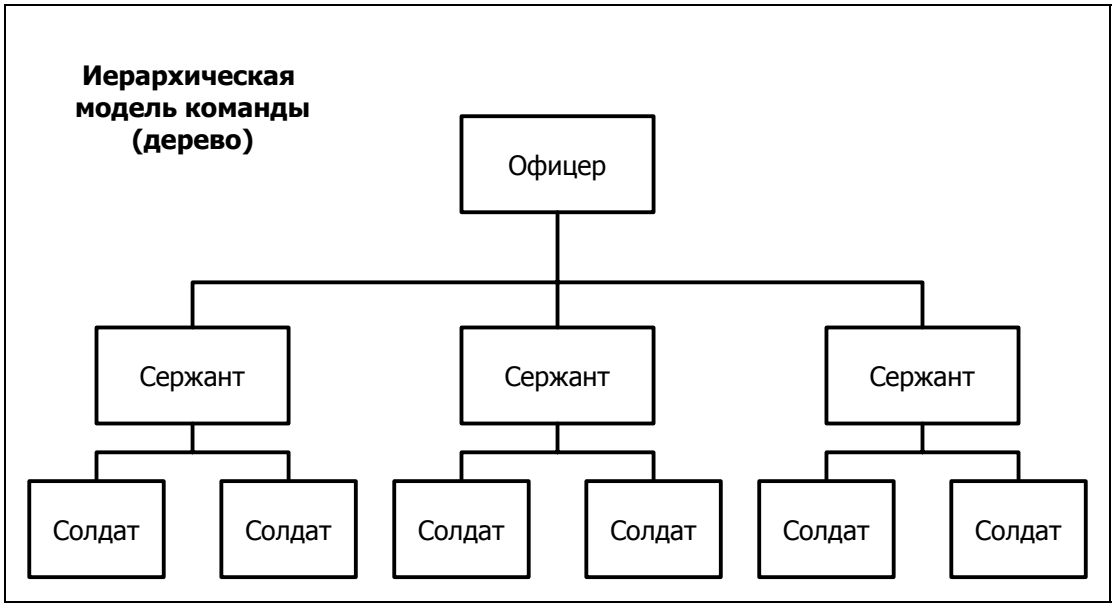


Рис. 29. Иерархическая модель команды

В то же время иерархической модели присущи некоторые принципиальные недостатки.

Иерархическая модель экстенсивна. Наращивание функциональности обеспечивается увеличением состава.

[26] По личному мнению автора, основной причиной малого использования формальных методов в программировании является не слабость формальных методов, а низкая математическая культура программистов.

[27] Чтобы подчеркнуть это обстоятельство, на рис. 29 использована "армейская" терминология.

Иерархическая модель жесткая, т.е. практически не допускает перестройки "на ходу" (в процессе). Она ориентирована на выполнение строго определенных функций. Изменение функций требует болезненной перестройки дерева субординации.
Иерархическая модель консервативна. При ее использовании имеется тенденция к жесткому закреплению за каждым исполнителем его ролевой функции. Она плохо приспособлена для быстрой смены технологий и парадигм.
Иерархическая модель не устойчива по отношению к личным качествам руководителей. Отрицательные личные качества руководителей оказывают отрицательное воздействие на эффективность команды, причем это воздействие непропорционально увеличивается с ростом уровня в дереве субординации.

4.3. Метод хирургической бригады

Модель главного программиста появилась во время первой технологической революции в программировании на рубеже 60–70-х годов. Долгое время модель главного программиста (или *метод хирургической бригады*, или *модель звезды*, рис. 30) являлась доминирующей моделью при разработке программного обеспечения.
В этой модели главный программист выполняет весь проект сам, а прочие члены бригады ассистируют в предопределенных рамках своих ролевых функций.[28]
Модель главного программиста имеет следующие достоинства.
Бригада главного программиста обладает высокой предсказуемостью. Если главный программист плох, то это выявляется на ранних стадиях проекта. Проект может быть прекращен или реорганизован практически без убытков. Если главный программист хорош, то вероятность успешного завершения проекта высока даже при наличии серьезных внешних факторов риска.
Бригада главного программиста обладает высокой автономностью. Она успешно функционирует даже в изменяющейся и неблагоприятной внешней среде.
Бригада главного программиста обладает достаточной функциональной гибкостью. За счет изменения набора "лучей" в звезде ее легко можно ориентировать на различные типы программных проектов.
Бригада главного программиста наследует все достоинства принципа единоначалия (поскольку главный программист единолично принимает все принципиальные решения по проекту).

[28] Этим бригада главного программиста отличается от иерархической системы, где начальник разделяет общий проект на части и ставит задачи подчиненным, которые и проводят конкретную работу по их выполнению.

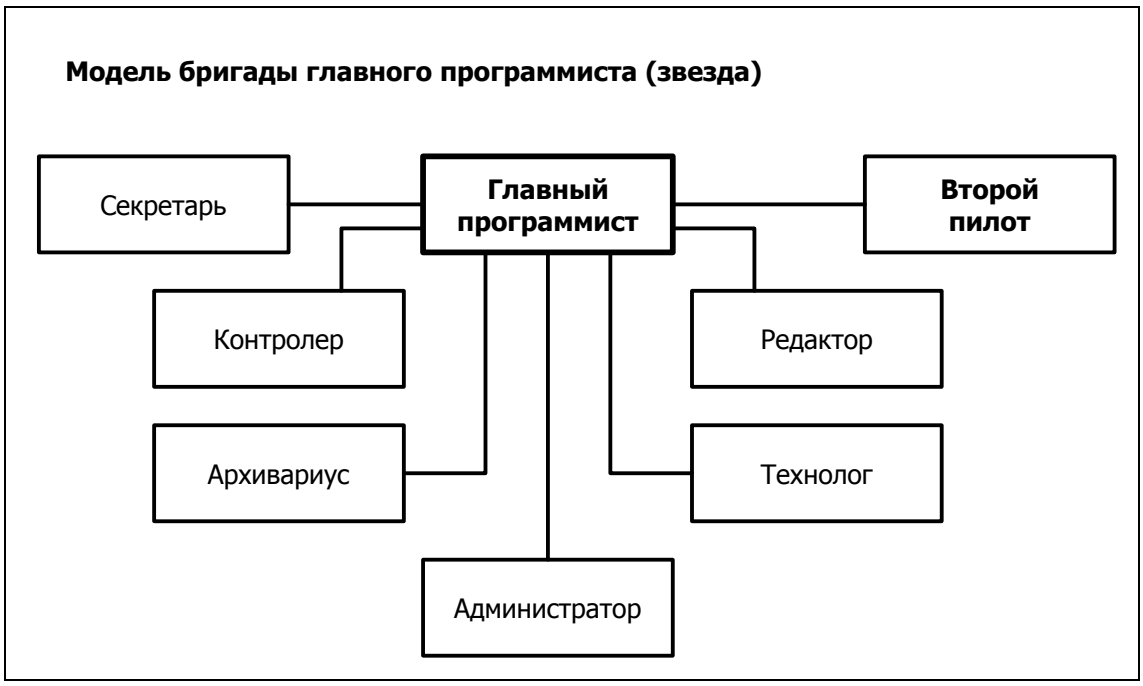


Рис. 30. Модель бригады главного программиста

Метод бригады главного программиста допускает различные модификации при сохранении своей сути.

Первая модификация: изменение количества и качества лучей в звезде. В классической модели, описанной в книге Брукса [3], в звезде еще присутствовали второй Секретарь и Языковед. В практических случаях количество лучей сокращают, например, объединяя Секретаря, Редактора и Архивариуса. Характеристическими ролями в бригаде являются Главный программист, Второй пилот и Администратор, остальные роли меняются по мере развития технологий программирования.

Вторая модификация: на роль главного программиста назначается не кодировщик, а, например, аналитик или менеджер продукта. В этом случае кодирование ведет Второй пилот, но все принципиальные решения принимает Главный программист. В современных условиях наблюдается тенденция перемещения принятия ключевых решений со стадии кодирования на более ранние стадии, поэтому вторая модификация является фактически стандартной.

Третья модификация: "сдвоенный центр" (рис. 31). Эта модификация позволяет хотя бы в некоторых пределах масштабировать бригаду. Имеются два Главных программиста, которые делят проект пополам, все решения принимают консенсусом и являются вторыми пилотами друг для друга.

Замечание

> При использовании большего количества главных программистов модель перестает работать, потому что коммуникации, достижение консенсуса и взаимное дублирование работы требует слишком больших накладных расходов.

Рис. 31. Модификация модели бригады главного программиста

Модель бригады главного программиста имеет определенные недостатки.

Бригада главного программиста не является масштабируемым решением. Она отлично работает на проектах объема 6-8 человек × 1-2 года. Если проект требует более коротких сроков или существенно больших объемов, то использование бригады главного программиста затруднено.

Замечание

> Если формально разрезать крупный проект на несколько частей и запустить несколько бригад параллельно, то результаты их работы будет очень трудно синхронизировать и интегрировать в одно приложение. Дело в том, что главный программист держит очень много в голове, часто опуская этапы формального документирования и спецификации. За счет этого повышается производительность, но затрудняется совместимость. Очень короткий проект бригаде главного программиста трудно провести потому, что главный программист последовательно выполняет всю основную работу, и его личные возможности ограничивают производительность бригады.

Бригада главного программиста не является распараллеливаемой структурой. Она действует по принципу: один проект – одна команда. Практически невозможно выполнять бригадой одновременно разные фазы разных проектов.

Бригада главного программиста имеет уязвимое центральное звено. Очень велик управленческий риск мгновенной аннигиляции бригады, если что-то случается с главным программистом.

Замечание

> Второй пилот в бригаде главного программиста должен тщательно отслеживать все действия главного программиста. Это несколько снижает риск аннигиляции.

4.4. Модель команды равных

Модель команды равных является составной частью Microsoft Solutions Framework (MSF) — методологии разработки программных проектов фирмы Microsoft. Это наиболее демократичная модель, поскольку в ней нет явно выделенного центра. Модель команды MSF — это команда равных. Схематически ее принято изображать в виде цикла (рис. 32), где все роли равноправны и связаны друг с другом.

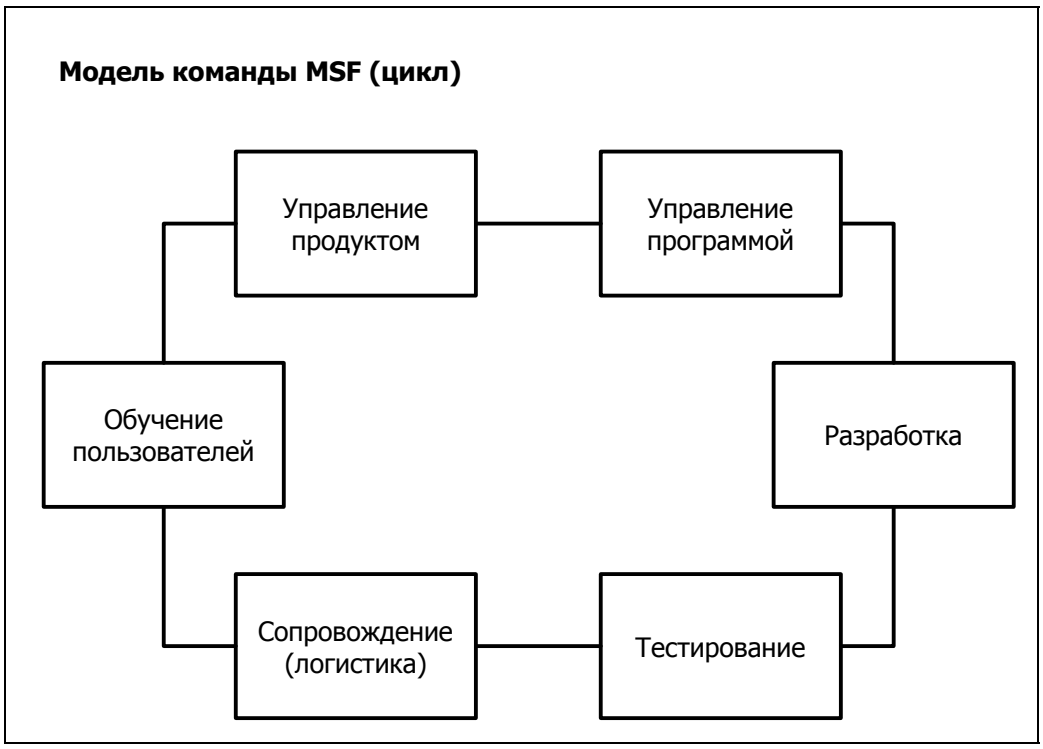


Рис. 32. Модель команды равных

Замечание

> Чтобы подчеркнуть отсутствие иерархии в команде равных, роли в команде обозначаются не названиями должностей, а названиями функций.

Преимущества модели команды равных.

Высокая производительность, поскольку непроизводительные трудозатраты на поддержание формальных и субординационных связей сведены к минимуму.

Сравнительно легкая масштабируемость. Каждый элемент в схеме команды может быть в свою очередь циклом.

Сильная положительная мотивация труда и равно высокая заинтересованность всех участников в конечном успехе.

Основные недостатки модели MSF являются продолжением ее достоинств.

Для формирования команды MSF нужны равные (равно квалифицированные и равно заинтересованные) участники.

Критическое значение имеет коммуникабельность (большая часть коммуникаций неформальны), умение и готовность работать в коллективе (артельный дух).

Демократичная модель команды MSF плохо сопрягается с жесткой иерархической моделью подразделения (предприятия).

4.5. Конструирование модели команды

Как уже отмечено в преамбуле к теме 3, на практике программирующие организации редко применяют абстрактные модели из литературы в «чистом виде». Как правило, используя известные модели и учитывая конкретные особенности и условия, организация создает специфические модели «под себя», которые наилучшим образом учитывают особенности организации.

К модели команды это относится даже в большей степени, нежели у модели процесса. Дело в том, что при конструировании модели команды необходимо учитывать *человеческий фактор*. Совершенно бесполезно и даже вредно пытаться «засунуть» живых людей в аб-

страктную схему. Гораздо полезнее подогнать схему под фактические особенности и возможности реального персонала в организации.[29]

Мы продолжаем пример, начатый в разделе 3.4, и до конца этой темы демонстрируем конструирование конкретной модели команды проекта для той же гипотетической программирующей организации.

Замечание по конструированию

> Замечания, которые присутствуют далее в тексте описания сконструированной модели, не являются частью описания модели команды, т.е. не являются частью примера. Эти замечания содержат объяснения и мотивацию того, почему было принято то или иное решение при конструировании, а также иногда указания на то, какие альтернативные решения можно было бы принять.

4.5.1. Особенности организации и требования к команде

Модель команды в значительной своей части опирается на известные и апробированные модели. Ее отличия от традиционных моделей обусловлены учетом ряда особенностей, специфических для организации в данный момент. Учитываемые особенности и вытекающие из них требования таковы.

Организация является успешно действующим предприятием. Резкая массовая и одномоментная перестройка производственных отношений нежелательна, поскольку может вызвать временный спад эффективности основной производственной деятельности. Таким образом, конструируемая модель команды должна являться консервативным расширением действующей модели.

Основной контингент исполнителей состоит из высокообразованных, дисциплинированных и динамичных молодых людей, обладающих широким кругозором. Таким образом, модель команды должна допускать и использовать возможность гибкой перестройки команды в процессе разработки.

Организация является малым, динамично развивающимся предприятием, которое не может использовать экстенсивные методы формирования команды. Таким образом, модель команды должна учитывать принципиальную ограниченность трудовых ресурсов.

На данном этапе развития организация ориентирована на разработку вертикальных приложений масштаба предприятия. Типичный объем таких проектов составляет от 2 месяцев × 1 человек до 9 месяцев × 3-5 человек. Сверх оперативные (on-line) и сверх большие проекты не рассматриваются. Организация выполняет параллельно 5-10 таких проектов. Таким образом, модель команды должна допускать масштабирование в пределах 10–50 человек.

4.5.2. Архитектура модели команды

Модель команды использует комбинацию трех перечисленных выше моделей со следующими добавлениями и изменениями.

В модели команды используется дифференцированная номенклатура функций, ролей и должностей.

Модель команды является статико-динамической, то есть (частично) изменяющейся по ходу проекта.

Модель команды является мультикомандной и мультипроектной, то есть предусматривает совместное использование нескольких команд для параллельной разработки нескольких проектов.

---

[29] На своем опыте руководства проектами автор неоднократно убеждался, что программистов совершенно невозможно заставить что-либо сделать, но довольно легко можно уговорить.

4.5.3. Функции, роли и должности

Понятие функции, роли и должности не являются независимыми, они в определенной степени определяют друг друга. Однако между этими понятиями в модели команды имеются различия.

4.5.3.1. Функции

Функция — это вид деятельности, выполняемой в ходе проекта. Выполнение каждой функции требует наличия определенной специфической квалификации и способностей.

В модели предусматриваются следующие основные функции.

Администрирование. Внешнее: ведение договоров; разговоры с заказчиком; составление внешних (полу)формальных документов. Внутреннее: доклады начальству; столы, стулья, компьютеры.

Проектирование. Составление бумажных и/или электронных человеко- и/или машинно-читаемых концепций, моделей, спецификаций и планов.

Замечание по конструированию

> Различаются проектирование приложения (решения) и проектирование процесса разработки приложения (решения). По роду деятельности и требуемой квалификации это родственные функции, но они применяются к разным объектам. Планирование считается частью проектирования.

Кодирование. Ручная и/или полуавтоматическая генерация кода на языке программирования. Автономная (поблочная) отладка кода. Рисование и тестирование интерфейсных элементов (форм).

Замечание по конструированию

> Дизайн интерфейса отнесен к кодированию, поскольку в данный момент в рамках организации дизайн интерфейса не выделен в явную профессиональную специализацию. В других программирующих организациях, например, в таких, которые специализируются на разработке компьютерных игр, дизайн интерфейса обычно выделяют в отдельную функцию.

Тестирование. Пробное использование приложения с целью сломать его, а не решить задачу.

Сопровождение. Развертывание, обучение пользователей, использование, демонстрирование, администрирование разработанного или установленного программного обеспечения в процессе опытной эксплуатации.

Замечание по конструированию

> Составление бумажной и электронной документации (документирование) не выделено в отдельную функцию, поскольку в модели команды считается, что все исполнители в достаточной степени владеют рабочим языком проекта (русским) для составления необходимых документов по ходу выполнения основной функции. Это предположение основано на том обстоятельстве, что порядок составления всех типовых документов описан соответствующим руководством или шаблоном. Если это предположение нарушается (например, если рабочим языком проекта должен быть английский), то набор функций и ролей должен быть расширен.

4.5.3.2. Роли

Роль[30] — это временное назначение сотруднику набора функций в рамках конкретного проекта.

[30] В Унифицированном процессе применяется термин «работник». Мы также применяли термин «работник» при описании модели процесса, чтобы не путать с термином «роль» в унифицированном языке моделирования UML. Однако при описании модели команды UML почти не используется, риска путаницы не возникает, и здесь применяется более удобный термин «роль».

Получение роли означает делегирование полномочий для выполнения определенных функций и принятие ответственности за результаты выполнения этих функций. Роль может требовать выполнения разных функций; некоторые функции могут быть присущи нескольким ролям. Определяющим признаком роли является не характер деятельности, а набор конкретных результатов (вех), ответственность за достижение которых налагает роль (см. модель процесса). В проектах разных типов используются разные наборы ролей из следующего множества:

Аналитик. Функции: планирование, администрирование. Вехи: Концепция, Обследование, Техническое задание, Функциональные спецификации, Внутренний язык, Дополнительные требования, План тестирования, План внедрения, Логический проект, Схема базы данных, Документация.

Замечание по конструированию

> Можно различать две роли: бизнес-аналитик и системный аналитик. Эти роли требуют сходной квалификации для выполнения сходных функций, выполняемых в несколько различном контексте. Явное различение этих ролей возможно в модели команды и не является существенным изменением.

Администратор. Функции: администрирование, планирование. Вехи: Техническое задание, Календарный план-график, План тестирования, План внедрения, Документация.

Программист. Функции: проектирование, кодирование, тестирование, сопровождение. Вехи: Прототип, Логический проект, Схема базы данных, Дизайн интерфейса, Физический проект, Код готов, Нет известных ошибок, Исходные тексты, Документация.

Тестировщик. Функции: тестирование. Вехи: Выпуск, Нет известных ошибок, Исходные тексты, Документация.

Эксплуатационник. Функции: сопровождение, тестирование. Вехи: Документация, Решение развернуто, База данных загружена, Пользователи обучены.

Замечание по конструированию

> В этом списке для каждой роли порядок перечисления функций и вех является существенным. Перечисление указывает возможные функции и вехи для каждой роли в порядке убывания типичности и обязательности. В списке функций для каждой роли перечислены наиболее типичные, с учетом конкретных обстоятельств список функций роли может быть расширен.

