ПРОГРАММИСТУ

Л. А. Мацяшек, Б. Лионг

ПРАКТИЧЕСКАЯ программная инженерия

на основе учебного примера

БИНОМ

УДК 681.1.06
ББК 32.973-018.2
М36

Серия основана в 2005 г.

Мацяшек Л. А.

М36 Практическая программная инженерия на основе учебного примера [Электронный ресурс] / Л. А. Мацяшек, Б. Л. Лионг ; пер. с англ. — 2-е изд. (эл.). — М. : БИНОМ. Лаборатория знаний, 2012. — 956 с. : ил. — (Программисту).

ISBN 978-5-9963-1182-8

Рассмотрены вопросы современных методов создания сложного программного обеспечения, использующего информацию, хранимую в базе данных. Подчеркнуты особенности создания такого программного обеспечения коллективом разработчиков: итеративный характер разработки, использование стандартных средств создания программ (стандартные компоненты, паттерны, Bean-компоненты и т. д.). Большое внимание уделено разработке структуры программного обеспечения, позволяющей наиболее просто организовать все стадии его жизненного цикла. Весь материал проиллюстрирован на одном достаточно сложном примере.

Для разработчиков сложного программного обеспечения, а также для студентов вузов, специализирующихся в вопросах создания современного ПО.

УДК 681.1.06
ББК 32.973-018.2

По вопросам приобретения обращаться:
«БИНОМ. Лаборатория знаний»
Телефон: (499) 157-5272
e-mail: binom@Lbz.ru, http://www.Lbz.ru

ISBN 978-5-9963-1182-8

Интерактивное оглавление

Начинать с определения основной терминологии — хороший стиль в области передачи знаний (типа написания книги или производства ПО). Термины нужно только определять и разъяснять, но по возможности не изобретать. Действительно, большинство терминов, с которыми мы работаем, было изобретено до нас.

Эта книга о ПО и системах. Она посвящена программной инженерии и разработке систем ПО. В ней рассматриваются большие проекты ПО и информационные системы предприятий. В книге используются следующие определения, данные видными экспертами в рассматриваемой области.

«*Программная инженерия* — область информатики, имеющая дело с созданием систем ПО, которые являются настолько большими или настолько сложными, что создаются коллективом или коллективами инженеров» [34]. В этом определении имеется ряд важных моментов.

«Создание систем ПО» — другими словами можно сказать «*разработка систем*». На самом деле программная инженерия — это несколько большее, чем собственно разработка ПО. Она включает также и *сопровождение ПО*. Подобно любому техническому изделию, типа моста или дома, ПО должно поддерживаться. В отличие от случая типичного технического изделия, сопровождение ПО включает его развитие (добавление новых модулей) и глубокие изменения в самих основах продукта. В этом смысле разработка ПО дополняется его обслуживанием.

«*Система* — целенаправленное собрание находящихся во взаимосвязи компонентов, которые работают вместе для достижения некоторой цели» [96]. Разработка системы (и обслуживание) связана с процессами, методами и инструментальными средствами производства ПО. Поскольку ПО является моделью действительности, его разработка связана с *моделированием* продуктов ПО.

Система — это больше, чем просто ПО. «*Инженерия систем* связана со всеми аспектами разработки и развития сложных систем, в которых ПО играет главную роль» [96]. «Инженерия систем концентрируется на разнообразии элементов, анализе, проектировании и организации этих элементов в систему, которая может быть изделием, сервисом или технологией преобразования информации или управления» [81]. А так как ПО играет главную роль в большинстве современных систем, эта книга также и об инженерии систем.

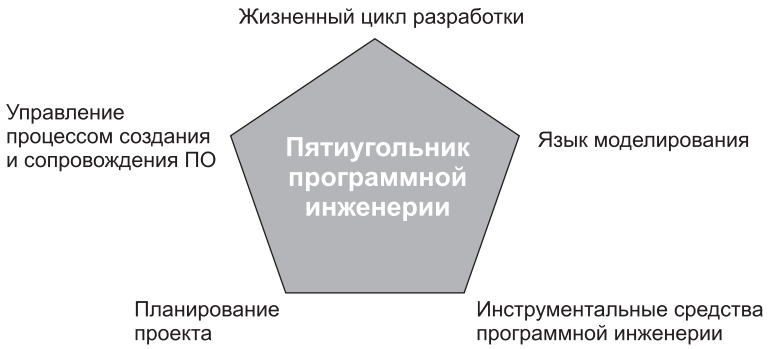
Большинство проектов ПО выполняется в связи с организационными потребностями заняться задачами идентификации в бизнес-процессе или улучшением этого процесса из-за конкуренции. *Проектирование ПО* — запланированная операция, предназначенная для создания программного продукта или сервиса и выполняющаяся в течение определенного времени. В этой книге особое внимание уделяется проектам ПО, предназначенным для *информационных систем предприятий*. Это большие и сложные системы. Как отмечено Фаулером [31], «Промышленные приложения часто содержат сложные данные, и большинство их работает на основе бизнес-правил, которым не хватает логических проверок».

Standish Group [98] исследует причины *отказов ПО*. Оригинальный *Отчет о хаосе* (Chaos Report), выполненный Standish Group в 1994 г., сообщил, что только 16.2 процентов проектов ПО были закончены вовремя и

в пределах выделенных средств. Насчет более крупных и сложных систем сведения оказались даже еще хуже: только 9 процентов таких проектов были закончены вовремя и без перерасхода средств. *Отчет о хаосе* за 2003 г. показал улучшение в этом деле — 34 процента проектов были закончены вовремя и в пределах выделенного бюджета. Хотя улучшение существенно, все это выглядит мрачно по сравнению с традиционными техническими дисциплинами типа архитектуры или электротехники.

Успех реализации проекта ПО обусловлен пятью взаимосвязанными аспектами — см. *пятиугольник программной инженерии*, изображенный ниже. Эти аспекты следующие:

- жизненный цикл разработки ПО — глава 1;
- язык моделирования ПО — глава 2;
- инструментальные средства программной инженерии — глава 3;
- планирование работ над проектом ПО — глава 4;
- управление процессом разработки и сопровождения ПО — глава 5.



Основные учебные цели части 1 состоят в том, чтобы получить знания в следующих вопросах:

- сущность программной инженерии;
- стадии жизненного цикла и модели; в частности, итеративная и пошаговая разработка;
- языки моделирования ПО, в частности, UML — объектно-ориентированный язык моделирования;
- инструментальные средства программной инженерии для управления проектом, моделирования систем, интегрированного коллективного программирования, управления изменением и конфигурацией проекта;
- планирование работы над проектом и оценка бюджета;
- технологии отслеживания хода выполнения проекта;
- управление людскими ресурсами, привлеченными к проекту;
- управление рисками проекта;
- управление качеством ПО;
- управление изменением и конфигурацией.

Глава 1

Жизненный цикл разработки программного обеспечения

В обычном использовании термин «жизненный цикл» означает «изменения, которые происходят в жизни животного или растения» [18]. В программной инженерии термин «**жизненный цикл**» (на английском языке lifecycle — обычно пишется единым словом) применяется к искусственным системам ПО и означает изменения, которые происходят в «жизни» программного продукта. Различные стадии между «рождением» изделия и его возможной «смертью» известны как **стадии жизненного цикла**.

Изменения, а следовательно, и стадии, являются последовательными. Изделие *создается* поэтапно, за ряд стадий. Таким образомь, разработка является повторяющейся и пошаговой. В конечном счете, изделие поэтапно *выводится из работы* — его использование постепенно прекращается. Следовательно, и прекращение его работы является пошаговым. Разумно думать, что ПО в любое время, кроме стадии непосредственного внедрения, находится или в фазе создания, или в фазе снятия с производства. Сопровождение ПО, несмотря на его эволюционный характер, одновременно начинает и процесс снятия с эксплуатации.