Перечисленный список ролей может быть расширен в соответствии со спецификой проекта.

Поскольку списки функций и вех для различных ролей пересекаются, в конкретных проектах могут быть задействованы не все роли. Для описания большинства типов проектов оказывается достаточным использовать только три основные роли: аналитик, программист и тестировщик / эксплуатационник. Эти роли выделяются тем, что они существенно различным образом соотносятся с жизненным циклом приложения, как показано в таблице 5 .

Таблица 5. Соотношение фаз, ролей и функций

Состояние приложения	Роль	Выполняемые функции
Еще не существует	Аналитик	Анализ (существующей ситуации) и Проектирование (будущего приложения)
Находится в процессе разработки	Программист	Кодирование (в широком смысле)
Уже существует	Тестировщик / Эксплуатационник	Тестирование (проверка работоспособности и других характеристик приложения), Развертывание, Документирование (и другие действия с имеющимся приложением)

Замечание по конструированию

Роли тестировщика и эксплуатационника во многом совпадают, хотя роль эксплуатационника шире. В обоих случаях речь идет о работе с уже имеющимся приложением. Эксплуатационник отличается от тестировщика прежде всего тем, что выполняет свои функции на территории заказчика.

Замечание по конструированию

Роль администратора по умолчанию играет руководитель проекта, поэтому в типичных случаях роль администратора не планируется на каждой фазе, а подразумевается по умолчанию. Это не исключает возможности делегировать выполнение административных функций другому сотруднику по решению руководителя проекта.

4.5.3.3. Должности

Должность — это сертифицированная способность играть определенные роли и выполнять определенные функции.

Замечание по конструированию

Порядок определения должностного соответствия (или несоответствия) и назначения на должность моделью команды не охватывается.

В отличие от роли и функции должность не привязана к конкретному проекту и носит (почти) постоянный характер по времени. В модели команды предусматривается три уровня иерархии должностей:
Начальник (отдела).
Руководитель (группы).
Исполнитель (инженер, технический специалист).

Замечание по конструированию

Наличие именно трех уровней иерархии должностей обусловлено ограничением на масштабируемость: 10-50 человек и наличием известной константы, ограничивающей ширину ветвления в дереве субординации: 7±2. Т.е. в иерархической структуре руководитель любого уровня должен иметь не менее 5 и не более 9 прямых подчиненных.

Замечание по конструированию

Высшее руководство (дирекция) моделью команды не охватывается.

4.5.4. Статико-динамическая структура команды

Конструируемая модель команды предусматривает назначение каждому участнику проекта функции, роли и должности, причем это назначение не является постоянным, а меняется со временем по определенным правилам:

- функция (функции) назначается динамически на каждую фазу,
- роль (роли) назначается статически на время проекта
- должность назначается статически, т.е. постоянно.

Замечание по конструированию

Точнее говоря, в плане выполнения фазы назначается веха. Поскольку тип вехи однозначно определяет требуемые функции, здесь употреблен термин "функция", как более привычный.

Назначение вех на фазу называется *командой фазы*, назначение ролей на проект называется *бригадой проекта*, а назначение сотрудников на должности называется *должностной структурой*.[31] Все эти понятия более подробно описаны в следующих разделах. Организационный порядок проведения назначений не является методическим элементом модели команды и поэтому описан в разделе *Порядок проведения типового проекта*.

[31] Назначение на должность не зависит от участия или не участия сотрудника в проектах.

Замечание по конструированию

Изобразить на одной диаграмме статико-динамическую структуру модели команды оказывается затруднительным. Поэтому ниже приведены примеры трех указанных структур для некоторого условного проекта. В приведенных на рис. 33-35 диаграммах латинские буквы **A, B, C**… обозначают конкретных людей, одних и тех же на всех диаграммах. Таким образом, из примеров видно, как может меняться функция и роль конкретного человека по ходу проекта.

4.5.4.1. Должностная структура

Имеется статическая (постоянно действующая) *должностная структура*. Должностная структура строится по иерархической модели (дерево). Прямое управление, бюджетное и материально-техническое обеспечение осуществляется по этой структуре. Изменение должностной структуры происходит вне модели команды. Работы, которые не входят в проекты, проводятся по этой структуре. На рис. 33 приведен пример должностной структуры.

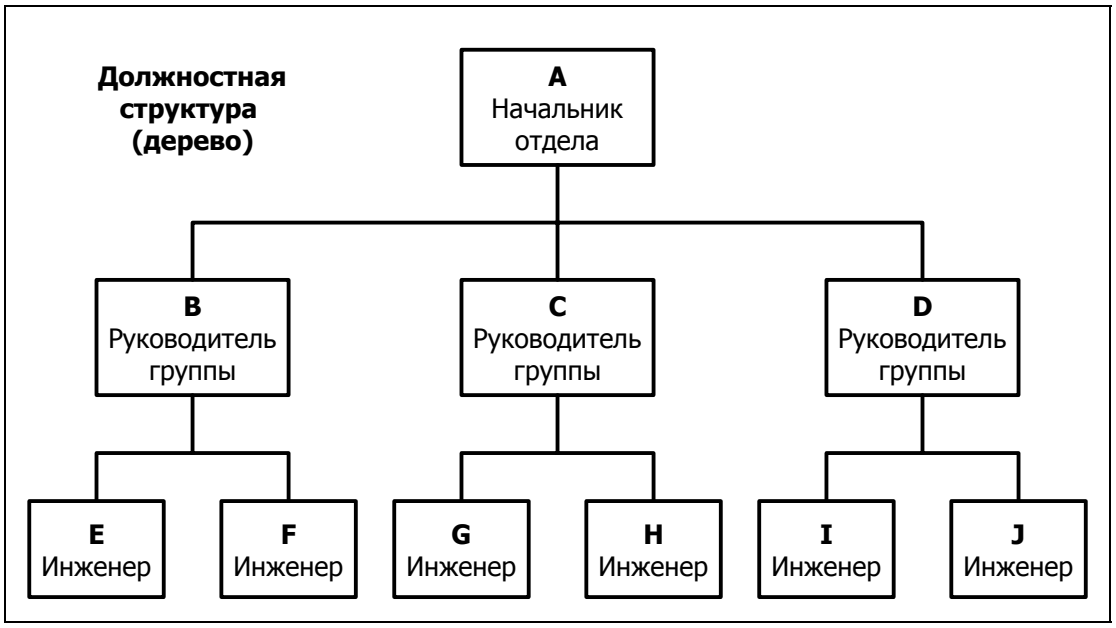


Рис. 33. Пример должностной структуры

4.5.4.2. Руководитель проекта

Для каждого проекта назначается *Руководитель проекта* (лидер команды). Он (единственный) вовлечен в проект от самого начала до самого конца. По ходу проекта Руководитель проекта может играть разные роли и выполнять разные функции. Конкретный сотрудник является Руководителем только в одном проекте. Руководитель проекта является лидером проекта в том смысле, что он единолично принимает все принципиальные решения по проекту. В то же время он не является административным начальником для всех вовлеченных в проект.

Замечание по конструированию

В модели команды рекомендуется, чтобы Руководитель проекта был вовлечен только в один (свой) проект. Руководитель проекта должен обладать целым рядом разнообразных способностей и иметь достаточную квалификацию в различных областях. На практике такое сочетание качеств встречается сравнительно редко, поэтому наблюдается дефицит потенциальных руководителей проектов. В связи с этим принцип "один руководитель – один проект" часто вынужденно нарушается. Это обстоятельство не противоречит прямо модели команды, но снижает эффективность ее применения. В модели рекомендуется, чтобы Руководитель проекта занимал должность не ниже Руководителя группы, в том случае, если в проект вовлечено несколько человек.

Руководитель проекта – это не функция, не роль и не должность. Руководитель проекта несет полную ответственность за весь проект в целом и, тем самым, за каждую веху в отдельности. В этом смысле он играет все роли. В то же время, в отличие от административного начальника, Руководитель проекта не только поручает и контролирует выполнение функций, но и сам выполняет определенные функции, причем, как правило, критические функции проекта. В этом смысле он играет конкретную (переменную во времени) роль.
Организационный порядок назначения Руководителя проекта описан в разделе *Порядок проведения типового проекта*.

4.5.4.3. Бригада проекта

Для каждого проекта (в соответствии с его типом) назначается статическая *бригада проекта*. Бригада проекта строится по модели главного программиста (звезда). Центром звезды является Руководитель проекта. При назначении бригады проекта определяется, какие сотрудники и в каких ролях в принципе будут задействованы в проекте. В то же время назначение сотрудника в бригаду не означает его закрепощения в этой бригаде. Он может быть назначен и в другие бригады, причем в другой роли. На рис. 34 приведен пример бригады проекта.

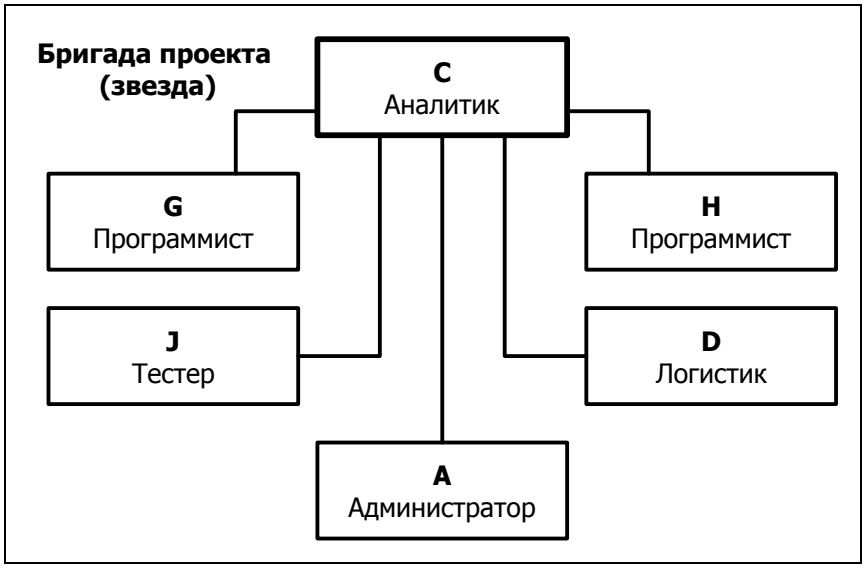


Рис. 34. Пример бригады проекта

4.5.4.4. Команда фазы

В каждый момент времени (с точностью до главных и промежуточных фаз и вех проекта) динамически определяется конкретная *команда фазы*, т.е. множество сотрудников из состава бригады проекта, которые фактически задействованы в определенной роли и выполняют определенную функцию (ответственны за достижение определенной вехи) в данном проекте на данной фазе. Команда фазы строится по модели команды равных (цикл). По мере движения проекта по фазам проекта команда фазы меняется. На рис. 35 приведен пример команды фазы, которая назначена на фазе проектирования некоторого проекта.

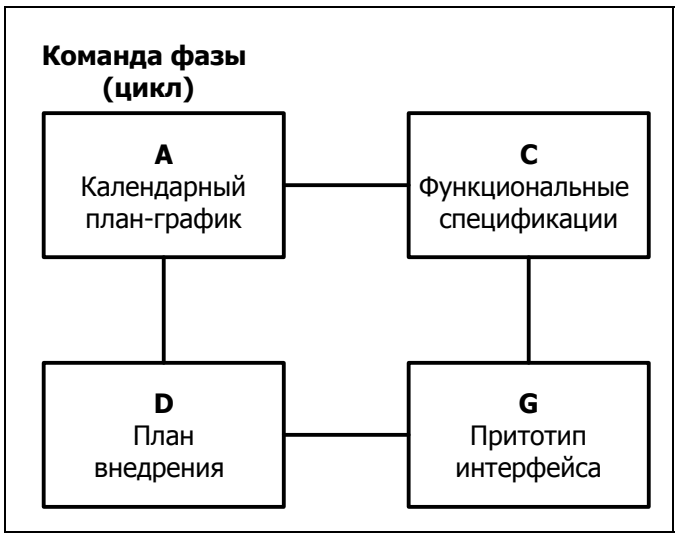


Рис. 35. Пример команды фазы

Замечание по конструированию

> Динамическое переназначение сотрудников на разных фазах не является самоцелью. Это средство управления персоналом в условиях дефицита человеческих ресурсов и разнородности выполняемых проектов. При благоприятных обстоятельствах фактического переназначения и изменения роли для данного сотрудника может и не потребоваться (см., например, подраздел *Занятость исполнителей* в разделе *Модель процесса*).

4.5.5. Распределение полномочий и ответственности

Распределение полномочий и ответственности в модели команды имеет статико-динамический характер.

Руководитель проекта назначается руководством организации. Процедура назначения описана в разделе *Порядок проведения типового проекта*.

Замечание по конструированию

> Смена Руководителя проекта в ходе проекта является чрезвычайным событием и не рекомендуется моделью команды.

Бригада проекта назначается по представлению Руководителя проекта и утверждается руководством организации.

Команда фазы назначается из состава бригады по представлению Руководителя проекта и утверждается (согласуется) с непосредственными начальниками всех исполнителей, задействованных на данной фазе.

Замечание по конструированию

> Согласование по транзитивности (т.е. только со старшим начальником, а не с непосредственным начальником) не рекомендуется моделью команды.

Руководитель проекта имеет все полномочия и несет всю ответственность за отчуждаемые результаты проекта.

За материально-техническое, финансовое и организационное обеспечение проекта отвечает непосредственный начальник Руководителя проекта.

Руководитель проекта может делегировать свою ответственность и соответствующие полномочия для достижения любой вехи любому члену команды фазы по своему усмотрению. При этом общая ответственность за окончательные отчуждаемые результаты с руководителя проекта не снимается.

Моральное и материальное стимулирование (как поощрение, так и наказание) производится руководством организации по результатам законченного проекта по представлению Руководителя проекта.

4.5.6. Достоинства и недостатки модели команды

Предлагаемая модель команды имеет следующие преимущества.

Сохраняется и действует привычная иерархическая структура, которая фактически является носителем (инфраструктурой) для прочих динамических структур. Через статическую структуру проводятся работы, которые не входят в проекты (так называемые "вводные", например, выставки и другие маркетинговые мероприятия, и постоянные рутинные работы, например, администрирование локальной сети).

Сохраняются все достоинства принципа единоначалия, присущего бригаде главного программиста. Всегда есть оракул, за которым сохраняется последнее слово по любому вопросу, относящемуся к проекту.

Имеется возможность учесть и использовать различие в способностях и гибкость наличного персонала. Узкий специалист может быть задействован во множестве проектов. Специалист широко профиля может выполнять разные функции в одном проекте. Реальная нагрузка может быть распределена (неравномерно) пропорционально способности и желанию ее нести.

Главным недостатком предлагаемой модели является ее сложность и динамичность. В частности, модель имеет следующие слабые места.

Динамическое движение команды требует наличия очень точных индивидуальных планов-графиков работы для каждого сотрудника. При средней длительности проектов в 6 месяцев и средней длительности фазы в 1 месяц ошибка в плане-графике не может превышать неделю.

Взаимозависимость проектов по ресурсам и эффект "цепной реакции". Сбой в одном из проектов (выход из плана-графика) немедленно распространяются через "разделяемых" сотрудников на другие проекты и так далее.

Требование высокой "стартовой скорости" сотрудников. Частые "переходы фокуса" с одного проекта на другой требуют от сотрудников умения быстро переключаться без "раскачки".

Тема 5. Дисциплина программирования

Дисциплина программирования – это совокупность различных технологических средств и приемов, применяемых программистами с целью повышения продуктивности своей индивидуальной работы.

Цель данной темы заключается в том, чтобы дать представление о решающей роли личной продуктивности в процессе разработки, и показать, какие факторы влияют на повышение продуктивности.

Разнообразие технологических средств и приемов, применяемых программистами, поистине необозримо. Указать среди них два-три «наилучших в большинстве случаев», так это сделано для моделей процесса и моделей команд в двух предыдущих темах, просто невозможно в кратком учебнике.

Поэтому в данной теме приведены общие соображения, относящиеся к дисциплине программирования, и указаны *некоторые* средства и приемы дисциплины программирования. Упоминаемые здесь средства и приемы имеют ограниченную область применения: в некоторых случаях они дают выигрыш в продуктивности программирования, а в других случаях оказываются бесполезными. Таким образом, необходимо указать область применимости указываемых средств. В качестве такой области мы взяли пример, начатый в двух предыдущих темах. Рассматривается гипотетическая программирующая организация, и описываются приемы, применяемые ее программистами. Мы считаем, что приложения, которые разрабатывают программисты гипотетической организации относятся к одному классу: это вертикальные (то есть ориентированные на одно предприятие) приложения для автоматизации бизнес-процесс типа информационных систем масштаба предприятия. Кроме того, мы считаем, что используются традиционные средства разработки: распространенные языки программирования и системы управления базами данных.

Таким образом, весь материал этой темы является развернутым примером, в который вкраплены некоторые теоретические отступления.

5.1. Природа программирования

Природа программирования (в рассматриваемом типичном случае) содержит имманентно присущие ей противоречия, которые являются причиной основных проблем организации программирования:

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 прямое соответствие между этими сущностями установить трудно или невозможно.[32] Это противоречие между статикой и динамикой.

С одной стороны, в распоряжении программиста находится неограниченное количество экземпляров атомарных и композитных объектов, которые он может комбинировать произвольным образом, конструируя программу. Таким образом, качественных ограничений на возможные комбинации, характерных для реального мира, в идеальном мире программных объектов не существует. Это порождает эффект "комбинаторного взрыва", с которым программисту справиться не просто, поскольку количественные возможности комбинаторного мышления человека чрезвычайно ограничены.[33] С другой стороны, компьютер способен выполнить программу неограниченного объема и сложности — количественных ограничений практически не существует. Более того, интерес представляют именно большие программы, которые программист не может в полном объеме единовременно обозреть и не может выполнить вручную. В то же время компьютер качественно ограничен в своих возможностях выполнения программы — выполнение носит строго детерминированный характер.[34] Коротко говоря, цель программирования (выполнение программы) превосходит по объему непосредственные количественные возможности программиста. Это противоречие между количеством и качеством.

Фактически известен только один метод разрешения этих противоречий — принцип "разделяй и властвуй". Суть этого метода применительно к программированию состоит в наложении на программу внешней структуры, которая бы обеспечивала отслеживание соответствия между статикой и динамикой и позволяла бы справиться с количеством рассматриваемых комбинаций.

Прежде чем рассматривать различные внешние структуры, нужно определить точку зрения для рассмотрения. Программирование можно рассматривать как науку, как искусство и как ремесло.[35]

5.1.1. Наука программирования

Программирование является интеллектуальной деятельностью и обладает некоторыми признаками науки. Наука программирования в России и Франции называется информатика,[36] а в прочих странах Computer Science. Информатика сравнительно молода и не устоялась как научная дисциплина: ее предмет плохо определен, а методы заимствованы из смежных дисциплин, в первую очередь из математики.

Замечание

> Классическими образцами научных исследований в программировании, которые доступны начинающим без специальной подготовки, являются монографии Э. Дейкстры *Дисциплина программирования*, Д. Гриса *Наука программирования* и А.П. Ершова *Введение в теоретическое программирование*.

Хотя информатика стремительно развивается и некоторые успехи налицо, но в настоящий момент влияние информатики на практику программирования пренебрежимо мало ввиду незрелости[37] самой науки. Дисциплина программирования не предназначена для изучения учеными программистами.

---

[32] Приведенный абзац является кратким упрощенным пересказом некоторых положений знаменитого письма Дейкстры GO TO statements considered harmful, которое положило начало технологии программирования, как самостоятельному направлению программистской мысли (см. разд. 1.2.2).

[33] Объем и сложность программы, которую может воспринять любая система программирования, намного превосходят объем и сложность программы, которую может написать любой программист.

[34] Подразумевается компьютер традиционной (морально устаревшей) архитектуры с центральным процессором, адресуемой линейной памятью и хранимой в памяти программой.

[35] В последние годы расцвело, в особенности в нашей стране, *спортивное программирование*. Соотношение между спортивным программированием и промышленным профессиональным программирование спорно и неоднозначно. Спортивное программирование здесь не рассматривается.

[36] Формальным признаком того, что информатика считается наукой, является наличие отделения информатики и вычислительной техники в структуре Российской академии наук.

[37] Признаком зрелости науки является наличие достаточного количества результатов (открытых законов природы, доказанных теорем). Обычно пороговым значением считают число 50.

Замечание

Некоторые достижения науки программирования востребованы опосредованным образом в дисциплине программирования — это, прежде всего, идея доказательного программирования и аннотирования программ.

5.1.2. Искусство программирования

Программирование имеет эстетическую ценность. Сам процесс программирования доставляет субъекту своеобразное удовольствие, а к программам применима категория красоты.

Замечание

Д. Кнут назвал свое выдающееся произведение The Art of Computer Programming (Искусство программирования для ЭВМ) не случайно. Эта монография является, по-видимому, наиболее представительным собранием шедевров искусства программирования.

Искусство программирования элитарно, не утилитарно и консервативно. Показательным примером является следующая классическая задача: "Напишите программу, которая печатает свой текст".[38] Решения обладают всеми характерными признаками произведений элитарного искусства: они абсолютно бесполезны, доступны далеко не всем (даже профессиональным) программистам и покрывают широкий спектр степеней изящности. Овладение искусством программирования (как и любым другим искусством) требует наличия призвания, мецената и гуру для каждого адепта. Дисциплина программирования бесполезна для искусных программистов.

Замечание

Некоторые эстетические понятия искусства программирования востребованы опосредованным образом в дисциплине программирования — это, прежде всего, понятие (хорошего) стиля программирования.

5.1.3. Ремесло программирования

Программирование является общественно востребованной деятельностью, а результаты программирования имеют потребительскую ценность. Именно эта ипостась программирования является предметом дисциплины программирования. Другими словами, здесь рассматривается частный случай[39] процесса программирования, когда имеется заказчик, которому нужны результаты выполнения программы, и который имеет возможность и согласен платить деньги за получение этих результатов.[40]

Таким образом, программирование является производственной деятельностью, которая носит по преимуществу ремесленный характер в настоящее время. В большинстве случаев практического программирования можно усмотреть следующие характерные признаки ремесленного производства, отличающие его от индустриального производства:

штучный (не серийный) характер продукции;

отсутствие унифицированной и сертифицированной технологии производства;

отсутствие специализации, кооперации и разделения труда (программист сам проводит все операции технологического цикла производства программы);

высокая доля живого и низкая доля овеществленного труда (каждая новая программа изготавливается из "сырья" — операторов языка и библиотечных объектов, "полуфабрикаты" используются редко);

отсутствие стандартной системы измерений (метрологии);

использование индивидуальных инструментов и приемов (наличие индивидуально "заточенных" ручных инструментов, цеховые "тайны" и "секреты мастерства");

---

[38] Тем программистам , которые еще этого не делали, настоятельно рекомендуется попробовать решить эту задачу.

[39] Этот случай является частным, потому что цель процесса определяется еще одним субъектом (заказчиком).

[40] Вырожденный, но распространенный случай, когда заказчик покупает программы не с целью использования, а с другими целями, здесь не рассматривается.

претензия на личную интеллектуальную собственность на произведенный продукт (программисты склонны считать код своей личной не отчуждаемой собственностью, не предназначенной для публичного ознакомления и повторного использования "посторонними").

Замечание

> Программисты титулуются инженерами (Software Engineer), но это, скорее, озвучивание желаемого, нежели констатация фактического положения дел.

Как следствие ремесленного характера производства, в программировании наблюдаются:
широчайший разброс в показателях производительности и продуктивности труда;
низкая трудовая и производственная дисциплина;
низкая управляемость и предсказуемость процесса программирования;
очень высокая себестоимость продукции;
высокий процент брака, причем статистические методы выборочного контроля качества продукции не применимы;
высокая консервативность персонала и связанные с этим трудности усовершенствования существующей технологии.

Профессионального программиста, который на постоянной основе вовлечен в производственный процесс программирования, причем этот процесс обеспечивает программисту средства существования, уместно назвать *умелым программистом*. Дисциплина программирования ориентирована на умелых программистов. Важно подчеркнуть, что для умелого программиста первичным является вопрос продуктивности его ремесла. Вопросы науки и искусства, если и присутствуют, то носят вторичный характер.

Замечание

> Умелыми программистами не рождаются, им становятся в результате накопления опыта.[41]

5.2. Парадигмы программирования

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.

5.2.1. Структурное программирование

Исторически первой оформленной парадигмой принято считать так называемое *структурное программирование*. Выявленная[42] 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**".

---

[41] Это обстоятельство объясняет стойкий стереотип принятия решений управляющих кадрами программистов, которые судят об умелости программиста по его опыту, который зачастую измеряется просто как непрерывный стаж работы по специальности.

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

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.1).

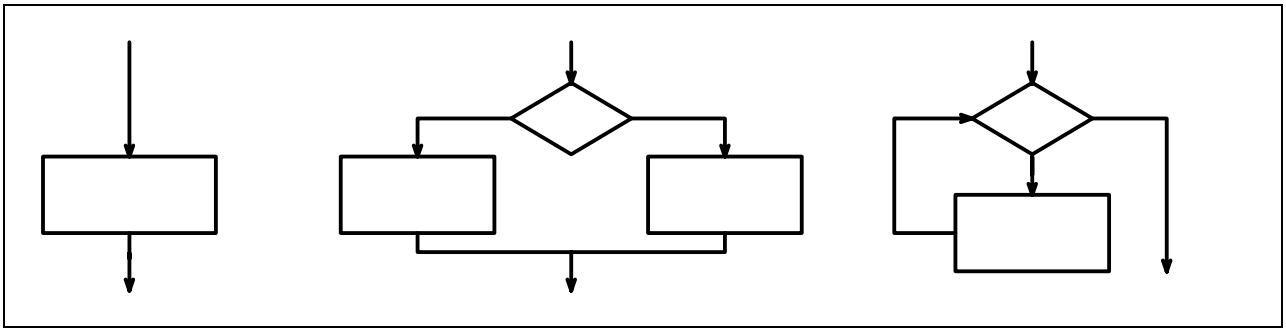


Рис. 36. Базовые структуры управления программирования без Go To

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) первое противоречие – между статикой и динамикой.

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.[43]

Замечание

> 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.[44]

Наличие иерархической структуры вложенности позволяет (до некоторой степени) разрешить и второе противоречие. А именно, последовательное развертывание структуры вложенности обеспечивает деление задачи на подзадачи и, таким образом, на каждом шаге ограничивает комбинаторную сложность. Это называется программированием *методом пошагового уточнения*.[45]

Все системы программирования (кроме уже забытых самых ранних) в той или иной форме поддерживают понятие *модуля*. В разных системах программирования модули называются и определяются по разному: подпрограмма, функция, класс, модуль, компонента и т.д. Суть дела, однако, одна и та же: модульность является средством преодоления количественных ограничений. Как правило, модульная структура допускает вложение модулей или ссылки между модулями (или то и другое), поэтому каждый модуль в отдельности может

[43] Как показано в старой работе И.Р. Агамирзяна и А.С. Иванова, необходимо и достаточно иметь процедуры с процедурными параметрами.

[44] Это наглядно продемонстрировано в известной статье Д. Кнута Structured programming with **go to**.

[45] Будучи поддержанным инструментальным средством, например, простым макрогенератором, пошаговое уточнение позволяет иметь самодокументированные и легко модифицируемые программы.

быть сделан обозримым для программиста.[46] Таким образом, *модульное программирование* является непосредственной реализацией принципа "разделяй и властвуй" в программировании.

Замечание

Кроме непосредственной структуризации текста программы на модули в системах программирования навешивается еще множество дополнительных функций: сокращение объема кода, инкрементальная компиляция, ограничение областей действия, раннее (статическое) и позднее (динамическое) связывание, инкапсуляция данных и пр.

Программирование сверху вниз – это обобщающее название для модульного программирования без **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 подзадача не станет столь простой, что ее реализация становится очевидной.

Парным к программированию сверху вниз является *программирование снизу вверх*,[47] 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 (рис. 37).

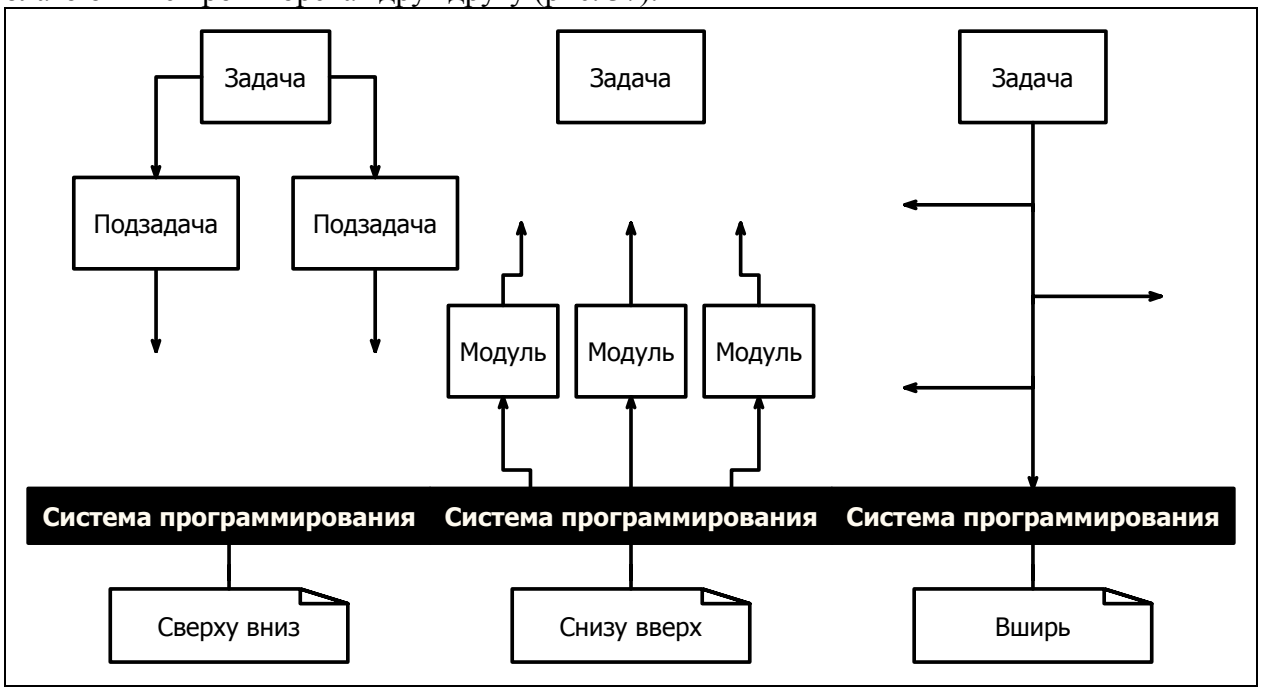


Рис. 37. Программирование сверху вниз, снизу вверх и вширь

Замечание

С.С. Лаврову принадлежит изобретение термина *программирование вширь*, когда, начиная с самого первого шага, создается и на всех последующих шагах поддерживается работоспо-

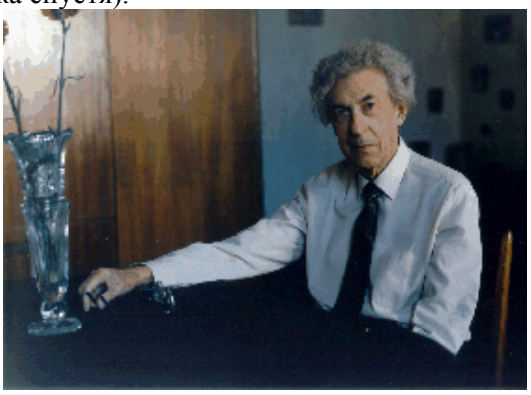
[46] Э. Дейкстра в образной форме формулировал это так: "в хорошо структурированной программе текст каждого модуля занимает ровно одну страницу".

[47] По-английски это называется bottom-up approach.

собная версия будущей программы.[48] Конечно, в начале работы эта программа не умеет выполнять 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,[49] 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. Сходные идеи пропагандирует Кент Бек в своем экстремальном программировании (но четверть века спустя).

Лавров Святослав Сергеевич (1923–2004) – российский ученый, член-корреспондент РАН (1991; член-корреспондент АН СССР с 1966). Труды по механике, автоматическому управлению, вычислительной математике. Ленинская премия (1957).

Научная биография С. С. Лаврова в определенном смысле уникальна. Совсем в молодом возрасте С. С. Лавров стал основоположником ракетно-космической баллистики в СССР и неоспоримым авторитетом в области динамики управляемого полета и автоматического управления. Появление цифровой вычислительной техники привело к резкому повороту в деятельности С. С. Лаврова и в течение нескольких лет сделало его классиком программирования в СССР. Он разработал первый отечественный транслятор Алгола и программное обеспечение первых полетов человека в космос.



Лавров С.С. Программирование. Математические основы, средства, теория. – СПб., БХВ-Петербург, 2001

Лавров С. С. Лекции по теории программирования. – СПб, НЕСТОР, 1999

Важно подчеркнуть, что структурность программы привносится программистом, а не предопределяется системой программирования. В любой системе программирования любую программу можно написать структурно (т.е. хорошо) и не структурно (т.е. плохо), как с использованием оговоренных операторов (например, **go to**) и инструментальных средств (например, пошагового уточнения), так и избегая их.

5.2.2. Логическое программирование

Парадигмы программирования до некоторой степени инспирированы моделями вычислимости.

Модель вычислимости — это формальное определение понятия вычислимой функции, или алгоритма.

Одной из классических моделей вычислимости является так называемая машина Тьюринга, которая состоит из памяти и программы (процессора). Память (потенциально бесконечная) разделена на ячейки, в которых могут быть записаны символы конечного алфавита. Машина работает дискретно, и всегда находится в одном из состояний, которых конечное число. За один шаг машина выполняет одну команду, которая определяется текущим состоянием и символом в текущей ячейке. Выполнение команды состоит в записи

[48] 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.

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

нового символа в текущую ячейку, перемещении к другой текущей ячейке и изменении состояния. Эта простая модель вычислимости отражена в наиболее распространенной архитектуре компьютеров, которая называется архитектурой фон Неймана. Компьютер имеет адресуемую память, состоящую из ячеек, и процессор, дискретно выполняющий команды. Команды оперируют с ячейками памяти (как правило, одна команда оперирует с одной ячейкой). Из соображений эффективности выполнения программ средства программирования ориентируются на подразумеваемую архитектуру. Во всех распространенных системах программирования имеются два фундаментальных понятия, индуцированные архитектурой машины — 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.[50] Программирование, в котором одно или оба понятия не используется, называется *непроцедурным*.

Замечание

> Важно отдавать себе отчет, что привычное процедурное программирование (и архитектура фон Неймана) обусловлены историческими причинами, но не более того, т.е. не являются законами природы.

Существуют и появляются новые разнообразные парадигмы непроцедурного программирования, основанные на иных моделях вычислимости,[51] нежели машина Тьюринга и ориентированные на иную архитектуру компьютера, нежели классическая архитектура компьютера фон Неймана.

Собственно *логическое программирование* базируется на следующем фундаментальном факте: из конструктивного доказательства теоремы существования $\forall x(P(x)\rightarrow\exists y Q(x,y))$ может быть эффективно извлечена (аннотированная) программа $\{P(x)\}y:=f(x)\{Q(x,y)\}$. Наибольшее распространение получала реализация, известная под названием язык Пролог, которая основана на применении метода резолюций с фиксированной стратегией "входная отрицательная" для автоматического доказательства теорем в хорновском подклассе исчисления предикатов первого порядка. В логической программе нет явного управления (оно предопределено стратегией поиска вывода), а присваивание неявное и сводится к запоминанию унифицирующих подстановок.

Замечание

> Программирование, в котором используется явное понятие управления, называется *императивным*, в противоположность *декларативному* программированию. Логическое программирование является декларативным.

Продукционное программирование тесно связано с нормальными алгоритмами Маркова. Продукционная программа состоит из набора продукций (правил) вида **if** P(D) **then** D:=f(D), которые применяются к глобальной базе данных D, пока не будет удовлетворено условие окончания. Продукционные программы обладают предельной модульностью, являются чрезвычайно легко модифицируемыми и часто применяются для программирования экспертных систем. Управление неявное (определяется стратегией), а переменные всегда глобальные (база данных D).

Функциональное программирование тесно связано с λ-исчислением Чёрча. Функциональная программа является композицией функционалов (т.е. функций, аргументами и результатами которых могут быть функционалы). В функциональной программе нет ни явного управления, ни явных переменных, а следовательно, нет и явного присваивания.

---

[50] Характерным признаком процедурного программирования является наличие явного оператора присваивания.

[51] Все известные модели вычислимости оказались эквивалентными, т.е. сводимыми друг к другу. Это обстоятельство является основным аргументом в пользу **тезиса Чёрча**, утверждающего, что формально определенное понятие алгоритма равнообъемно неформальному понятию алгоритма.

Замечание

Практические непроцедурные системы программирования, ориентированные на эти парадигмы, отнюдь не требуют от программиста владения изощренным математическим аппаратом. Наоборот, зачастую они проще, понятнее и удобнее в использовании, чем привычные процедурные системы программирования. Поскольку непроцедурные программы в меньшей степени ориентированы на классическую архитектуру компьютера, реализация таких программ оказывается несколько менее эффективной на обычных компьютерах. Но существуют (в промышленных масштабах) и все шире применяются компьютеры нетрадиционной архитектуры, для которых непроцедурное программирование оказывается более естественным, продуктивным и эффективным.

В силу целого ряда объективных и субъективных (в основном, исторических) причин перечисленные непроцедурные парадигмы программирования сравнительно редко применяются при решении интересующих нас задач разработки вертикальных приложений типа информационных систем масштаба предприятия, поэтому в данном изложении дисциплины программирования рассматривается исключительно процедурное программирование.

5.2.3. Объектно-ориентированное программирование

Парадигма *объектно-ориентированного программирования* (далее ОО программирование) — это наиболее популярная в данный момент парадигма, которая является консервативным расширением упомянутых выше подходов.

Программирование без **go to** (вкупе с необходимыми расширениями типа структурных переходов, обработки исключительных ситуаций и модульности) удовлетворительным образом решает проблему структурирования управления. Однако, как известно,[52] 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.[53] 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 ОО программирования является *инкапсуляция*, т.е. структурирование программы на модули особого вида, объединяющего данные и процедуры их обработки, причем внутренние данные модуля не могут быть обработаны иначе, кроме как предусмотренными для этого процедурами. Каждый такой модуль 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).[54] В разных вариациях ОО программирования эту структуру называли по-разному: класс, абстрактный тип данных, модуль, кластер и др. В настоящее время наиболее популярно слово *объект*. Кроме основной идеи инкапсуляции, с ОО принято ассоциировать также понятия *наследование* и *полиморфизм*.

---

[52] См. Н.Вирт. Программы = Алгоритмы + Структуры данных.

[53] Отказ от этого привычного предрассудка приводит к функциональному программированию

[54] На самом деле достаточно методов, свойства являются частным случаем, который предусмотрен для удобства.

Замечание

Интересно и поучительно проследить историю развития ОО 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), получили общепризнанное языковое выражение к 1988 году (C++) и стали естественным инструментом для профессиональных программистов к 1998 году.

Важным элементом парадигмы ОО программирования является следующая идея: *класс* (который тоже является объектом) может иметь методы, называемые *конструкторы* и *деструкторы*, позволяющие в время выполнения программы динамически порождать и уничтожать *экземпляры* (объекты) данного класса. Объекты одного класса сходны между собой (например, наследуют методы класса), но имеют различия (например, имеют разные *значения* свойств).

Замечание

Механизм динамического порождения экземпляров оказывается очень полезен, а в некоторых случаях незаменим, однако это не всегда так. Более того, очень часто в приложениях типа информационной системы масштаба предприятия оказывается, что экземпляры объектов если и порождаются, то как пассивные хранилища конкретных наборов значений свойств, а их методы статически предопределены и неизменны (и, следовательно, достаточно иметь методы в единственном экземпляре). В таком случае, экземпляры объектов (наборы значений свойств) удобно хранить в виде записей таблицы базы данных, а их общие методы — в виде специального модуля, содержащего процедуры работы с этой базой. Такой модуль обычно называется *служба* (service).[55] Программирование в терминах служб является ОО программированием, потому что главным в инкапсуляции является не формальное объединение данных и процедур их обработки в какой-то специальной программной конструкции, а регламентация доступа к данным (см. также раздел *Службы*).