Рис. 1.1 показывает типичные стадии жизненного цикла ПО (объясненные более подробно в разделе 1.2). Эти стадии следующие:

1. анализ требований;
2. проектирование системы;
3. реализация;
4. интеграция и внедрение;
5. процесс функционирования и сопровождения.

Рис. 1.1 демонстрирует, что как только программный продукт внедрен в организацию, он остается там навсегда, хотя и под различными «перевоплощениями». Организация уже не может вернуться к ручному способу ведения бизнеса. Однажды включенный в действие, программный продукт поддерживается «до смерти».

Сопровождение, даже если оно развивает систему, ведет, в конечном счете, к ухудшению ее первоначальной структуры. Система становится **унаследованной системой** — она не может быть больше «усовершенствована», и даже вспомогательное и корректирующее сопровождение становится большой проблемой. Вся система или основные ее компоненты должны постепенно удаляться. Осознание, что система является унаследованной, приводит к

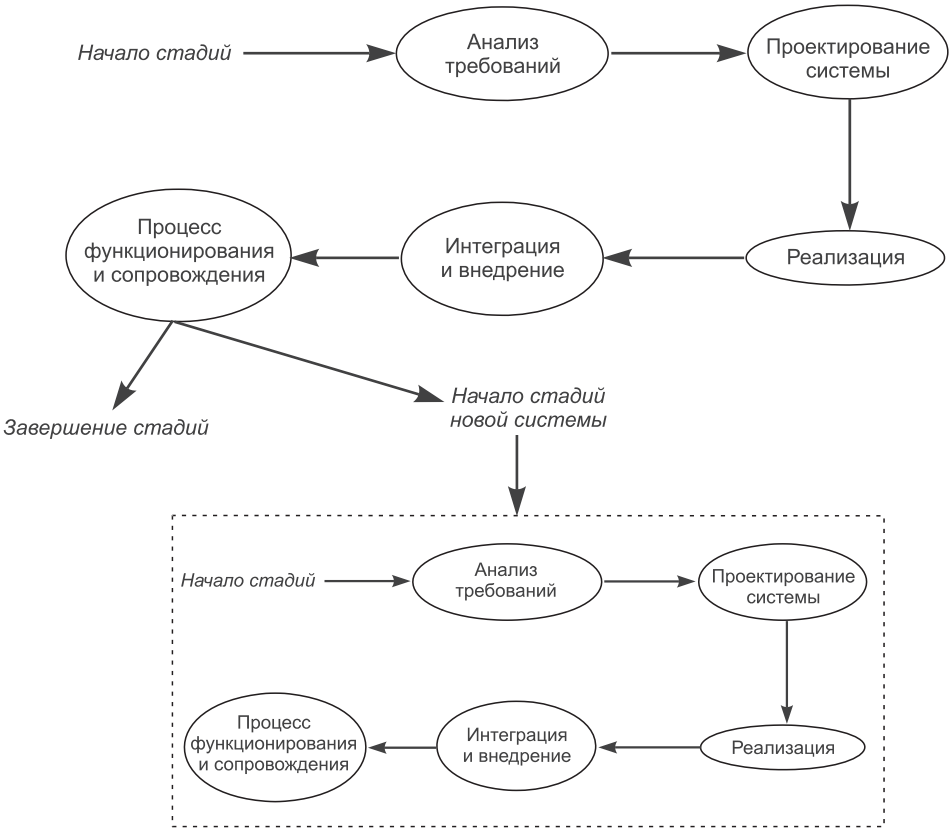


Рис. 1.1. Стадии жизненного цикла ПО

решению разработать новую систему. Это стимулирует новый жизненный цикл, показанный в нижней части рис. 1.1. Постепенное сокращение старой системы и синхронизация с новой системой проводится в параллель, пока новая система не будет полностью развернута для пользователей. Но даже и после развертывания старая система может оставаться в эксплуатации в течение некоторого времени, пока новая система не продемонстрирует свою полноценность.