Из сказанного следует важный вывод:[56] 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 ОО программы, используя все преимущества ОО программирования, равно как и в самой современной ОО системе никто не застрахован от безнадежно недисциплинированного манипулирования объектами. Более того, в некоторых случаях слишком автоматизированные средства ОО программирования *мешают* ОО программированию, поскольку *навязывают* программисту конкретные решения. Например, в большинстве современных систем ОО программирования требуется, чтобы метод был отнесен в точности к *одному* классу, хотя это отнюдь не всегда естественно и удобно.[57]

С парадигмой ОО программирования связано развитие и практическое воплощение важной мысли, (уже упомянутой в разделе *Структурное программирование*) о неразрывной связи проектирования и кодирования и о примате проектирования. Человек мыслит реальный мир состоящим из объектов.[58] Кодированию *всегда*[59] предшествует проектирование, важнейшей частью которого является моделирование, т.е. вычленение в предметной области задачи объектов и связей, которые существенны для решения задачи. ОО программирование позволяет установить прямое соответствие между программными конструкциями и объектами реального мира через объекты модели. Таким образом, ОО система программирования поддерживает не только кодирование, но и (до некоторой степени) проектирование. При написания ОО программы основные усилия программист тратит на определение состава классов, их методов и свойств. Эта информация является в чистом

[55] Впервые этот термин был введен И.В. Романовским в статье "Программирование в терминах служб" (1974). Часто используемый сейчас эквивалентный термин "сервис", полученный прямой транслитерацией, гораздо менее удачен с точки зрения чистоты русского языка.

[56] См. последний абзац в разделе *Структурное программирование*.

[57] Утрируя, можно сказать, что самым ОО языком является Фортран, поскольку он не навязывает никаких ОО решений.

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

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

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 ОО программе оказывается почти тривиальным.

Замечание

> В непрерывном и текучем реальном мире никаких объектов, конечно, нет. Объекты вычленяются субъективно, по произволу программиста. Это вычленение отнюдь не однозначно и может быть сделано более или (чаще) менее удачно. Программисты, умеющие удачно выделять объекты и связи, называются *системными аналитиками*. Во многих предметных областях, с целью снижения отрицательных последствий неудачного анализа, имеются готовые модельные решения, в которых аккумулирован удачный опыт. Например, для информационных систем предприятий, которые прежде всего имеются в виду, очень часто вместо объектов реального мира рассматриваются их модели, которые называются *документами*. Документы более формальны и абстрактны, чем те явления, которые они моделируют, и поэтому программировать информационные системы, которые начинаются и заканчиваются документами, гораздо проще.

5.3. Циклы повышения продуктивности

Дисциплина программирования нацелена на решение следующей экстремальной задачи: максимизировать среднюю по организации продуктивность программирования при выполнении ограничений снизу на качество, которые задает заказчик. Продуктивность программирования измеряется как чистый доход от выполненного программного проекта, деленный на чистое затраченное время, т.е. имеет размерность деньги/(время*люди). Методика измерения чистого затраченного времени определяется документированной процедурой *Рекомендации по учету рабочего времени*. Качество программы считается удовлетворительным, если таковым его признает заказчик (о чем заказчика надлежит явным образом спросить, см. раздел *Порядок проведения типового проекта*).

Дисциплина программирования представляет собой набор эмпирических правил и рекомендаций для умелых программистов. В этой связи было бы преувеличением утверждать, что Дисциплина программирования представляет собой исчерпывающий список наилучших приемов, которые нужно обязательно применять. Это скорее представительный список советов, как избежать нежелательного применения наихудших приемов.

5.3.1. Продуктивность программирования

Повышение продуктивности программирования возможно за счет двух основных факторов:

сокращение количества внеплановых изменений кода,

увеличение объема повторно использованного кода.

Величина положительного воздействия первого фактора на продуктивность программирования определяется информационным потоком в цикле, проходящем через заказчика (см. рис. 38). Действительно, внеплановые изменения кода — это изменения, производимые для исправления выявленных ошибок. Существуют различные классификации типов ошибок, в своем большинстве они довольно близки; здесь используется следующая классификация:

синтаксические ошибки — нет исполнимой программы;

ошибки кодирования — имеется исполнимая программа, некоторые протоколы выполнения которой не удовлетворяет спецификации;

ошибки проектирования — имеется исполнимая программа, выполнение которой формально не противоречит спецификации, но результаты выполнения фактически не удовлетворяют заказчика.

Синтаксические ошибки здесь не рассматриваются, поскольку, во-первых, в современных системах программирования они исправляются очень легко и быстро, а, во-вторых, уме-

лые программисты просто не допускают таких ошибок. Доля синтаксических ошибок в снижении продуктивности пренебрежимо мала.

Ошибки кодирования составляют только малую долю в снижении продуктивности. При использовании современных средств формальной спецификации, автоматизированных средств отладки и, самое главное, следовании эмпирическим правилам надежного программирования умелые программисты практически не допускают ошибок кодирования.

Самыми дорогими, т.е. снижающими продуктивность в наибольшей степени, и, к сожалению, чаще всего встречающимися, являются ошибки проектирования. Ошибки проектирования обусловлены неточностью, неполнотой или неадекватностью спецификаций и моделей, построенных по спецификации. Выявление и фиксация этих ошибок происходят через заказчика (или его заменителя в лице независимого тестировщика). Чем лучше организовано взаимодействие с заказчиком, тем более продуктивным оказывается программирование.

Замечание

> Кроме внеплановых изменений, на продуктивность влияют плановые изменения, которые происходят на втором витке при переходе от прототипа к полнофункциональному приложению (см. раздел *Модель процесса*). Как показывает практика, двух витковая разработка с прототипом, хотя и несколько увеличивает затраты за плановые модификации, но зато резко снижает затраты на внеплановые модификации и обеспечивает суммарное увеличение продуктивности. Полное повторное проектирование и кодирование оказывается более выгодным, чем латание кода с помощью "заплат".

Величина положительного воздействия второго фактора (повторное использование кода) на продуктивность программирования определяется информационным потоком в цикле, проходящем через программиста (см. рис. 38). Хотя выгоды повторного использования очевидны и хорошо осознаются как теоретиками, так и практиками программирования, эта проблема фактически далека от своего разрешения. Реально повторно используются два вида компонент: стандартные библиотеки, созданные фирмами, производящими системы программирования, и абсолютно неформальный "опыт", накапливающийся в головах у разработчиков. Стандартные библиотеки хороши, но по очевидным причинам оказываются слишком универсальными, перегруженными и требуют дополнительных усилий от программиста при использовании в конкретном проекте. Фактически, это просто средство обогащения системы программирования. Индивидуальный опыт ценен, но он накапливается слишком медленно и дорого, а исчезает (в случае увольнения) мгновенно и без следа. Самым ценным и самым трудным является накопление и повторное использование компонент корпоративного уровня, а не индивидуальных и не всеобщих.

Замечание

> В настоящее время используются различные средства программной поддержки процесса накопления и повторного использования корпоративных программных решений, которые обычно называют репозиториями (repository). Этим средствам принадлежит будущее, но пока что реальный эффект невелик. Причина состоит не в качестве программной реализации репозитория, а в методологии его использования. Императивный код современных систем программирования, даже объектно-ориентированных, с трудом поддается повторному использованию. Довольно трудно вычленить из вертикального приложения компонент, например, библиотеку объектов, который бы допускал повторное использование в другом проекте без дополнительной настройки. Настройка же зачастую сводится к перелицовке кода и "пришиванию заплат", что непродуктивно. Это ситуацию можно пояснить следующим сравнением. В хорошем научном учреждении есть научная библиотека и архив диссертаций. Библиотека имеет высокую степень повторного использования знаний за счет наличия библиотекаря, который обеспечивает комплектование, индексирование, реферирование, составление указателей и просто может дать хороший совет. Архив диссертаций имеет почти нулевую востребованность, хотя в нем, может быть, хранится не менее ценная информация. Пока что имеющиеся репозитории больше напоминает архивы, чем библиотеки.

5.3.2. Спецификации и модели

Центральной идеей дисциплины программирования является выбор уровня *формализации* информации, циркулирующий в циклах повышения продуктивности (см. рис. 38), проходящих через программистов и заказчиков. Если информация, циркулирующая в левом цикле, представлена совершенно формально, например, в виде программного кода, то она оказывается недостаточно гибкой (т.е. недостаточно легко модифицируемой) для повторного использования при проектировании. Если информация, циркулирующая в правом цикле, представлена совершенно неформально, например, в виде текстов (устных или письменных) на естественном языке, то она оказывается слишком гибкой (т.е. неоднозначной и неопределенной) для использования при анализе и последующем проектировании. Опыт и принцип экономии мышления подсказывают, что в обоих случаях следует использовать один и тот же уровень формализма. Более того, в обоих циклах можно и нужно использовать одно и то же средство представления информации, каковым на сегодняшний день может и должен служить в унифицированный язык моделирования UML (Unified Modeling Language).

В циклах повышения продуктивности можно выделить типы информации и направление ее движения, как показано на рис. 38:[60]

Пользовательские требования (или внешние спецификации) — от заказчиков на фазы анализа и проектирования;[61]

Пользовательская документация (или детальные спецификации) — от фаз стабилизации и внедрения к заказчику;[62]

Реализованный компонент со спецификацией (важно наличие кода) — от фазы реализации к программистам;[63]

Специфицированный компонент (важно наличие спецификации)[64] — от программистов (или из репозитория) к фазе проектирования.

Во всех случаях информация имеет одну и ту же *семантику* — это некоторая спецификация (т.е. описание) программы или компонента программы, но разную *прагматику* (т.е. предназначение). Очевидно, что если все виды спецификаций будут вдобавок выражены в одном и том же *синтаксисе*, причем понятном как человеку, так и компьютеру, то можно надеяться на интенсификацию информационных потоков и увеличение продуктивности программирования.

---

[60] Рис. 38 немного уточняет рис. 4.

[61] Тот факт, что на практике внешние спецификации зачастую готовятся силами сотрудников , не меняет сути дела: при этом сотрудники просто *играют роль* заказчика, выполняя его работу.

[62] На рис. 4 и 38 отсутствуют фазы опытной эксплуатации и внедрения. Это сделано только для наглядности рисунка. На самом деле эти фазы подразумеваются.

[63] Или в репозиторий.

[64] При проектировании вполне можно использовать еще не реализованные компоненты, лишь бы они были хорошо специфицированы!

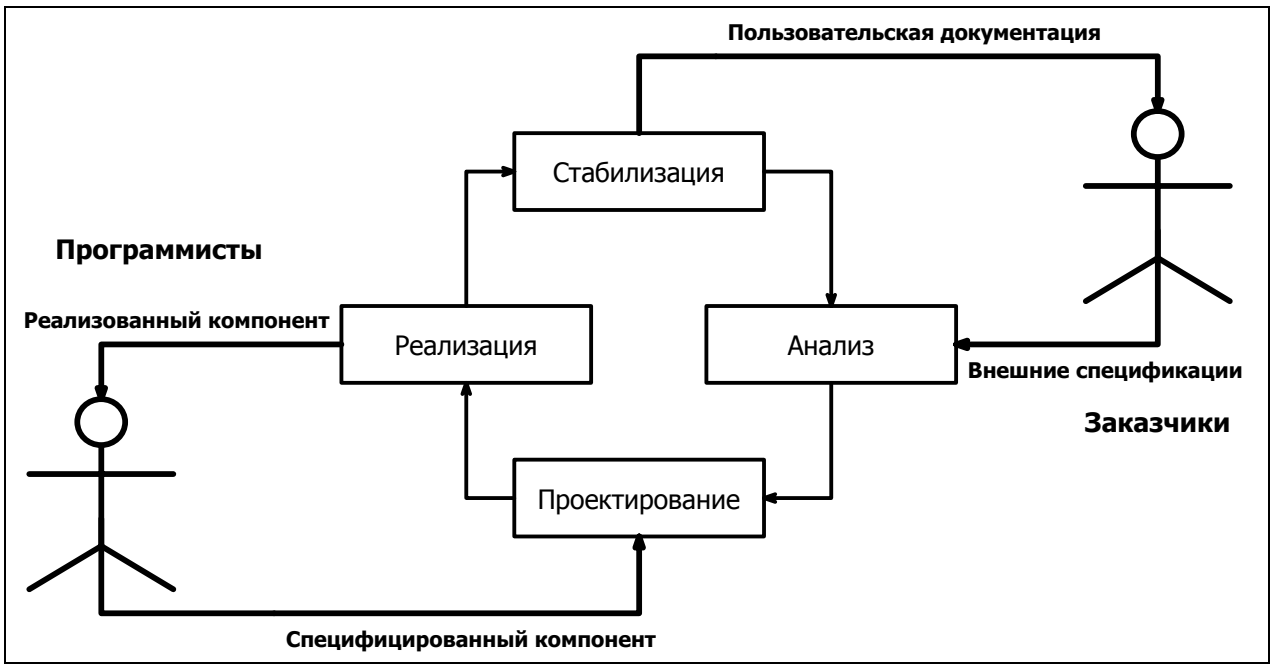


Рис. 38. Движение спецификаций в циклах повышения продуктивности

С прагматической точки зрения для разных видов спецификаций на первый план выдвигаются разные требования:

- при спецификации реализованного компонента наиболее важными являются *точность и полнота* спецификаций, а наилучшей в этом смысле спецификацией программы является сам текст программы;
- при описании программы для пользователя важнее всего *понятность*, и поэтому в пользовательской документации традиционно используется естественный (причем родной) язык;[65]
- при повторном использовании компонентов самым важным является спецификация *интерфейсов*, поэтому в этом случае активно используются различные объектные модели;[66]
- при составлении внешних спецификаций первостепенной является *наглядность*, потому что для заказчик может надежно верифицировать только *обозримые* спецификации.

В качестве средства спецификации, которое одновременно удовлетворяет (в достаточной степени) этим противоречивым требованиям наиболее целесообразно использовать *модель программы*, выраженную в графической нотации UML.[67] Диаграммы, которые составляют модель UML, удовлетворяют поставленным требованиям в следующем смысле.

Будучи строгим и формальным, хотя и графическим языком, UML обладают достаточной точностью для (полу) автоматической генерации кода.

Замечание

> Не следует рассматривать UML в качестве альтернативы обычным средствам программирования. Генерация кода — это хотя и полезный, но побочный эффект основной функции моделирования.

Диаграммы UML достаточно понятны, чтобы служить средством коллективного моделирования и обмена мнениями с заказчиками.

[65] Пользователи не будут читать код, будь он хоть трижды самодокументированный!
[66] Если объектная модель представлена в графической форме, пользователь может согласиться ее посмотреть.
[67] В некоторых случаях целесообразно дополнительно к UML использовать и другие средства.

Замечание

> Традиционно считается, что самым понятным является текст с иллюстрациями. Но иллюстрации с текстом еще понятнее, т.к. короче — из текста удалена связующая "вода".

Диаграммы UML хорошо подходят для спецификации интерфейсов.

Замечание

> Диаграммы UML гораздо удобнее для спецификации интерфейса не иерархической системы объектов (т.е. службы) и не связаны ограничениями конкретной системы ОО программирования.

Графическая форма моделей UML является наиболее наглядной из известных.

Замечание

> Конечно, использование UML не избавляет от риска составить совершенно ненаглядную, перегруженную и непонятную модель. Но это уже проблема субъекта моделирования, а не средства моделирования.

Наконец, визуальное моделирование *естественно* (после преодоления невысокого входного порога непривычности). Программе всегда предшествует модель, потому что программа не может возникнуть ниоткуда. Модель всегда имеет место, даже если беспринципный программист открещивается от нее. Начиная с некоторой пороговой величины программы (для всех проектов современных программирующих организаций это порог явно превзойден) дополнительные трудозатраты на визуальное моделирование в каждом отдельном проекте окупаются суммарным увеличением продуктивности для всех проектов в целом.

5.4. Программная архитектура

Архитектура приложения — это набор принципов, используемых при принятии решений в процессе разработки и оценки приложения.

Замечание

> Набор принципов может отсутствовать. Такое программирование называется беспринципным и здесь не рассматривается.

На архитектуру приложения (или программного решения иного типа) оказывают влияние множество факторов:
тип приложения (информационная система, система управления, встроенная система и т.п.);
характер использования (одноразовая, индивидуальная, групповая, массовая программа);
дополнительные требования (надежность, секретность и пр.);
предметная область (бизнес, наука, транспорт и т.п.);
субъективные предпочтения пользователей и разработчиков и многое другое.
Сколько существует возможных комбинаций значений перечисленных (и не перечисленных) факторов, столько можно усмотреть различных (и в некотором смысле наилучших) архитектур. Таким образом, единственной рекомендуемой архитектуры (набора руководящих принципов) нет и быть не может.
Однако вертикальные приложения типа информационная система масштаба предприятия *могут иметь* сходную архитектуру. В сущности, типовая архитектура (или модель) приложения, которая предлагается, например, в MSF, является обобщением некоторых типовых программных решений, которые на практике доказали свою жизнеспособность будучи применены в большом числе реальных приложений для бизнеса. Сама возможность построения архитектуры приложения без привязки к предметной области основана на вычленении некоторых типовых черт, присущих приложениям рассматриваемого класса. Рассматриваемые приложения имеют следующие характерные особенности:

- **интерфейс с пользователем**, т.е. в интерфейсе приложения подразумевается наличие пользователя, интерфейс с другими программами или с аппаратурой либо отсутствует, либо носит вспомогательный характер;

- **многофункциональность**, т.е. наличие нескольких, зачастую разнородных функций, которые выполняются в разных комбинациях и при разных условиях (пример приложения, которое не обладает такой особенностью — транслятор с языка программирования);
- **специализация**, т.е. ориентированность на конкретную предметную область (вид бизнеса) с учетом особенностей самого бизнеса и даже конкретного предприятия;
- **рутинность**, т.е. принципиальная заведомая разрешимость и невысокая наукоемкость решаемых задач;
- **много пользователей**, т.е. наличие нескольких категорий пользователей приложения, которые решают различные задачи и, следовательно, обладают различными правами по отношению к приложению;[68]
- **модифицируемость**, т.е. постоянное изменение функций в процессе эксплуатации в соответствие с изменением бизнеса и потребностей пользователей.

В следующих разделах рассматриваются некоторые архитектурные решения, характерные и рекомендуемые для рассматриваемого класса приложений.

5.4.1. Событийное управление

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 (т.н. пассивный интерфейс).
В настоящее время более распространенным, особенно в приложениях рассматриваемого класса, является второе решение (так называемый *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[69] (event-driven), который описан ниже;
устоявшейся за последние несколько лет привычкой массового пользователя к пассивному интерфейсу.

Замечание

> 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. 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 не-

[68] Одно время для характеристики категории пользователей и соответствующего набора функций и прав употреблялся термин АРМ — Автоматизированное Рабочее Место.
[69] 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.

явным 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 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.

Замечание

> В сущности, событийное управление является развитием очень старой идеи *прерываний*, которые были придуманы для организации взаимодействия синхронно и монотонно работающего центрального процессора с асинхронными и немонотонными внешними устройствами. Поэтому событийное управление называют еще *асинхронным*.

В современных системах программирования имеются богатые и все время развивающиеся библиотеки готовых компонент, которые называются *элементы управления* (controls), и которые тесно интегрированы со встроенными механизмами событийного управления. Использование готовых элементов управления удобно, продуктивно и должно быть рекомендовано в большинстве случаев.

Замечание

> У неумелых программистов существует иллюзия, что программировать вертикальные приложения с пассивным интерфейсом очень просто: достаточно выслушать (даже не записав!) перечень команд приложения, в том виде, как их представляет себе заказчик (пользователь), "натаскать" готовых элементов управления в программу, определить некоторые реакции и приложение готово. Это именно иллюзия, потому что хотя описанный способ действий действительно прост и хорошо поддержан системой программирования, он непродуктивен: в недопустимо большом числе случаев заказчик будет не удовлетворен результатом.

Предположение, что пользователь приложения для бизнеса с пассивным интерфейсом имеет в голове *правильную* (с точки зрения приложения) программу выполнения функции бизнеса является очень сильным и далеко не всегда выполняется. Например, похвальное и естественное стремление программиста сделать набор элементарных команд приложения компактным и ортогональным приводит к тому, что часто последовательность команд, выполняющих некоторую функцию бизнеса, обязана быть транзакцией (для сохранения целостности корпоративных данных). В лучшем случае проверка выполнения всех условий целостности предусматривается умелым программистом в процедуре реакции последнего элемента управления, при этом пользователь просто встает в тупик, когда получает отказ программы выполнять операцию при нажатии самой невинной кнопки ОК. В худшем случае программист не проверяет выполнение условий целостности в расчете на то, что пользователь будет действовать только допустимым образом. В результате получается ненадежное приложение, в котором *можно* неумышленно нарушить целостность корпоративных данных.[70]

[70] Снобистски настроенные программисты иногда называют такого рода проверки в программах обидным термином "защита от дурака". Вопрос о том, кто же в данном случае является "дураком" — программист или пользователь — очень спорен.

Существуют три основных метода решения этой проблемы.
Полный отказ от пассивного интерфейса и переход на активный интерфейс, который держит пользователя в рамках преопределенного сценария и не позволяет недопустимых действий. Такой подход был характерен для старых приложений на больших машинах коллективного доступа, но применяется и сейчас, например, во встроенных системах, в критически важных (critical mission) приложениях или в приложениях с малым числом жестко специфицированных функций типа систем резервирования авиабилетов.
Включение в приложение с пассивным интерфейсом большого количества проверок целостности, контекстной справки с подсказками и предупреждениями о потенциальной опасности или недопустимости выполняемых действий, интеллектуальных средств анализа, прогнозирования и коррекции действий пользователя. Такой подход очень хорош, но дорог, поэтому он применяется, в основном, только в массовых горизонтальных приложениях.
Реализация в рамках в целом пассивного интерфейса активных сценариев для особо сложных, важных или опасных функций. Наиболее показательным примером являются *мастера*, знакомые всем по приложениям Microsoft Office. Этот подход дешев в реализации и обеспечивает разумный компромисс между ортогональностью базовых команд, надежностью приложения и удобствами пользователя. Можно рекомендовать использование мастеров в большинстве вертикальных приложений для бизнеса.

Мастер — это активный сценарий (как правило, линейный), выраженный средствами пассивного интерфейса.

Типичный мастер реализуется как предопределенная последовательность диалоговых окон, в которых пользователь шаг за шагом устанавливает значения параметров одной сложной транзакции. Возможности современных элементов управления позволяют легко реализовать линейного или линейно ветвящегося мастера над набором базисных команд приложения. Разумно подобранные значения параметров по умолчанию позволяют резко ускорить выполнение типовых процедур. Мастера являются превосходной альтернативой чисто событийному управлению при реализации процедур бизнеса. Выделение сценариев, подлежащих реализации в виде мастера, является одной из основных задач концептуального проектирования (см. ниже). Явными кандидатами на реализацию в виде мастера являются элементы диаграммы использования (use case) верхнего уровня в модели приложения.

Замечание

> Microsoft Office содержит, по видимому, одну из самых богатых коллекций мастеров. Для освоения этой техники настоятельно рекомендуется знакомство с мастерами Microsoft Office, причем как с поистине гениальными Мастером диаграмм и Мастером сводных таблиц в Excel, так и с другими, довольно убогими, которые нет нужды называть здесь.

Важно подчеркнуть, что решения об архитектуре интерфейса следует принимать с оглядкой на пользователей, а не идя на поводу у системы программирования и тем паче не по произволу программиста.

5.4.2. Архитектура клиент/сервер

Термин *архитектура клиент/сервер* стал активно использоваться по мере распространения локальных сетей и распределенных многопользовательских приложений, поэтому часто ассоциируется с наличием выделенного в сети компьютера (сервера) или с программой, выполняющейся на этом компьютере. Вообще говоря, архитектура клиент/сервер подразумевает не более чем определенный способ организации взаимодействия компонент многокомпонентной программы. А именно, программа содержит некоторую компоненту, называемую сервером, и одну или несколько других компонентов, называемых клиентами. Клиент имеет возможность асинхронно для сервера инициировать выполнение процедур сервера и получать результаты выполнения. Как правило, механизм

асинхронного вызова обеспечивает возможность нескольким клиентам взаимодействовать с сервером параллельно или квазипараллельно. Возможны два основных способа реализации архитектуры клиент/сервер:
пассивный сервер: сервер управляется событиями (т.н. *запросами* к серверу), которые инициируют клиенты;
активный сервер: сервер опрашивает клиентов на предмет наличия запроса.
В случае реализации на одном компьютере с фиксированным числом клиентов архитектура клиент/сервер сводится к событийному управлению. Интерес представляет случай, когда серверы и клиенты выполняются на физически различных компьютерах, реально параллельно и число клиентов (и, может быть, серверов) динамически меняется. В настоящее время имеются разнообразные готовые компоненты, которые могут быть с успехом использованы при реализации серверной части, подобно тому, как готовые элементы управления могут быть использованы при реализации клиентской части. Типичной задачей, на которую хорошо проецируется архитектура клиент/сервер, является задача управления централизованными корпоративными данными с нескольких рабочих мест. Если разрабатываемое вертикальное приложение предусматривает такую задачу, то настоятельно рекомендуется использование архитектуры клиент/сервер и готовых компонент. Для отказа от использования архитектуры клиент/сервер должны быть очень веские основания, обусловленные спецификой приложения.

Замечание

> Не следует думать, что для реализации приложения в архитектуре клиент/сервер обязательным является использование дорогой готовой СУБД со словом "сервер" в названии. Поскольку все, что нужно от сервера, это поддержка событийного управления для программно инициируемых удаленных событий, оказывается, что очень многие современные дешевые программы, поддерживающие указанные механизмы, достаточны для реализации приложений в архитектуре клиент/сервер. Если требования заказчика по производительности, безопасности и количеству клиентов не очень высоки, то использование более простых серверов может обеспечить более дешевое, а значит более продуктивное решение.

5.4.3. Службы

Служба (service) — это программная конструкция, обеспечивающая набор выполнение набора операций (или *услуг*) с некоторыми объектами (или одним объектом).

Важно, что служба полностью специфицируется своим интерфейсом (сигнатурой услуг), а внутренняя реализация службы скрыта от внешних клиентов службы, которые пользуются услугами. В такой формулировке служба является очень широким понятием, включающем в себя, как частные случаи, основные методы ОО (и не только ОО) программирования. Программная реализация служб может быть существенно различной и зависеть от используемой системы программирования и модели приложения, и, что важнее всего, от характеристик службы, как компоненты ОО программы. Можно принимать во внимание различные объектные характеристики службы и, соответственно, использовать различные методы реализации. Например, существенным образом влияют на выбор метода реализации такие характеристики:
Постоянство объектов: имеет служба внутреннюю память или нет, т.е. должно ли состояние объектов сохраняться между предоставлением услуг и если должно, то какое время. Возможные значения этой характеристики:
Служба без памяти, т.е. служба не управляет временем жизни объектов, с которыми оперирует (например, машинная арифметика).
Оперативная служба, т.е. служба, объекты которой существуют во время выполнения приложения, использующего услуги службы (например, соединение ODBC).

Постоянная служба, т.е. служба, объекты которой существуют и сохраняют свое состояние независимо от выполнения приложения, использующего услуги службы (например, файловая система).
Множественность экземпляров: обслуживает служба фиксированное или переменное количество объектов, т.е. порождаются ли экземпляры объектов динамически. Возможные значения этой характеристики:
Служба обслуживает единственный (безымянный) экземпляр объекта. Например, так устроено большинство служб операционных систем.
Служба обслуживает статически определенный набор объектов (как, правило, именованных).
Служба позволяет обслуживать заранее не известное множество объектов (как правило, в состав службы включаются конструкторы и деструкторы).
Динамичность интерфейса: может ли меняться интерфейс службы, т.е. может ли меняться сигнатура и/или семантика услуг службы. Возможные значения этой характеристики:
Службы имеет постоянный по сигнатуре и семантике интерфейс. Например, так устроено большинство служб операционных систем.
Служба имеет консервативно расширяемый интерфейс (как, правило, так ведут себя службы горизонтальных приложений одной линии[71]).
Служба имеет заранее не известный клиентам интерфейс, который они должны определять в динамике (для этого используется какой-либо стандартизованный механизм динамического определения интерфейса).
В таблице 6 приведены некоторые примеры способов реализации служб.

Таблица 6. Примеры способов реализации служб

Характеристика	Представление	Идентификация объектов
Постоянная служба над множественными объектами с постоянным (или консервативно расширяемым) интерфейсом	Реляционная СУБД с хранимыми процедурами. У всех объектов одинаковая структура и они различаются только значениями свойств. Объект представляется записью в БД	Первичный ключ записи
Служба без памяти с единственным объектом и статическим интерфейсом	Библиотека динамического связывания	Отсутствует
Оперативная служба над множественными объектами и статическим интерфейсом	Библиотека динамического связывания. Объект представляется структурой (или объединением), указатель на которую передается параметром методам объекта[72]	Типизированный указатель
Оперативная служба над фиксированными объектами со статическим или динамическим интерфейсом	Библиотека объектов	Имя объекта

Выделение набора служб приложения является одной из основных задач концептуального проектирования, а выбор представления служб является одной из основных задач детального проектирования (см. соответствующие разделы ниже).

[71] Так называемая "преемственность снизу-вверх".
[72] Распространен предрассудок, что только синтаксис вида `Object.Method(Arguments)` является допустимым в ОО 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)`.

Замечание

> Не следует воспринимать таблицу 6 догматически: это только *примеры* возможных решений. В каждом конкретном случае умелый программист, как творец программы, волен в своих решениях. С другой стороны, не следует пытаться каждый раз изобретать велосипед, выдумывая новое представление для службы, которая по ОО характеристикам однородна с уже разработанной. Готовые программные решения представления служб следует заимствовать из репозитория, если они прошли проверку практикой.

5.4.4. Трехслойная архитектура

В недавнем прошлом широкое распространение получила типовая модель приложения, которая здесь называется *трехслойной* (three-tier). В наиболее рафинированной форме эта модель изложена в MSF. В этой модели приложение мыслится как набор взаимодействующих служб, которые распределены по трем слоям, перечисленным в Таблице 7.

Таблица 7. Трехслойная архитектура приложения

Слой	Функция
Пользовательский интерфейс (User Services)	Ввод и отображение информации, навигация в приложении
Бизнес-правила[73] (Business Services)	Принятие решений, обработка данных
Управление данными (Data Services)	Доступ к данным

Трехслойная архитектура обладает целым рядом очевидных структурирующих достоинств:

Выделение служб пользовательского интерфейса позволяет иметь сменные (альтернативные) интерфейсы для одного приложения. Такая возможность важна, например, если приложение предусматривает различное оборудование рабочих станций для различных категорий пользователей или если возникает потребность сменить интерфейс приложения по другим причинам.

Выделение служб обработки позволяет сосредоточиться на проектировании и реализации самих процедур обработки, формулируя их в терминах автоматизируемого бизнеса, безотносительно к деталям представления и отображения данных.

Выделение служб управления данным позволяет увеличить степень повторного использования программных решений, поскольку типичной является ситуация, когда над одной базой корпоративных данных строится несколько различных приложений.

Замечание

> В чисто информационных системах (т.е. в таких системах, которые представляют собой не более чем пользовательский интерфейс для просмотра данных в базе) слой служб обработки кажется излишним. Однако, если службы доступа к данным спроектированы с наибольшей возможной универсальностью, то службы обработки могут играть роль того, что теории баз данных называется подсхемой или представлением (view). Это дает возможность, например, запрограммировать любую схему защиты данных безотносительно к возможностям используемой СУБД.

Еще одной важной возможностью, которую предоставляет трехслойная архитектура, является маневренность, которая появляется при распределении служб по компонентам и компонентов по компьютерам в распределенном приложении. Здесь возникают различные варианты, например, такие, которые приведены на рис. 39 в разделе *Распределение компонентов*.

5.5. Проектирование программ

Проектирование занимает львиную долю в процессе разработки программы, а качество проектирования является решающим фактором повышения продуктивности программи-

[73] Этот крайне неудачный буквальный перевод уже широко используется, хотя гораздо лучше было бы называть данный слой служб просто "обработка".

рования (см. раздел *Продуктивность программирования*). В классических инженерных областях термин проектирование (или конструирование) имеет вполне определенный смысл: проектирование начинается, когда имеется описание (как правило, текстовое) назначения и требуемых характеристик проектируемого изделия (Технические условия) и заканчивается, когда появляются *чертежи*, которые можно воплотить "в металле". В программировании, которое приближается, но еще не достигло уровня классических инженерных дисциплин, термин *проектирование* носит более размытый характер. К проектированию программ относят работу над программой на различных стадиях ее жизненного цикла: определение назначения программы на фазе анализа (что аналогично формулированию Технических условий), функциональную спецификацию программы на фазе проектирования, построение кода методом пошагового уточнения на фазе реализации (которое уже гораздо ближе к воплощению "в металле").

Замечание

> Приведенная аналогия носит условный характер. Например, для современного CAD/CAM производства чертежи могут быть электронными моделями, а воплощение "в металле" может иметь форму перфоленты для станка с ЧПУ. Тем не менее, эта аналогия полезна, поскольку показывает направление, в котором ориентированы методы проектирования программ в дисциплине программирования . Целью является приближение *моделей* программ к тем значениям показателей управляемости, повторного использования, надежности и качества, которые достигнуты в *схемах* инженеров-электронщиков или *чертежах* инженеров-автомобилестроителей.[74]

Проектирование программы (в широком смысле этого слова) имеет различные цели и может принимать разные формы. В дисциплине программирования различаются следующие виды проектирования:[75]

- концептуальное проектирование;
- логическое проектирование;
- детальное проектирование.

5.5.1. Концептуальное проектирование

Концептуальное проектирование выполняется на фазе анализа и его результаты отражаются в Концепции и во Внешней функциональной спецификации. Методы концептуального проектирования сравнительно неформальны, а результаты концептуального проектирования, как правило, оформляются в виде текста на естественном языке с включением некоторых диаграммам и схем. На стадии концептуального проектирования должны быть достигнуты следующие основные цели.

5.5.1.1. Определение существа проблемы

Прежде всего, на стадии концептуального проектирования необходимо установить цель разработки программы: построение решения для вновь возникшей проблемы бизнеса, повышение эффективности имеющегося решения, снижение стоимости имеющегося решения, комбинация выше названного или нечто другое.

5.5.1.2. Идентификация предметной области

Необходимо определить границы области, которую покрывает разрабатываемое решение, причем как внешние границы (область желаемого), так и внутренние (область абсолютно необходимого).

---

[74] Простой пример из быта. Новичок *не может* самостоятельно построить печку в садовом домике удовлетворительным образом, поскольку обычная печка является довольно сложной тепловой машиной. Любой аккуратный человек с двумя руками может построить *хорошую* печку за три дня, если у него будет инженерно проработанная *порядовка* (особый вид чертежей, используемый в данной предметной области).

[75] Аналогичная классификация видов проектирования используется в MSF.

5.5.1.3. Классификация пользователей

Классификация подразумевает группирование пользователей по категориям и фиксацию характеристических признаков и особенностей каждой категории. Причем речь идет обо всех категориях пользователей: не только о пользователях, работающих с программой, но и об администраторах, сторонних клиентах, вовлеченных в процесс бизнеса и пр.

5.5.1.4. Составление сценариев

Сценарии являются основным выходом стадии концептуального проектирования. Сценарий — это фиксация на языке, в равной мере понятном как пользователям, так и разработчикам, существа процесса бизнеса, для которого разрабатывается решение. Используются два вида сценариев: "как есть" и "как должно быть". Если речь не идет о разработке абсолютно нового приложения, то сценарий "как есть" является наиболее надежным средством установления взаимопонимания между пользователями и разработчиками, а сценарий "как должно быть" является одним из самых эффективных средств определения требований.[76]
Рекомендуемым средством построения сценариев являются диаграммы использования (use-cases) на языке UML с подробными текстовыми комментариями.
Результаты концептуального проектирования *всегда* передаются заказчику и согласуются с ним.

5.5.2. Логическое проектирование

Логическое проектирование является основным содержанием фазы проектирования, результаты которой отражаются в различных спецификациях: Внешние функциональные спецификации, Внутренние структурные спецификации, Специальные технические требования. Логическое проектирование является ядром процесса разработки. Именно в этой стадии таятся корни успехов и неудач. К счастью, наиболее важный процесс логического проектирования в наибольшей степени подкреплен теоретическими, технологическими и инструментальными средствами. Такие концепции, как структурность, модульность, инкапсуляция, ОО, рассмотренные выше, поддерживаются практически всеми современными инструментальными средствами разработки программ. Некоторые примеры и рекомендации по применению инструментальных средств рассмотрены ниже в разделе *Инструментальные средства*.
Каковы бы ни были используемые инструментальные средства, основным содержанием логического проектирования является построение *модели* будущей программы. В данном контексте *модель* выступает как совокупность описаний, образующих единое целое и обладающих рядом свойств, обеспечивающих повышение продуктивности программирования (см. выше раздел *Спецификации и модели*).
Как уже было сказано, для построения моделей приложений типа информационная система масштаба предприятия в дисциплине программирования на стадии логического проектирования настоятельно рекомендуется использовать UML, возможно в комбинации с некоторыми дополнительными средствами.

Замечание

> Слова "моделирование" и "модель" многозначны. Здесь подразумевается то значение, которое соответствует английскому термину modeling, а не simulation. Модель в данном контексте — это, в первую очередь, средство описания (спецификации) программы и только в последнюю очередь средство проведения вычислительного эксперимента.

Центральной задачей логического проектирования в рекомендованной трехслойной модели приложения масштаба предприятия является вычленение и спецификация *служб*. Любая модель привносит структуру, а объектно-ориентированная модель, каковой является совокупность диаграмм UML, привносит объектную структуру и, таким образом, очень

[76] Составление сценариев «как должно быть» является основным, если не единственным, средством определения требований в экстремальном программировании.

удобна для описания служб приложения. Проектирование служб обычно начинается с определения номенклатуры услуг, затем производится группировка услуг по службам и, наконец, определяется структура объектов, обеспечивающих службы. В терминах диаграмм UML это означает, что начинать проектирование следует с диаграмм использования, которые по большей части создаются уже на стадии концептуального проектирования и могут уточняться на стадии логического проектирования. Затем диаграммы использования дополняются соответствующими диаграммами взаимодействия, где появляются объекты, структура которых уточняется с помощью диаграмм классов. Сложные услуги, которые требует выполнения многошаговых транзакций, удобно описывать с помощью диаграмм деятельности, а поведение сложных объектов с памятью удобно описывать с помощью диаграмм состояний. Диаграммы последовательности при логическом проектировании вертикальных приложений типа информационных систем используются сравнительно редко.[77]

Замечание

> На заре программирования программисты *писали* программы, то есть буквально рисовали код (или псевдокод, иногда с блок-схемами) на бумаге и этот код в бумажном виде некоторое время жил, подвергаясь обсуждению и модификации. Видимая причина "бумажного" программирования заключалась в технической трудоемкости непосредственного кодирования и высокой трудоемкости отладки плохо написанной программы. Дело в том, что работа велась в *пакетном режиме*. Цикл исправления пустяшной описки (трансляция – пропуск теста – локализация ошибки – исправление ошибки – трансляция – пропуск теста) мог занимать часы, а не секунды, который такой цикл занимает в современной интерактивной интегрированной системе программирования. Написать программу и пару раз просмотреть ее «в сухую», без прогона на машине, а не запускать ее сразу бывало выгоднее. Таким образом, написание программ являлось своеобразной формой проектирования. Развитие инструментальных средств избавило современных программистов от технических трудностей (сейчас никто программ на бумажке не пишет), но одновременно породило дурную, но, к сожалению, распространенную привычку проскакивать стадию логического проектирования и начинать "лепить" код, имея только смутную "концептуальную модель" в голове.

Умело программиста отличает наличие явной и полной модели программы, которая является результатом логического проектирования. Отсутствие модели снижает продуктивность программирования, за исключением небольшого числа следующих редко встречающихся случаев:

- маленькая программа (существенно меньше 1000 строк);
- высокая повторность задачи (тоже самое больше чем в третий раз подряд);
- выдающиеся индивидуальные способности[78] (объем сверхоперативной памяти превосходит мировую константу 7±2 более чем на порядок).

В области разработки вертикальных приложений типа информационных систем масштаба предприятия все исключения встречаются настолько редко, что ими можно пренебречь. Поэтому логическое проектирование и построение явной модели программы являются обязательными требованиями дисциплины программирования.

Результаты логического проектирования в обязательном порядке передаются заказчику в том случае, когда предполагается сопровождение и/или модификация программы силами заказчика. В противном случае можно ограничиться передачей только внешних функциональных спецификаций.

---

[77] Диаграммы последовательности используются для описания распараллеленных приложений реального времени, где время критично. По той же причине в рассматриваемом классе приложений редко используются активные объекты и механизмы синхронизации.

[78] Реальных супер программистов на порядок меньше, чем считается, а в области разработки информационных систем предприятий подвизается еще на порядок меньше.

5.5.3. Детальное проектирование

Детальное проектирование выполняется на фазе реализации и заключается в *трансляции* построенной на предыдущей стадии логической модели в коды используемых инструментальных систем программирования.

Замечание

> Границы между стадиями проектирования довольно размыты, потому что в сущности проектирование — это многошаговый процесс последовательного уточнения, стадии в котором выделяются условно в основном с целью административного планирования процесса разработки, привязанного к фазам и вехам (см. раздел *Модель процесса*). В качестве эмпирических правил для выделения стадий можно использовать следующие: появление компьютерной модели служит признаком перехода от концептуального проектирования к логическому, а появление в модели таких деталей, как атрибуты таблиц, сигнатуры методов и форматы сообщений служит признаком перехода от логического проектирования к детальному.