Особенностью рис. 1.1 является отсутствие *тестирования* как стадии жизненного цикла. Тестирование, как и действия по управлению проектом, включая сбор *системы показателей* проекта, является всеобъемлющей деятельностью, которая выполняется на всех стадиях жизненного цикла.

1.1. Сущность программной инженерии

Понимание жизненного цикла ПО является необходимым условием понимания сущности программной инженерии — ее фундаментальной природы, контекста формирования ПО. Суть **программной инженерии** отражается в следующих ключевых выводах:

- система ПО меньше, чем информационная система предприятия;
- процесс создания и эксплуатации ПО является частью бизнес-процесса;
- программная инженерия отличается от традиционной инженерии;
- программная инженерия больше, чем программирование;
- программная инженерия напоминает моделирование;
- система ПО сложна.

1.1.1. Система ПО меньше, чем информационная система предприятия

Система ПО просто является частью намного большей **информационной системы предприятия**. Это означает, что разработка системы ПО является только частью (хотя и фундаментальной) разработки информационной системы предприятия. Диаграмма Венна на рис. 1.2 демонстрирует включение системы ПО в информационную систему предприятия. Она показывает также, что информационная система предприятия является компонентом предприятия как целого, и что предприятие является частью бизнес-среды.

Информационная система обеспечивает формирование и управление информацией в интересах людей. Часть этой информации генерируется автоматически компьютерными системами. Другая информация вводится людьми вручную. Дело в том, что информационные системы — социальные системы, которые включают и используют ПО и другие компоненты в интересах предприятия. Бенсон и Стандинг [7] перечисляют следующие компоненты информационной системы:

- люди;
- данные/информация;
- процедуры;
- ПО;
- аппаратное обеспечение;
- линии связи.

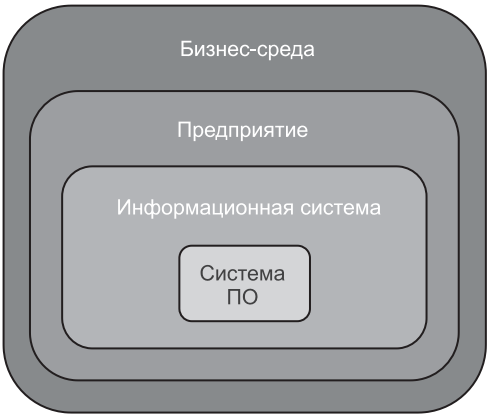


Рис. 1.2

Например, система управления счетами банка состоит из кассиров банка, данных/информации о клиентах и их счетах, процедур, управляющих изъятием и внесением денег, ПО, способного обработать данные/информацию, аппаратных средств, на которых может работать ПО (включая банковские автоматы), а также персональных и автоматизированных каналов связи, которые объединяют все эти компоненты в единую систему.

1.1.2. Процесс создания и эксплуатации ПО является частью бизнес-процесса

Процессом можно назвать нечто, где запланированы, организованы, скоординированы и выполнены в определенный период времени некоторые виды деятельности по производству продукта или сервиса. Различие между процессом создания и эксплуатации ПО, с одной стороны, и бизнес-процессом, с другой, определяется и связано с продуктом или сервисом, которые ожидается получить от этих процессов. Результат **процесса создания и эксплуатации ПО** — ПО. Результат **бизнес-процесса** — бизнес.

Имеются четкие отношения между ПО и бизнесом. ПО — потенциально главный вкладчик в бизнес-успех. ПО — часть бизнеса, но не наоборот. Фактически это отношение старшинства было отображено на рис. 1.2. Предприятие на рис. 1.2 — это другой термин для бизнеса. Цель функционирования предприятия состоит в том, чтобы сформировать цепочку создания ценностей, которая обеспечивает реализацию бизнес-назначения, задач и целей.