На стадии детального проектирования создаются следующие элементы проекта:
распределение компонентов;
скелет кода;
схема базы данных;
прототип интерфейса.
Результаты детального проектирования передаются заказчику только в том случае, если разработка ведется с открытым кодом, то есть если все права на программу отчуждаются от разработчика. В противном случае детали проекта, знание которых излишне при эксплуатации программы, могут только отпугнуть заказчика.

5.5.3.1. Распределение компонентов

Компонент – это элемент физической структуры кода приложения, т.е. исполнимый файл, библиотека динамического связывания, командный файл и т.п.[79] Проектирование компонентов требует решения двух вопросов:

- какие службы (объекты, функции) должны быть упакованы в одном компоненте;
- какие компоненты должны выполняться на каких компьютерах (в случае распределенной системы).

Критерием для ответа на первый вопрос является оценка выигрыша в продуктивности *программиста* за счет повторного использования компонентов. Критерием для ответа на второй вопрос является оценка выигрыша в продуктивности *программы* за счет повышения производительности, аппаратной надежности, снижения нагрузки на линии связи и т.п. Решение по первому вопросу обычно оформляется в виде диаграммы компонентов, а решение по второму вопросу оформляется в виде диаграммы размещения.
Наиболее простой вариант упаковки компонент в службы при использовании трехуровневой архитектуры, который может служить отправной точкой при проектировании, это когда каждому слою соответствует один компонент, т.е. один компонент содержит весь интерфейс, другой компонент – всю обработку и третий компонент – все информационные услуги. В этом случае каждый слой приложения является монолитным. Известен основной недостаток такой реализации: недостаточная гибкость. Действительно, маловероятно, что таким образом спроектированный компонент одного приложения окажется подходящим для использования в другом приложении. Другой предельный случай – каждая служба (объект) является отдельным компонентом. В этом случае возникают библиотеки объектов, имеющие тенденцию к быстрому росту. Библиотека мелких объектов создает видимость наилучшей структуризации, но на самом деле не обеспечивает наибольшей выгоды от повторного использования, поскольку сборка компонента из таких объектов фактически заново инициирует стадию логического проектирования.

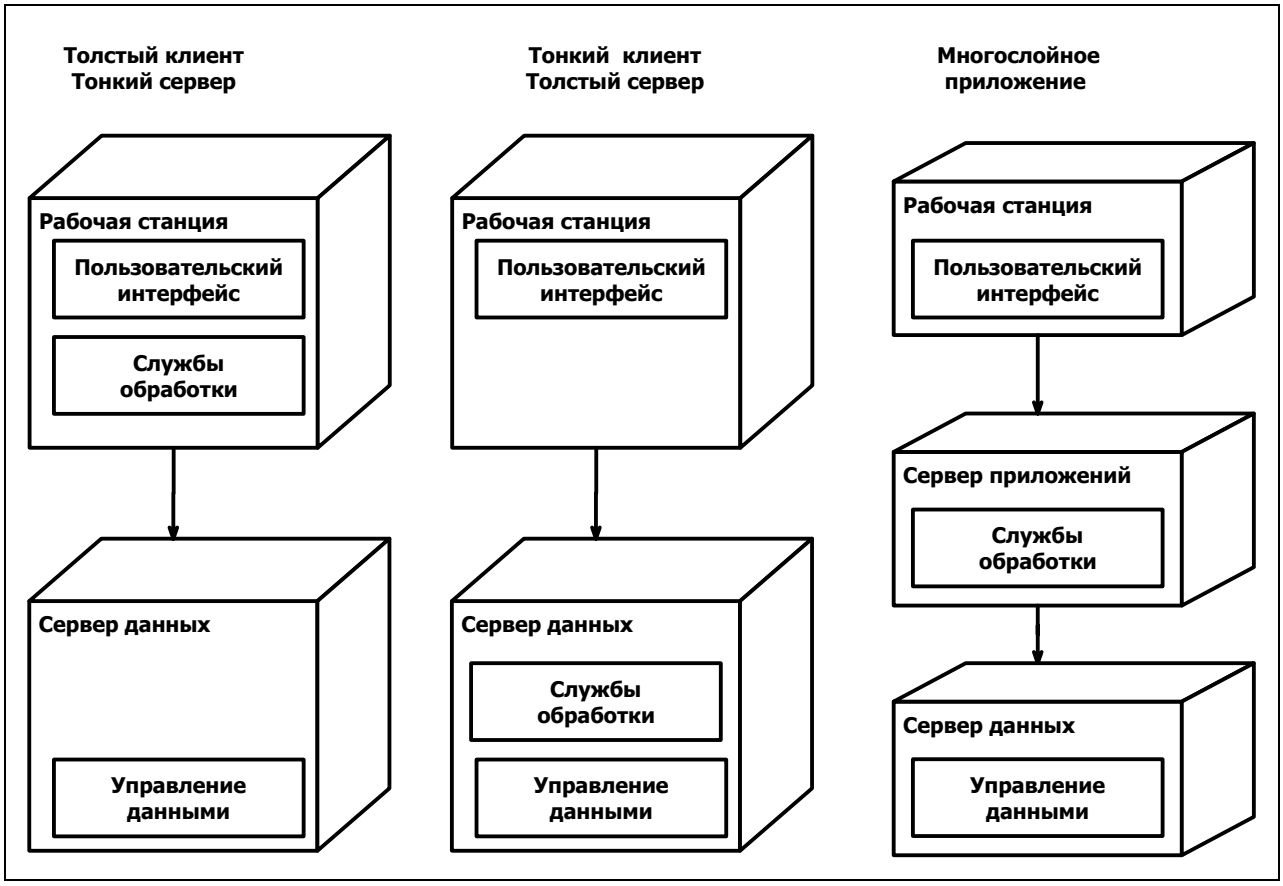
[79] Что именно может быть компонентом в каждом конкретном случае зависит от используемой административной системы времени выполнения приложения.

Интуитивно ясно, что для каждого конкретного набора инструментальных средств можно подобрать некоторую оптимальную величину компонента, при которой суммарный выигрыш в продуктивности от повторного использования компонентов окажется наибольшим. Невозможно дать априорную рекомендацию, какой должна быть величина компонента, руководством здесь должен быть здравый смысл и опыт. Хотя одна рекомендация, достаточно очевидная, может быть дана: в одном компоненте следует упаковывать службы, тесно связанные между собой, тогда как независимые службы могут быть расположены в разных компонентах. Такой подход позволит увеличить вероятность повторного использования компонента.

Редко случается так, что разработчик абсолютно свободен в выборе физической архитектуры, для которой создается приложение. Как правило, у заказчика уже существует некоторое аппаратное обеспечение и создаваемое приложение должно быть приспособлено к имеющимся ограничениям. В связи с этим при планировании распределения компонентов нужно перебрать только немногие технически возможные варианты. На рис. 39 приведены три стандартных модели распределения компонент для приложения трехслойной архитектуры и монолитными слоями. Как правило, для приложений стандартной архитектуры одной из этих моделей оказывается достаточно.[80] Если применяется более сложная упаковка служб в компоненты, то количество возможных вариантов размещения компонент возрастает.

Замечание

Распределение и перераспределение служб по компонентам и распределение и перераспределение компонент приложения по компьютерам является ключевым приемом управления производительностью при масштабировании приложений. На рис. 39 при просмотре слева направо при прочих равных условиях каждая следующая схема обеспечивает большую производительность и, одновременно, является более дорогой и более сложной в реализации.



[80] На рисунке не показан еще один возможный случай (тривиальный), когда все компоненты размещаются на одном компьютере. В таком случае диаграммы размещения не нужны.

Рис. 39. Варианты распределения компонентов

5.5.3.2. Скелет кода

Код программы не пишется от начала к концу: у умелого программиста, даже не использующего инструментальные средства CASE, заголовки *всех*[81] процедур появляются раньше, чем тело первой процедуры. Та часть кода, которая отражает модульную и объектную структуру программы, но еще не содержит деталей реализации, в дисциплине программирования относится к проектной документации стадии детального проектирования и называется *скелет кода*.[82] Современные развитые инструментальные средства разработки (см. раздел *Инструментальные средства*) поддерживают автоматическую (или автоматизированную) *генерацию* кода скелета. Например, почти все системы, поддерживающие UML, позволяют автоматически генерировать скелет кода. Если используемые инструментальные средства поддерживают возможность автоматической генерации, то этой возможностью следует обязательно пользоваться. Дело не только в экономии труда на ввод текста скелета, но, прежде всего, в том, что путем автоматической генерации скелета можно гораздо легче и надежнее поддерживать соответствие между логической моделью программы и физическим кодом. По очевидным причинам постоянная поддержка такого соответствия является обязательным требованием дисциплины программирования.

Замечание

> К сожалению, не все инструментальные средства корректно меняют генерируемый код при изменении модели (например, теряют введенный вручную код). Этот недостаток инструментальных средств не отменяет требования строгого соответствия модели и кода, но перекладывает ответственность на программиста.

5.5.3.3. Схема базы данных

Распространенные СУБД используют, в основном, реляционную модель данных. Эта модель отлично приспособлена для решения стандартных задач обработки транзакций над множествами объектов постоянной структуры, но не всегда оказывается адекватной при решении задач, выходящих за стандартные рамки. Программиста реляционной СУБД подстерегают, в частности, следующие подводные камни.

Стандартный язык программирования реляционных СУБД SQL является языком очень высокого уровня и не позволяет программисту контролировать выполнение примитивов. Например, программист SQL не может управлять порядком выполнения выборок, проекций, сортировок, группировок и соединений в сложном запросе и вынужден полагаться на встроенный (или отсутствующий!) механизм оптимизирующих *редукций*. В то же время этот порядок оказывает огромное влияние на эффективность выполнения запроса. SQL слишком легко позволяет написать катастрофически неэффективный запрос, причем эта неэффективность не бросается в глаза при чтении программистом текста запроса.[83]

Для эффективной работы реляционной СУБД база данных должна быть приведена к *нормальной форме*. Нормализация приспосабливает структуру базы к требованиям механизма интерпретации, причем эти требования далеко не всегда естественны с точки зрения предметной области. Более того, нормальные формы схем реляционных СУБД со строгих математических позиций нормальными формами не являются, потому что не обладают свойствами единственности и гарантированной достижимости. Например, многополюсная (не бинарная) ассоциация классов, вполне естественным образом возникающая при моделировании на UML, требует нетривиального преобразования в реляционную схему. Это преобразование может оказаться удачным или неудачным, а может и вовсе вызвать за-

[81] Во всяком случае, всех процедур одной службы.

[82] Хорошие системы программирования поддерживают скелет кода в явном виде, хотя называют его по разному: заголовочные файлы в C, **interface** в Паскале и т.п.

[83] Особенно если используются не эквисоединения по не ключевым атрибутам. В этом случае просто трудно сказать a priori как быстро будет выполняться запрос.

труднение у программиста. Программисту приходится вводить искусственные сущности, вспомогательные атрибуты для реализации связей и преодолевать сложности, порождаемые используемой СУБД, а не самой задачей.

Реляционные СУБД хорошо приспособлены для работы с регулярными множествами пассивных объектов постоянной структуры. Объекты, которые меняют свою структуру при смене состояния, допускают множественные значения свойств (например, история изменений) или иным образом уклоняются от канона, очень плохо ложатся в жесткую реляционную схему. Даже форматы представления значений примитивных типов не всегда оказываются согласованными с системой программирования, в которой предполагается реализовывать службы обработки и интерфейса. Наличие таких средств, как определяемые пользователем и динамические типы данных современными коммерческими СУБД, как правило, просто игнорируется.

Таким образом, на стадии детального проектирования возникает важная задача построения *схемы базы данных* приложения для используемой реляционной СУБД.

Замечание

> Составление схемы базы данных относится в дисциплине программирования на стадию детального проектирования, чтобы избежать отрицательного влияния жестких реляционных ограничений на принятие концептуальных решений на ранних стадиях проектирования. Начинать концептуальное проектирование с составления схемы базы данных методически неправильно и нецелесообразно, хотя и возможно.

Качество решения задачи проектирования схемы БД существенным образом влияет на такие характеристики приложения как производительность, надежность и масштабируемость, поэтому вопросу проектирования схемы следует уделить особое внимание. Логическое проектирование служб управления данными фактически определяет множество запросов, которые потребуются приложению для доступа к данным. Построение такого множества запросов на стадии логического проектирования позволяет получить надежный критерий правильности проектирования схемы БД на стадии детального проектирования: все запросы должны иметь очевидную реализацию в языке манипулирования данными используемой СУБД и должны иметь точную оценку времени выполнения (в терминах функций от длин хранимых таблиц). Если этот критерий не выполняется, то это служит признаком того, что схема БД спроектирована не вполне удовлетворительно, даже если схема семантически правильно отражает все существенные данные приложения и связи между ними.

Для проектирования схем БД целесообразно использовать дополнительные инструменты и приемы, хорошо приспособленные для решения именно этой задачи, например, так называемые *диаграммы "сущность-связь"* и соответствующие программы, например, BpWin. Использование автоматизированных инструментальных средств рекомендуется в данном случае по тем же причинам, что и при генерации скелета кода: сокращение ручного труда и повышение надежности соответствия модели и кода.

5.5.3.4. Прототип интерфейса

Современные инструментальные системы программирования содержат в своем составе различные средства автоматизации построения графического интерфейса пользователя. Как правило, в число этих средств входит так называемый *Дизайнер форм*, который позволяет нарисовать графический интерфейс пользователя, пользуюсь палитрой элементов управления, которую поддерживает система программирования. В большинстве случаев при этом автоматически генерируется скелет кода процедур, которые образуют службы пользовательского интерфейса (см. раздел *Событийное управление*). Набор элементов управления, образующих графический интерфейс пользователя разрабатываемого приложения, которые не связаны со службами обработки и управления данными (или связаны только частично) и, таким образом, не поддерживают функциональность приложения (или поддерживают только частично) называется *прототипом интерфейса*. Использование

средств автоматического построения прототипа интерфейса настоятельно рекомендуется дисциплиной программирования, поскольку позволяет обеспечить следующие важные характеристики:
согласованность (consistency) интерфейса;
низкую трудоемкость;
независимость интерфейса от прочих служб.
Согласованность (т.е. единообразие) *интерфейса* является ключевым приемом для обеспечения таких неформальных, но важных свойств, как "понятность" и "простота использования". Интерфейс должен быть предельно банальным, скучным и однообразным. Везде и всюду следует использовать простейшие стандартные элементы управления с параметрами оформления по умолчанию. Такой подход увеличивает вероятность того, что пользователь не будет испытывать затруднений при работе с приложением. Отклонение от стандартного оформления интерфейса, тем паче разработка собственных элементов управления и способов организации интерфейса являются в большинстве случаев неоправданными излишествами.[84] В то же время надлежит использовать *всю* палитру стандартных элементов управления, выбирая элемент подходящего типа в соответствии с задачей, а не вкусом программиста.

Замечание

> Некоторые программисты лелеют иллюзии, что с любовью разработанное приложение может само по себе заинтересовать пользователей. Это опасное заблуждение: пользователей интересуют только результаты использования приложения, и даже скорее интерпретация этих результатов в глазах их (пользователей) начальников.

Низкая трудоемкость построения прототипа интерфейса важна, потому что интерфейс – это весьма изменчивая вещь, для которой следует планировать вероятное построение *нескольких* прототипов при разработке приложения. Трудоемкость построения интерфейса должна быть настолько низка, чтобы было не жалко выбросить созданный прототип, если он чем-то не понравится заказчику, и быстро "нарисовать" новый. Это возможно только при систематическом использовании средств автоматического построения интерфейса.[85]
Независимость интерфейса от прочих служб при использовании автоматизированных средств обеспечивается гораздо надежнее, чем при ручном программировании, потому что автоматизированные средства, как правило, и не могут учесть наличие семантических связей интерфейса со службами обработки и управления данными.[86]

Замечание

> Построение прототипа интерфейса отнесено в дисциплине программирования на стадию детального проектирования, потому что гораздо легче и удобнее "рисовать" интерфейс по готовым диаграммам взаимодействия и имея специфицированные услуги служб обработки и управления данными. В то же время строить прототип интерфейса можно уже после концептуального проектирования по диаграммам использования. Иногда так и приходится делать, особенно если заказчик желает видеть внешний вид форм приложения уже во внешних функциональных спецификациях.

5.6. Кодирование

Кодирование, т.е. непосредственное составление и ввод текста программы на языке программирования, выполняется на фазе реализации.

---

[84] Если разработчик желает сделать свой интерфейс запоминающимся (в маркетинговых целях), то гораздо надежнее этому послужат анимированные заставки и другие украшательства "*сбоку*" от основного интерфейса. Можно сделать плавающие кнопки и это произведет сильное впечатление на заказчика, но скорее всего маркетинговый эффект будет отрицательный.
[85] Программисты склонны "цепляться" за вручную закодированный интерфейс.
[86] Исключением является использование дизайнера форм конкретной СУБД, который может навязывать связь интерфейса со схемой БД.

Замечание

Обычно используются системы программирования, в которых программы имеют вид линейных текстов, хотя это не всегда так. Например, "рисование" запроса к базе данных на бланке QBE или перетаскивание полей на третьем шаге мастера сводных таблиц Excel является, безусловно, кодированием, хотя при этом программист может и не видеть привычного линейного текста на экране.

Замечание

Для непосвященного кодирование отождествляется с программированием, но умелый программист ни на секунду не забывает, что код – это только видимая надводная часть айсберга программы (типичное соотношение 1:10).

Кодирование вторично по отношению к проектированию и в пределе может рассматриваться как особый вид детального проектирования. Тем не менее, при кодировании на традиционных текстовых языках программирования используются некоторые специфические приемы, часть из которых обсуждается ниже. Смысл и назначение всех этих приемов являются вариациями двух основных принципов повышения продуктивности, которые уже упоминались выше:
сокращение доли внепланово изменяемого кода;
увеличение доли повторно использованного кода.

5.6.1. Программирование по образцу

Умелые программисты знают, что, образно выражаясь, "все нужные программы уже давно написаны – остается их только найти, взять и использовать".[87] Умелое программирование (кодирование) базируется на систематическом применении образцов. *Образец* – это готовое (т.е. выраженное в форме кода) программное решение для некоторой частной, но типичной подзадачи. Образец может иметь различные формы в зависимости от используемой системы программирования и степени общности задачи. Некоторые формы представления и использования образцов кода приведены в таблице 4.

Таблица 8. Формы представления и использования образцов

Характеристика задачи	Примеры задач	Представление	Способ использования образца	Степень изменения кода образца
Служба с фиксированными сигнатурами услуг	Графические примитивы, арифметика неограниченной точности	Библиотека	Вызов функции	Код образца не меняется
Параметризованная по типам служба	Стек, Двусвязный список, другие стандартные, но не встроенные структуры данных	Шаблон	Подстановка	Регламентируется механизмом автоматической подстановки параметров шаблонов
Разовая многошаговая услуга	Просмотр текстового файла в данной системе программирования	Пример	Вставка текста с контекстной заменой	Ручное выделение и подстановка параметров
Ориентированная на предметную область служба	Начисление зарплаты	Модель	Повторное детальное проектирование	Неограниченное изменение кода
Абстрактный объект высокого уровня	Конечный автомат, таблица решений	Идея в голове	Фиксированная схема кодирования	Код создается заново

[87] К сожалению, последнее не всегда просто сделать, как показывает следующий классический английский анекдот. Капитан вызывает юнгу и приказывает: "Найди мой серебряный кофейник". Юнга спрашивает: "Если я знаю, где находится вещь, можно ли считать, что я ее нашел?". "Разумеется", – отвечает капитан. "В таком случае, ваш кофейник на дне моря, сэр".

Программирование по образцу – это прямой путь к повторному использованию кода и повышению продуктивности, именно поэтому накопление и актуализация запаса образцов в различных формах является важнейшей постоянной задачей как отдельно взятого умелого программиста, так и программирующего коллектива в целом.

5.6.2. Образцы проектирования

Особого обсуждения заслуживает последняя строка таблицы 8. С момента выхода в свет книги "банды четырех"[88] одной из наиболее горячих тем, обсуждаемых программисткой общественностью, стали образцы проектирования. [89] Образцы проектирования – это сравнительно новая форма обмена программистским опытом и удачными проектными решениями, но реализует она все ту же принципиальную идею повышения продуктивности – увеличение объема повторного использования кода.

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 может научить искусству программирования невероятно трудно — такие книги можно пересчитать по пальцам, в то время, как действующих мастеров программирования, обладающих ценнейшим опытом, многие тысячи.

---

[88] GoF — Gang of Four (Gamma, Helm, Johnson, Vlissides)

[89] Design Pattern — в программистский жаргон уже довольно прочно вошла калька с английского — "паттерн" (pattern), но мы, насколько это возможно, избегаем использования жаргона.

Появление UML радикальным образом изменило ситуацию. По нашему мнению, появление UML имеет для программирования 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 проектирования* — это параметрическая кооперация классов (т. е. шаблон кооперации). Напомним, 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 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.[90]

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 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. 40 приведена диаграмма кооперации, описывающая данный шаблон взаимодействия.

[90] 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.

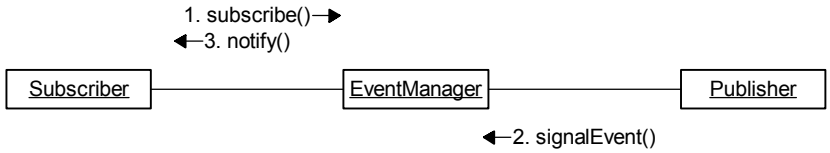


Рис. 40. Диаграмма кооперации для образца проектирования Publish-Subscribe

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 проектирования в дисциплине программирования. 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 – типичного вертикального приложения масштаба предприятия. 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. 41), что к классам `Department` и `Person` следует применить образец проектирования Publish-Subscribe, причем класс `Department` играет роль Publisher, а класс `Person` играет роль Subscriber.

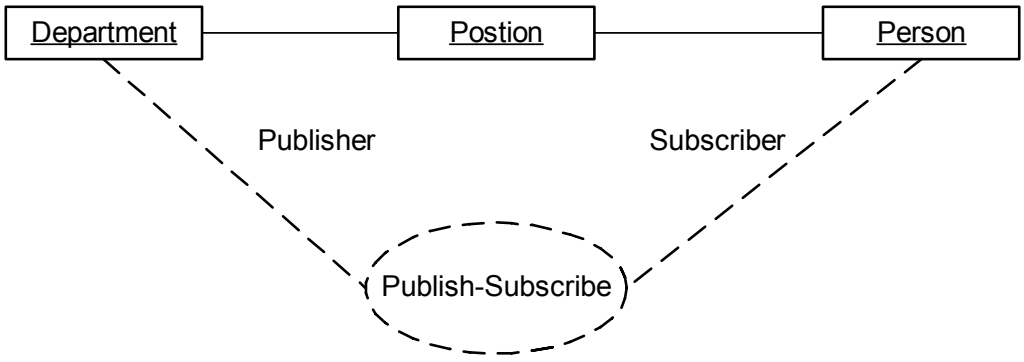


Рис. 41. Применение образца проектирования

По замыслу авторов UML, простая нотация на рис. 41 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.

5.6.3. Доказательное программирование

Идея автоматического *доказательства правильности* (или *автоматической 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. Существует целый ряд частных случаев,[91] 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.

Замечание

> Ручное доказательство правильности целесообразно использовать для верификации часто используемых небольших *образцов*. 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.

Замечание

> Вставка утверждений в программу иногда называется *аннотированием программ*.

В простейшем случае утверждения представляют собой эффективно вычислимые предикаты над значениями переменных. Например, условия, ограничивающие область допустимых значений входных аргументов функции, инварианты цикла, ограничители числа итераций плохо сходящихся процедур и т.п. Если программист *знает*[92] какое-либо (доста-

[91] Например, если известны инварианты всех циклов.
[92] Это знание может быть почерпнуто из спецификации, из опыта или из частичной ручной верификации.

точно сильное) условие на значения переменных в данной точке программы, 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 формального параметра процедуры содержит, как правило, неформальное описание области допустимых значений этого параметра. 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 (signal) или исключительная ситуация (exception).

Запись утверждения в виде ловушки. 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.

Замечание

> 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 другие дело, когда программа хотя и не работает как должно, но вежливо информирует об этом пользователя.

5.6.4. Программирование вширь

Выше, в разделе *Структурное программирование* было введено понятие *программирование вширь*, как один из вариантов последовательности разработки. Применительно к кодированию эта идея означает, что код пишется не по горизонтали (по слоям приложения), а по вертикали (по функциям приложения), т.е. путем последовательного наращивания услуг в службах всех слоев. Кодирование методом программирования вширь обладает целым рядом преимуществ:

быстрое появление работающей версии приложения, которую можно регулярно демонстрировать заказчику (наращивая постепенно функциональность);

отсутствие стадии автономной отладки и, как следствие, более тщательное и дисциплинированное оформление программистами кода с самого начала;

частичное совмещение по времени комплексной отладки с кодированием и, как следствие, сокращение продолжительности фазы стабилизации.

Замечание

> Программирование вширь подразумевает некоторое увеличение объема тестирования, но при умелом программировании, когда каждый подключаемый вертикальный срез содержит не слишком много ошибок, программирование вширь оказывается более продуктивным, чем привычное программирование сверху вниз.

Программирование вширь оказывается особенно естественным, если используется стандартная трехслойная архитектура приложения и службы реализуются в объектно-ориентированной среде. В этом случае расширение функциональности сводится к доопределению классов и кодирование становится консервативным и предсказуемым: код только добавляется и добавленный на каждом шаге код всегда нужный и правильный.

Замечание

Программирование вширь подразумевает полное и тщательное логическое проектирование, т.е. точную спецификацию всех услуг всех служб до начала кодирования любой услуги, в противном случае кодирование может оказаться неконсервативным и сопряженным с внеплановым переписыванием кода, что снижает продуктивность. Например, пусть в службе доступа к данным, которые являются динамически изменяемым множеством однородных объектов, кодируется услуга, вычисляющая некоторую интегральную характеристику множества. Тогда естественным представлением множества может быть связный список. Но если в этой службе предполагается еще и услуга поиска, которая критична по времени выполнения, то список, скорее всего, будет неудачным решением и, может быть, разумнее использовать хэширование. При наличии исчерпывающей спецификации услуг службы, созданной на стадии логического проектирования, умелый программист может выбрать адекватное представление и написать безошибочный код с первой попытки; если же спецификация неполна или неточна, то ошибки кодирования, которые придется исправлять и переписывать, очень вероятны.

5.6.5. Форматирование кода

В случае использования традиционной системы программирования с линейным языком, большое значение имеет оформление текста программы. В момент написания программы начинающему программисту зачастую кажется, что он пишет программу для компьютера, но это не так. Программу, которую не придется *много* раз читать *человеку*, не стоит и писать. Плохо оформленный текст читать трудно, и это существенно снижает продуктивность. Хорошо оформленный текст читать легко и приятно, и это существенно повышает продуктивность. Для повышения читабельности[93] программы используются следующие приемы:
синтаксически ориентированный текстовый редактор;
комментарии;
дисциплина имен;
расположение текста.

5.6.5.1. Синтаксически ориентированный текстовый редактор

В системе программирования, которой пользуется умелый программист, должно быть предусмотрено *удобное* средство ввода и редактирования текста программы. Обычно это синтаксически ориентированный редактор, т.е. текстовый редактор, который "знает" синтаксис языка программирования и позволяет ускорить ввод и редактирование программы. Например, могут предусматриваться следующие возможности:
автоматическое завершение ввода стандартных лексем языка;
клавиатурные комбинации для ввода стандартных лексем языка;
автоматический ввод парных ограничителей (скобок);
автоматическое выравнивание и отступ;
автоматические выделение (например, цветом) лексических и синтаксических конструкций;
всплывающие подсказки по синтаксису конструкций и сигнатурам стандартных функций.
Если система программирования имеет эти или аналогичные возможности, то ими надлежит систематически пользоваться. Если таких средств нет, то умелому программисту нетрудно их создать, запрограммировав какой-либо текстовый редактор общего назначения. Выгода от систематического использования специальных средств редактирования состоит в единообразии оформления текста программ и, тем самым, улучшении читабельности.

[93] Этот корявый термин, который теперь внесен в словари русского языка, возник именно из практики программирования.

5.6.5.2. Комментарии

Современные императивные языки программирования, которые называются языками "высокого уровня"[94] на самом деле имеют очень низкий уровень. Низкий уровень языка проявляется в том, что смысл программы не так просто усмотреть в ее тексте. Причина заключается в очень низком (можно сказать, примитивном) уровне архитектуры (системы команд) современных массовых компьютеров и незначительных алгоритмических возможностях компиляторов языков программирования.[95]

Замечание

> Формальный язык совсем не обязательно должен иметь низкий уровень. Вот два примера: язык алгебраических формул, UML.

Все без исключения современные языки программирования содержат механизм для частичной компенсации означенного недостатка, который называется комментариями.

Комментарий – это неопределяемое расширение языка.

С точки зрения языка, определяется только способ, каким компилятор может игнорировать комментарии. Таким образом, эта часть языка программирования предназначена исключительно для человека.[96]

Умелые программисты пишут комментарии всегда (но, к сожалению, по-разному). В дисциплине программирования рекомендуется следующий принцип комментирования*: все определяющие вхождения всех имен и только они должны иметь отдельный комментарий*.

Замечание

> Этот принцип особенно хорош для распространенных императивных процедурно ориентированных языков, в которых все семантически существенные объекты (модули, процедуры, переменные и константы) именованы.

Поскольку в подавляющем большинстве случаев определяющее вхождение имени единственно и текстуально предшествует использующим, комментарии оказываются расположенными в тексте программы привычным образом и попадаются на глаза читателю в удобной для восприятия последовательности при естественном порядке чтения текста программы.

Если язык допускает как явные, так и неявные объявления (например, по первому использующему вхождению), то неявного объявления следует избегать (кроме, может быть, совершенно элементарных случаев объявления существенно локальных переменных, таких как, например, счетчик цикла).

Комментарии не должны содержать тривиальной информации, непосредственно следующей из ближайшего контекста. Например, не следует указывать в комментарии конструктор типа в типизированных языках, потому что тип имени и так виден; не следует начинать комментарий к имени функции со слов "Эта функция…", потому что и так ясно, что комментарий относится именно к этой функции и т.п.

Комментарий обязательно должен содержать неявно подразумеваемую программистом информацию, которая не видна в тексте программы. Например, если целая переменная на самом деле используется как битовая шкала (множество флагов), то это обязательно следует указать. Самое важное назначение комментария – это указать семантическую связь между комментируемым объектом в программе и элементом модели, который представляется данным программным объектом.

---

[94] Корректно было бы говорить "языки несколько более высокого уровня, чем машинный код".

[95] Распространенные компиляторы производят, в сущности, тривиальное преобразование текста, поскольку опираются только на формальный синтаксис программы, а не на ее семантику. Как только компилятор удается наделить хотя бы самым элементарным пониманием смысла программы, уровень входного языка резко возрастает. Поучительный исторический пример на эту тему можно найти в книге Э.Х. Тыугу *Концептуальное программирование*.

[96] В некоторых системах программирования можно обнаружить попытки частично извлекать прагматику или семантику программы из комментариев (псевдокомментарии, указания компилятору, проверяемые утверждения и т.п.).

5.6.5.3. Дисциплина имен

Единственными лексемами языка, которыми программист пользуется совершенно произвольно, являются имена (идентификаторы).[97] Недисциплинированный программист использует свободу выбора идентификаторов для глупых шуток. Умелый программист использует эту свободу для повышения читабельности программы.

Читабельность программы повышается, если пишущий программу придерживается определенных правил формирования идентификаторов, а читающий программу знает эти правила. Правила формирования идентификаторов в программе называются *дисциплина имен*.

Дисциплина имен должна отражать три аспекта:

набор различных характеристик имен (и области значений этих характеристик), которые учитываются в данной дисциплине,

набор правил формирования идентификаторов по заданным значениям выбранных характеристик (с учетом возможных лексических ограничений системы программирования),

набор операций (помимо операции чтения человеком), которые выполняются над множествами имен.

Набор различных характеристик имен, которые целесообразно учитывать в дисциплине имен, зависит от языка программирования и вряд ли может быть предложен универсальный набор, пригодный во всех случаях. Для рассматриваемого случая императивного объектно-ориентированного типизированного языка можно, например, дисциплинировать следующие характеристики имен.

Тип (включая класс и вид именуемого элемента). Например: *массив* (это класс, определяет конструктор элемента) *функций* (это вид, определяет синтаксический контекст, в котором может использоваться элемент), возвращающих *целое* (это тип, определяет множество операций, применимых к элементу).

Семантика (т.е. основное назначение именуемого элемента). Наилучший способ задания семантики имени – указать связь именуемого элемента программы с элементом логической модели. Проще всего этого добиться, согласовав дисциплину имен при моделировании и кодировании.

Прагматика (т.е. указание особого способа использования именуемого элемента). Например, *итератор* (процедура организации цикла по некоторой структуре данных) или *функционал* (функция, которая работает только вместе с некоторой другой функцией, передаваемой параметром) и т.п.

Структурная позиция (т.е. указание, где находится определяющее вхождение имени). В ОО языках этой же цели до некоторой степени служит использование *составных* имен различного вида.

Специфические особенности (например, области видимости или времени жизни имени).

Набор правил формирования идентификаторов зависит от различных особенностей системы программирования:

допустимые символы (буквы, цифры, знаки, пробелы, подчеркивания и т.д.);

различение регистра букв в идентификаторах;

ограничения на общую длину и/или на длину распознаваемой части идентификатора.

Набор выполняемых операций зависит от вариантов использования текста программы (помимо очевидных: чтения и компиляции). Например, с текстом программы могут проводиться следующие операции.

Поиск в тексте (или в множестве текстов) определяющего вхождения для данного использующего вхождения имени или же наоборот, поиск для данного определяющего вхождения всех использующих. Потребность в этой операции часто возникает при анализе "чужой" программы.

---

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

Извлечение из текста (или из множества текстов) списка имен с фильтрацией и/или сортировкой и/или группировкой. Такая операция полезна, например, при анализе использования стандартных библиотек.

Систематическое переименование (с возможной фильтрацией). Например, потребность в систематическом переименовании (с заменой и перестановкой частей идентификаторов) возникает при использовании в качестве образца фрагмента кода, составленного с применением другой дисциплины имен.

Известен целый ряд хорошо проработанных дисциплин имен, из которых чаще всего упоминается так называемая *венгерская нотация*. Фактически это название объединяет множество сходных дисциплин, в основе которых лежит следующая идея. Идентификатору (как слову) приписывается некоторая искусственная морфология. Как правило, идентификатор предлагается делить на следующие части: *префикс*, *приставка*, *корень*, *суффикс* и *окончание*.[98] Далее определяется, какую характеристику идентификатора отражает каждая его часть. Как правило, в диалектах венгерской нотации корень должен отражать семантику, приставка – тип, суффикс – прагматику, префикс – специальные характеристики. Окончание используется для индивидуализации имен, которые по всем остальных признакам совпадают. Например, следуя венгерской нотации, переменные для хранения вещественных корней квадратного уравнения в процедуре Visual Basic можно было бы назвать dblRoot_1 и dblRoot_2 (здесь dbl – это приставка, задающая тип, Root – это корень, а _1 и _2 – окончания; префикс и суффикс не использованы). Кроме того, в конкретной дисциплине имен регламентируются (часто в виде предопределенных списков) наборы возможных значений кодовых частей (таких как префикс и приставка), определяются правила (неформальные) выбора корней, суффиксов и окончаний, а также правила, по которым можно опускать те или иные части идентификатора. Для многих языков и систем программирования имеются детальные описания конкретных диалектов венгерской нотации. Венгерская нотация является неплохой дисциплиной имен и ее можно рекомендовать к использованию, особенно в следующих обстоятельствах:

венгерская нотация уже строго описана (имеется стандарт) для данного языка и системы программирования;

проект предусматривает передачу кода и заказчик не возражает против данной дисциплины имен;

в проекте повторно используется большой объем кода, написанного в венгерской нотации.

В то же время, следует иметь в виду, что венгерская нотация

не отражает структурную позицию имени,

затрудняет выполнение систематического переименования,

требует согласования с системой типов языка программирования.

Выбор между готовой дисциплиной имен и разработкой собственной (часто путем усечения лишнего) является прерогативой руководителя проекта.

Замечание

> Поскольку при моделировании и кодировании используются, как правило, разные системы программирования, то маловероятно, что они все окажутся локализованными, причем с одинаковым пониманием особенностей русского языка. Отсюда следует, что идентификаторы, как правило, приходится записывать, используя буквы латинского алфавита. Для условных (кодовых) частей идентификаторов это не важно и даже удобно: иероглиф не должен быть похож на слово.[99] Но для содержательных (семантических) частей желательно, чтобы они являлись узнаваемыми словами или словосочетаниями. Использование транслитерирован-

[98] Здесь указаны русские названия частей слова. В буквальных переводах иноязычных описаний диалектов венгерской нотации можно встретить различные "тэги", "квалификаторы" и пр. Суть от этого не меняется.

[99] 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.

ных русских слов выглядит ужасно.[100] 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.[101]

В идеальном случае, когда имена элементов логической модели могут быть прямо перенесены в корни идентификаторов, а правила формирования прочих частей идентификаторов просты и достаточны для отражения всей информации об именах, необходимой для понимания их назначения и способа использования, строгая дисциплина имен оказывается могучим средством повышения читабельности и продуктивности. Идеал достижим с трудом, но к нему следует стремиться, поэтому дисциплина программирования не навязывает конкретной дисциплины имен, но требует *наличия* в каждом конкретном проекте, предполагающем написание объемного кода, документированной дисциплины имен.

5.6.5.4. Расположение текста

Для компьютера расположение текста программы (т.е. отступы, выравнивание, ширина строк, пустые строки и т.п.) не имеет, разумеется, никакого значения. Однако для человека, который читает программу, это имеет огромное значение. Для улучшения читабельности за счет форматирования используются следующие приемы.

Отступы. В языках допускающих вложенность конструкций для выделения структуры вложений используются отступы. При этом элементы вложенной конструкции сдвинуты вправо относительно элементов объемлющей конструкции, которые могут находиться до и/или после вложенной конструкции. Величина отступа не должна быть большой (слишком большие отступы затрудняют чтение) – для моноширинных шрифтов достаточно 2-3 знакомест. Самое главное, чтобы отступы использовались *единообразно*.

Замечание

Величина отступа должна *строго* соответствовать семантической вложенности конструкций. Решение вопроса о том, что с чем следует выравнивать по вертикали должно опираться на *семантику* языка, а не на случайные особенности синтаксиса. Очень удобно придерживаться следующего принципа: удаление и вставка вложенных конструкций всегда выполняется над целой строкой. Например,

```
function Next (x : integer) : integer; begin
  if x >= 0 then begin
    Next := x + 1;
  end else begin
    Next := x - 1;
  end;{x >= 0}
end;{Next}
```

существенно лучше, чем более привычная запись

```
function Next (x : integer) : integer;
  begin
    if x >= 0 then Next := x + 1
              else Next := x - 1
  end;
```

Пробелы и *пустые строки*. В большинстве языков почти все пробелы, равно как и пустые строки, игнорируются, а несколько пробелов эквивалентно одному. Разумное употребление пробелов и пустых строк позволяет до некоторой степени компенсировать убожество

[100] 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.

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

моноширинных шрифтов фиксированного кегля, которые навязывают некоторые (даже сравнительно современные) системы программирования.

Замечание

> При употреблении пробелов и пустых строк важны единообразие и чувство меры.

Ширина строки. Большинство читателей программ привыкло, что основная конструкция языка (оператор) занимает одну строку программы, во всяком случае начинается с новой строки.[102] Отступление от этого принципа сейчас вряд ли можно чем-либо оправдать. Однако бумага и экран, на которых отображается текст программы, имеют весьма ограниченную ширину.[103] Это не составляет проблемы в лаконичных языках типа Форт, но в многословном Visual Basic приходится пользоваться такими устаревшими приемами, как символ продолжения строки и т.п. Текст программы хорошо читается, если примерно половина знакомест пусты, а по ширине в среднем занято примерно две трети отведенного пространства.

5.7. Тестирование и отладка

Тщательные исследования и измерения производительности, проведенные еще в период первой революции в программировании, показали, что *в среднем* при промышленном программировании прямые трудозатраты на изготовление окончательного продукта составляют очень малую долю суммарно затраченного рабочего времени программистов – не более 20%. Основная же часть затраченного времени и ресурсов приходится на непроизводительные потери и на косвенные трудозатраты, связанные с выявлением и исправлением *ошибок*. Процесс, имеющий целью выявление ошибок в программе, называется *тестирование*, а процесс исправления ошибок называется *отладка*.

За прошедшие четверть века ситуация изменилась, но все еще оставляет желать лучшего: доля тестирования и отладки в среднем неприемлемо велика.

Замечание

> Тестирование и отладка (может быть, под другими названиями) присущи почти всем видам инженерной деятельности. Очевидно, что чем ниже доля затрат на тестирование и отладку, тем лучше организована инженерная деятельность. В этом смысле *средние* показатели для программирования по прежнему принадлежат к числу наихудших.