Различие между процессом создания, эксплуатации ПО и бизнес-процессом сродни различию между эффективностью процесса и результативностью. **Эффективность** (efficiency) означает делать что-то правильно. **Результативность** (effectiveness) означает делать правильную вещь. В организационных терминах результативность подразумевает достижение бизнес-назначения, задач и целей. Все они — то, что нужно получить как результат процесса *стратегического планирования*, проводимого на предприятии. Частью стратегического планирования является *бизнес-моделирование*. Следовательно, целью бизнес-процесса является обеспечение результативности.

В противоположность этому процесс создания и эксплуатации ПО должен обеспечить эффективность. Следовательно, возможна ситуация, когда процесс создания и эксплуатации ПО даст очень *эффективный* программный продукт или сервис, который будет *нерезультативен* для бизнеса. В лучшем случае нерезультативность может означать нейтральный результат с точки зрения бизнеса. В худшем случае это может сделать бизнес уязвимым для конкурентов и даже привести к банкротству.

Поэтому ясно, что процесс создания и эксплуатации ПО является характерной частью бизнес-процесса, жизненно важной для успеха предприятия. Чтобы обеспечить результативность наряду с эффективностью, процесс создания и эксплуатации ПО должен быть составной частью бизнес-процесса. В конце концов, решение разрабатывать ли программный продукт или сервис в первую очередь будет результатом стратегического планирования и бизнес-моделирования.

Процесс программной инженерии устанавливает соответствие ПО и бизнес-процессов. С одной стороны, разработка ПО все более и более внедряет-

ся в среду бизнес-моделирования. Главы 6 и 7 этой книги иллюстрируют явное проявление этой тенденции. С другой стороны, разработка ПО предназначена для поставки программных продуктов и сервисов, увеличивая для предприятия стоимость бизнеса. Это имеет отношение к трем **уровням управления**, которые бизнес-процессы обслуживают: оперативный, тактический и стратегический.

Помещение разработки ПО в среду бизнес-моделирования означает, что процесс создания и эксплуатации ПО получен из более широкой бизнес-модели, и он старается поддерживать и реализовывать конкретный бизнес-процесс в этой модели. Отсюда следует, что программный продукт/сервис не может быть только информационным сервисом. Он должен также реализовывать бизнес-операции или содействовать им. Проект информационной системы должен или явно определить бизнес-процесс, который он обслуживает, или, что лучше, он должен быть частью *системы управления знаниями*. Одним из аспектов такого проекта является координация между автоматическими информационными действиями, ручными действиями и творческими действиями по принятию решения.

Обычно система ПО обслуживает один уровень управления — оперативный, тактический или стратегический (рис. 1.3). *Оперативный уровень* обрабатывает оперативные бизнес-данные и документы, типа заказов и счетов. Это — царство OLTP-систем (**online transaction processing** — оперативная обработка транзакций), сопровождаемых обычной технологией *баз данных (БД)*. *Тактический уровень* обрабатывает информацию, полученную от анализа данных, типа ежемесячных тенденций в заказах продуктов. Это — царство OLAP-систем (**online analytical processing** — аналитическая обработка в реальном времени), которые сопровождает технология *хранилищ данных*. *Стратегический уровень* обрабатывает организационные знания, типа правил и фактов, обусловливающих высокий уровень выгодной продажи изделий. Это — царство систем знаний, которые сопровождает технология *баз знаний*.

Программные продукты/сервисы на оперативном уровне обязательны для предприятия. Без них современное предприятие не может функционировать.