Развитие технологии программирования за прошедший период, ориентированное на повышение продуктивности, шло по трем основным направлениям:

совершенствование инструментальных систем программирования с целью сокращения непроизводительных потерь рабочего времени программистов;

совершенствование методов тестирования и отладки с целью снижения трудозатрат на этот процесс (при заданном объеме);

совершенствование самого программирования с целью уменьшения объема тестирования и отладки.

По всем трем направлениям достигнуто заметное, но неравномерное продвижение. Наибольший прогресс достигнут в развитии инструментальных средств программирования: воплощение программистской мысли в машинно-читаемую форму на всех стадиях и фазах программирования может выполняться быстро и эффективно. Визуальное моделирование, рисование интерфейса, эффективный ввод и редактирование текста программы, получение справочной информации о повторно используемых компонентах, управление версиями и проектами надежно обеспечиваются практически любой инструментальной системой программирования и соответствующими утилитами. От технологов и руководителей программных проектов требуется только *выбрать* подходящие инструментальные средства

---

[102] Именно поэтому объем исходного кода программы следует измерять *в строках*, а не в килобайтах.

[103] Особенно это сказывается в современных многооконных системах программирования, где для отображения самого текста программы остается не так уж много места.

(см. раздел *Инструментальные средства*) и обеспечить их освоение и использование программистами.

Наименьший прогресс, к сожалению, наблюдается в самих парадигмах программирования, в особенности массового промышленного программирования. *Безошибочное программирование* пока остается близким, но, в среднем, не достигнутым идеалом.

Замечание

> Из сказанного не следует, что идеал безошибочного программирования недостижим, более того, не следует, что он недостижим в ближайшем будущем. Революционные изменения в физических принципах действия и архитектуре устройств обработки информации вполне возможны. Изменение аппаратной архитектуры повлечет за собой смену парадигмы программирования и сами понятия тестирования и отладки могут неузнаваемо измениться.[104]

Таким образом, следует исходить из того, что в рассматриваемом типичном случае разработки приложения типа информационной системы программа, полученная на фазах проектирования и реализации, *неизбежно содержит ошибки*. Вопрос заключается в *количестве* и *качестве* ошибок. Если ошибок настолько много и/или они настолько тяжелы, что программа не удовлетворяет требованиям и ожиданиям пользователя, то программу нельзя считать отлаженной, а в противном случае – можно. Целью тестирования и отладки, выполняемых на фазе стабилизации, является выполнение определенных мероприятий, направленных на снижение количества ошибок, имеющихся в программе, до уровня, приемлемого для заказчика.

5.7.1. Критерии приемлемости

Простыми комбинаторными рассуждениями несложно показать, что для нетривиальной программы исчерпывающее тестирование невозможно. Тестирование всегда частичное. Отсюда следует, что тестирование может обнаружить ошибку в программе, но не может достоверно установить отсутствие ошибок. Таким образом, не существует объективного достоверного критерия (основанного исключительно на результатах тестирования), позволяющего сделать вывод о достижении приемлемого для пользователя уровня ошибочности программы. Выводы о количестве ошибок, оставшихся не выявленными в программе, являются сугубо приближенными и делаются на основании различных статистических, вероятностных и эмпирических соображений, среди которых наибольшее распространение получили следующие.

5.7.1.1. Представительное тестирование

Представительным называется набор тестов, при прогоне которых выполняется определенный объективный критерий, связанный с самой тестируемой программой. Например, все исполнимые операторы кода прорабатывают хотя бы один раз. Если при прогоне представительного набора тестов ошибки не выявляются, то программа считается приемлемой. К сожалению, построение представительного набора тестов для нетривиальной программы ненамного проще построения самой программы. На практике, особенно для многофункциональных программ с развитым интерфейсом, часто считают достаточно представительным набор, в котором каждая функция программы (грубо говоря, каждый элемент управления интерфейса) оказывается задействованной хотя бы один раз.

5.7.1.2. Регрессионное тестирование

В этой группе методов управления тестированием используются хорошо известные в других инженерных областях различные статистические методы выходного контроля. Например, измеряется количество известных ошибок как функция продолжительности тестирования. Если производная по времени отрицательна, то тестирование называется *сходящимся*. Программа объявляется приемлемой, если тестирование сходится, и в течение

[104] Например, что такое "программирование" и "отладка" персептрона или нейросети?

заданного интервала времени *t* количество известных ошибок не превышает пороговой величины *n* (обычно полагают *n*=0). Такой подход к управлению тестированием, применяет, например, Microsoft, подробности описаны в дополнительных материалах MSF. Понятно, что как и в любом другом статистическом методе, для получения значимых результатов нужно прогнать достаточно *много* тестов,[105] поэтому тестирование оказывается весьма трудоемким процессом. Предлагается использовать автоматическое тестирование и генерацию тестов, что требует разработки специального инструментального средства (отладочного средства). Регрессионное тестирование наиболее целесообразно использовать в больших проектах и в проектах, связанных с разработкой горизонтальных приложений.

5.7.1.3. Тестирование по соглашению

Самым простым и удобным с точки зрения разработчиков является случай, когда заранее известен набор тестов, прогон которых без выявления ошибок означает, что программа приемлема для заказчика *по соглашению*. При этом вполне возможно, что в программе даже остались известные разработчику и не устраненные ошибки, но контрольные примеры, удовлетворяющие заказчика, выполняются успешно и этого достаточно. Такой прагматический подход к тестированию можно объяснить следующими обстоятельствами. Конкретного заказчика интересует в первую очередь эффект, который вертикальное приложение оказывает на его (заказчика) бизнес, и только во вторую очередь "правильность" приложения и т.п. Известно, что влияние приложения на бизнес распределено по функциональности приложения далеко не равномерно: только небольшая доля функций критически важна для бизнеса, остальными можно пожертвовать (временно) без особого ущерба.

Замечание

> При использовании UML уже на стадии концептуального проектирования можно выделить, документировать и согласовать тот набор тестов, которые будут являться тестами приемлемости по соглашению.[106]

5.7.2. Виды тестирования

Различаются следующие виды комплексного тестирования.

Тестирование устойчивости (reliability). Целью этого вида тестирование является выявление не перехватываемых ошибок времени выполнения, т.е. тесты устойчивости нацелены на то, чтобы "сломать" приложение. Типичными приемами, применяемыми при тестировании устойчивости, являются ввод данных, выходящих за пределы области допустимых значений, нарушение порядка действий, предусмотренных сценарием, создание ситуаций, нарушающих количественные ограничения.

Тестирование функциональности (functionality). Целью этого вида тестирования является проверка того, что для допустимых (правильных) входных данных получаются допустимые (соответствующие спецификациям) результаты. Ожидаемые правильные результаты для каждого теста функциональности должны быть получены заранее независимо от тестируемого приложения. Типичным приемом является ввод предельных (находящихся на границе области допустимых значений) данных.

Тестирование применимости (usability). Целью этого вида тестирования является проверка того, что предусмотренные способы использования приложения являются удовлетворительными для пользователя при выполнении типичных сценариев работы. При тестировании применимости проверяются, например, следующие параметры: реактивность, за-

---

[105] В данном случае дело усугубляется тем, что нет никаких достоверных сведений о вероятностных характеристиках измеряемых случайных величин

[106] Этот подход в явном виде использовался в стандарте ЕСПД: документ "Программа и методика испытаний" по ЕСПД является одним из обязательных и ранних документов проекта.

щищенность данных, способность к восстановлению после ошибки, понятность интерфейса и др.

Обязательным требованием дисциплины программирования является наличие *плана* тестирования, регламентирующего порядок проведения тестирования по видам тестов и критериям приемлемости.

Ход тестирования, выявленные ошибки и приятые меры по их исправлению фиксируются в базе данных тестирования.

5.7.3. Методы отладки

Отладка заключается в *локализации* и *исправлении* ошибок, выявленных тестами. Как правило, локализация ошибки более трудоемка, чем ее исправление. Это объясняется тем, что тест обнаруживает внешнее проявление ошибки в форме несоответствия результата спецификациям, причем это несоответствие может быть очень отдаленным следствием истинной причины, которая и является ошибкой, подлежащей исправлению.

Самым распространенным приемом локализации является просмотр программистом *протокола выполнения* программы с целью обнаружения того места, где поведение программы, демонстрируемое протоколом, расходится с ожидаемым поведением. Протокол может иметь различные формы: это может быть действительно файл или база данных, в которой фиксируются сведения о ходе выполнения программы, или это может быть динамически изменяемое состояние программы, ход выполнения которой воспроизводится в интерактивном отладчике.

Замечание

> Интерактивный отладчик, который имеется в большинстве современных систем программирования, является полезным инструментом, котором должен владеть умелый программист.

Отладчик удобен тем, что позволяет легко и просто проследить ход выполнения программы и изменение значений ее переменных. Не следует, однако, переоценивать возможности локализации ошибок с помощью отладчика и игнорировать другие методы отладки. Прослеживание хода работы программы по шагам – это крайнее средство, которое стоит применять, только если более эффективные приемы не позволили локализовать ошибку достаточно точно.

Другим распространенным приемом получения протокола является трассировка, которая также доступна в большинстве систем программирования. *Трассировка* – это автоматизированная генерация статического протокола (log file). Как минимум, в трассу включается протокол потока управления с точностью до функций, иногда добавляется и дополнительная информация об изменении состояния. Трассировка имеет перед отладчиком то преимущество, что трассу можно просматривать в любом порядке, а не только от начала к концу. Иногда просмотр трассы с конца позволяет быстрее локализовать ошибку.

Умелый программист заранее знает, что отладка неизбежна, что в его любимой программе есть многочисленные ошибки и их придется мучительно искать. Поэтому умелый программист начинает готовиться к отладке еще на стадии детального проектирования и кодирования, включая в программу дополнительный код, который сможет помочь при тестировании и отладке. Конкретные используемые приемы специфичны для системы программирования, но общая идея состоит в том, чтобы завести специальную базу данных и расставить в коде ловушки и "отладочные печати", которые бы протоколировали в этой базе семантически важную информацию о ходе выполнения программы. Такого рода дополнительный код, написанный с целью упрощения отладки, называется *отладочный стенд*.

Замечание

> Наличие средств условной компиляции (и им подобных) позволяет создавать сколь угодно развитые отладочные стенды без вреда для эффективности программы.

Удивительно эффективным методом отладки является т.н. "структурная экспертиза",[107] когда автор программы публично читает код программы вслух перед аудиторией из коллег по разработке. Присутствующие могут задавать вопросы и делать замечания, автор должен их фиксировать, но не должен отвечать. При этом очень важно, чтобы чтение велось именно вслух (это задает правильный темп информационного обмена), чтобы это было именно чтение готового текста (это исключает добавление исправлений и фантазий по ходу чтения) и чтобы вопросы и замечания (в том числе и прежде всего самого автора[108]) тщательно фиксировались. Не испытав на практике трудно в это поверить, но объективно измерено, что эффективность такой отладки (средние затраты времени на обнаружение, локализацию и исправление одной ошибки) в разы превышает эффективность других методов, когда программист отлаживает свой код один на один с компьютером.

Локализованную ошибку надлежит *исправить*. Исправление ошибок также имеет различные формы:

Изменение кода. Ошибки кодирования (см. раздел *Продуктивность программирования*) надлежит исправлять, т.е. заменить неправильный код на другой код (возможно, правильный).[109]

Изменение спецификаций. Ошибки проектирования иногда невозможно исправить, не выйдя за рамки бюджета проекта. Иногда в таким случаях бывает возможно приемлемым для заказчика образом изменить спецификации таким образом, что наблюдаемое поведение программы переходит из разряда ошибочного в разряд допустимого.[110]

Обход ошибки. В некоторых случаях программа может допускать выполнение одной функции несколькими способами. Если существует альтернативный способ выполнения функции, то неработающий вариант можно просто запретить, оставив только обходной путь.

Фиксация ошибки. Самым неприятным является случай тяжелой проектной ошибки, которую нет времени исправить. В таком случае ее надлежит зафиксировать в документации и оставить "как есть".

5.8. Инструментальные средства

Выбор инструментальных средств разработки является важной (хотя и не критически важной) задачей, для которой прерогатива принятия решений принадлежит безусловно руководителю проекта, а ответственность за проведение принятых решений в жизнь делегируется в одну из ролей команды фазы.[111]

Замечание

> Практика показывает, что, как правило, имеется некоторый набор альтернативных инструментальных средств, *пригодных* для данной разработки. Все эти средства оказываются примерно одинаково продуктивными, различие в продуктивности является практически несущественным. Т.е. не следует уповать на применение какого-то необыкновенного инструмента, который явится панацеей и сможет сам по себе гарантировать успех. В тоже время, для каждой разработки можно указать набор привлекательных, но *непригодных* для данной разработки инструментов, попытка использования которых предопределяет неудачу. Задача руководителя проекта состоит в том, чтобы отсеять непригодные инструменты и выбрать из пригодных один, руководствуясь политическими, историческими, корпоративными и другими субъективными соображениями.

Совершенно обязательным требованием дисциплины программирования является использование одного инструмента из данного класса инструментов всеми без исключениями

---

[107] Термин принадлежит Ф.Я. Дзержинскому, подробно описавшему и обосновавшему этот прием.

[108] Подавляющее большинство ошибок находит сам автор, но присутствующие должны быть и должны внимательно следить за происходящим, в противном случае эффект не наблюдается. Видимо, этот метод опирается на малоизученные свойства человеческого подсознания.

[109] Если заменяющий код содержит больше ошибок, чем заменяемый, то говорят, что процесс отладки *расходится*.

[110] Это не bug, a feature.

[111] На разных фазах одного проекта используются разные инструментальные средства

разработчиками – самодеятельность в выборе инструментов является грубым нарушением технологической дисциплины. Например, даже смена версии используемого компилятора в ходе проекта является экстраординарным и исключительным событием.

5.9. Выводы

В программирующей организации дисциплина программирования является частью корпоративного стандарта и в этом смысле является нормативным документом. В тоже время, дисциплина программирования нацелена на непрерывное улучшение и совершенствование программирования в организации, а не на фиксацию и замораживание существующего порядка. Поэтому положения дисциплины программирования являются, в основном, рекомендующими и предписывающими, а не ограничивающими и запрещающими.
Несмотря на гибкость дисциплины программирования, в ней имеется неизменяемое жесткое ядро. Далее следует список основных положений, которые являются характеристическими для дисциплины программирования и которые должны неукоснительно выполняться при проведении каждого проекта.

- Программирование рассматривается как экономически обоснованная деятельность, имеющая целью получение прибыли организацией.
- Дисциплина программирования ориентирована на повышение продуктивности программирования за счет снижения доли внепланово изменяемого кода и увеличение доли повторно используемого кода.
- Модель жизненного цикла программы в дисциплине программирования согласована с моделью процесса.
- Повышение продуктивности в дисциплине программирования достигается за счет унификации уровня представления информации в циклах повышения продуктивности.
- Дисциплина программирования постулирует примат проектирования над кодированием, тестированием и отладкой.
- Проектирование разделяется на три стадии: концептуальное, логическое и детальное проектирование, которые выполняются, соответственно на фазах анализа, проектирования и реализации. Во всех случаях конкретному кодированию предшествует абстрактное моделирование.
- Кодирование в дисциплине программирования следует общепринятым правилам хорошего стиля и рассматривается как публичная деятельность, а код является корпоративным достоянием.
- Тестирование в дисциплине программирования является планируемым и протоколируемым.

Продуктивность программирования является мерой умелости программистов , а потому дисциплина программирования является *сознательной* дисциплиной.

Предметный указатель

40-часовая рабочая неделя....38
Автоматическая верификация программы....122
Акт сдачи-приемки....69
Активный интерфейс....104
Аннотирование программ....122
Артефакт....19
Архитектура клиент/сервер....106
Архитектура приложения....103
Архитектура программного обеспечения....25
Архитектурная метафора....38
Безошибочное программирование....130
Бригада проекта....86
Валидация....31
Венгерская нотация....127
Верификация....31
Веха....19
Взаимодействие....76
Виртуальные машины....25
Виток....18
Выпуск....14
Декларативное программирование....96
Дерево субординации....76
Деструктор....98
Диаграмма "сущность-связь"....116
Дизайнер форм....116
Динамическое планирование....38
Динамичность интерфейса....108
Дисциплина имен....126
Дисциплина программирования 12, 89, 99
Доказательство правильности программ....122
Должностная структура....85
Должность....84
Доступный заказчик....38
Жизненный цикл программы....13
Заинтересованное лицо....20
Иерархическая модель....76
Императивное программирование....96
Инженерный анализ программы....28
Инкапсуляция....97
Инкремент....35
Инкрементная модель процесса....35
Инспектирование....29
Инструмент....20
Инструментальная программа....20
Интерфейс....97
Информатика....7
Итерационная стратегия....34
Итерация....19
Кодирование....27, 117
Коллективное владение кодом....38
Команда проекта....74
Команда фазы....86
Комментарий....125
Компонент....113
Конструктор....98
Контекстное модульное тестирование..31
Контроль качества....29
Концептуальное проектирование....110
Кооперация....76
Линейная стратегия....34
Линейно-функциональная организация 75
Ловушка....123
Логическое программирование....96
Мастер....106
Матричная структура организации....75
Метод пошагового уточнения....93
Метод хирургической бригады....77
Множественность экземпляров....108
Модель....17, 111
Модель водопада....34
Модель вычислимости....95
Модель главного программиста....77
Модель звезды....77
Модель команды....12, 74
Модель команды равных....79
Модель конвейера....34
Модель программы....102
Модель процесса....11
Модуль....93
Модульное программирование....94
Нагрузочное тестирование....32
Наследование....97
Независимые компоненты....25
Непрерывная интеграция....38
Непроцедурное программирование....96
Нормальная форма....115
Обзор....30
Образец....118
Образец проектирования....119, 120
Объектно-ориентированное программирование....97
Опережающее тестирование....38
Отзыв заказчика....69
Отладка....129
Отладочный стенд....132
Отладчик....132
Отступ....128

Оценка риска....66
Пакетный режим....112
Парадигма программирования....92
Парное программирование....38
Пассивный интерфейс....104
Периодическая отчетность....71
Полиморфизм....97
Постоянство объектов....107
Потоки данных....25
Представительное тестирование....130
Представительный набор тестов....130
Приемосдаточные испытания....32
Принцип единоначалия....76
Пробел....128
Программирование....6
Программирование без go to....93
Программирование вширь....94, 123
Программирование по образцу....119
Программирование сверху вниз....94
Программирование снизу вверх....94
Продолжающаяся разработка....32
Продукционное программирование....96
Проектирование программ....110
Проектно-ориентированная организация....75
Простое проектирование....38
Прототип интерфейса....116
Процедурное программирование....96
Процесс....17
Пустая строка....128
Работник....20
Реализация....27
Регрессионное тестирование....31, 131
Репозиторные архитектуры....25
Рефакторинг....38
Роль....82
Руководитель проекта....85
Системное тестирование....31
Системный аналитик....99
Скелет кода....115
Служба....98, 107
Событийное управление....104
Согласованность интерфейса....117
Сопровождение программы....32
Состояние требования....23
Специализация....76
Спецификация....101
Спортивное программирование....90
Стадия....18
Стандарт кодирования....28
Стандарты кодирования....38
Структурное программирование....92
Схема базы данных....116
Тестирование....28, 129
Тестирование инсталляции....32
Тестирование интерфейсов....31
Тестирование применимости....131
Тестирование удобства и простоты использования....32
Тестирование устойчивости....131
Тестирование функциональности....131
Технология программирования....7
Трассировка....132
Требование....21
Трехслойная архитектура....109
Управление требованиями....23
Усовершенствование приложения....32
Устранение дефектов....32
Учет рабочего времени....71
Фаза....18
Функциональное программирование....96
Функция....82
Цель тестирования....31
Частые выпуски....38
Человеческий фактор....80
Ширина строки....129
Эволюционная стратегия....34
Экземпляр....98
экстремальное программирование....37
Этап....18
Язык Пролог....96

Литература

1. Бек К. Экстремальное программирование. – Спб.: Питер, 2002.
2. Брауде Э. Технология разработки программного обеспечения. – СПб. : Питер, 2004.
3. Брукс-мл. Ф. П. Как проектируются и создаются программные комплексы. М.: Наука, 1975; новое издание перевода: Мифический человеко-месяц. СПб.: СИМВОЛ+, 1999.
4. Вирт Н. Алгоритмы + структуры данных = программы. М., Мир, 1978.
5. Вирт Н. Систематическое программирование. Введение. М.: Мир, 1977.
6. Дал У., Дейкстра Э., Хоор К. Структурное программирование. М.: Мир, 1975.
7. Дейкстра Э. Дисциплина программирования. М.: Мир, 1978.
8. Орлов С. Технологии разработки программного обеспечения. – СПб.: Питер, 2002.
9. Терехов А.Н. Технология программирования. М.: БИНОМ, 2006.
10. Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения. СПб : Питер, 2002.