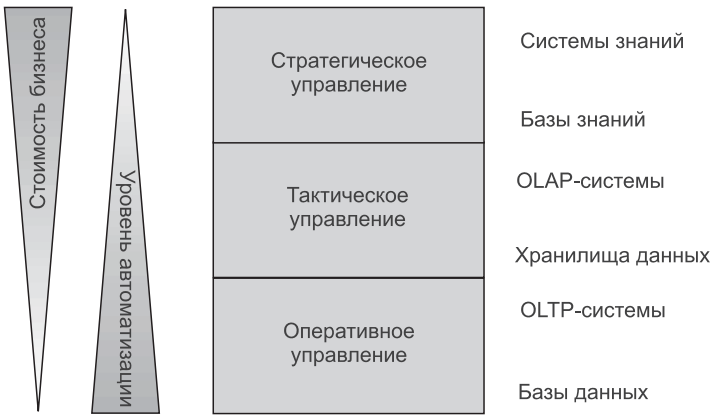


Рис. 1.3. ПО различных уровней управления

Однако одно оперативное ПО не дает предприятию никакого конкурентного преимущества. Ведь конкуренты также работают с подобными системами ПО. Бизнес-ценность ПО увеличивается с более высокими уровнями управления, к которым оно применяется.

Интересно, что программные продукты/сервисы, которые потенциально обладают наибольшей бизнес-ценностью для предприятия, наиболее трудно автоматизировать. И это понятно. Стратегическое управление — из области организационных *знаний* и *мудрости*. Как отмечено Бенсоном и Стандингом: «Мудрость и знания существуют только в умах людей. Когда люди говорят относительно знания чего-то, типа номера телефона, на самом деле они говорят о данных. Понимание того, как использовать эту часть данных — знание. Решение не вызывать кого-то в 3.00 утра — пример мудрости» [7]. Обработка и передача знаний (не будем упоминать мудрость) являются и будут оставаться, главным образом, социальным явлением, мотивационным и интуитивным, но не техническим и не предсказуемым.

1.1.3. Программная инженерия отличается от традиционной инженерии

Факт, что система ПО является компонентом информационной системы, подразумевает, что программная инженерия — лишь часть более широкой дисциплины — инженерии систем. Следовательно, инженер ПО должен понимать требования всей системы и должен быть компетентен в ее предметной области, чтобы проектировать интерфейсы, которыми ПО должно снабдить внешние устройства системы. Инженер ПО должен также понимать, что получение некоторых данных или обработку информации лучше реализовать с помощью аппаратных средств ЭВМ, чем с помощью ПО, и что некоторую обработку не нужно автоматизировать вообще.

Инженерия систем связана с изучаемыми принципами, которые определяют внутреннюю работу сложных систем. Существует длинная история инженерии систем в **традиционных** технических дисциплинах, типа проектирования механических или электрических систем. Разработанные принципы инженерии систем формализованы в математических моделях. Модели утверждаются и используются в технических изделиях. Эти изделия материальны по своей природе — мосты, строения, электростанции.

С программными продуктами дело обстоит иначе. Как выявлено в оригинальной работе Брукса [15], ПО нематериально по природе. Классические математические модели применяются к некоторым, но не ко всем аспектам ПО. ПО определено в нечетких терминах — «хороший», «плохой», «приемлемый», «удовлетворяющий требованиям пользователя» и т. д. Подобные качества используются в области обслуживания, где качество связано с нечеткими терминами типа «хорошее обслуживание», «удобство клиента», «компетентность», «знание работы» и т. д. Программная инженерия может заниматься «нечеткими» проблемами, но это не подразумевает, что они должны быть менее строгими или недоказуемыми. Очевидно, что программная инженерия должна использовать различные области математики, типа нечеткой логики или нечетких множеств, обеспечивающие строгость и доказательность.

В этом контексте следует обратить внимание, что *строгость* — это не то же самое, что *формальность*. Процесс создания и эксплуатации ПО может быть строгим даже в том случае, когда он не доказан формально в соответствии с математическими законами. Действительно, дело обстоит так даже в классической математике. Как указано Гецци и др. [34]: «Учебники по функциональному исчислению строги, но редко формализованы: доказательства теорем провсдены очень осторожно, как последовательности промежуточных выводов, которые ведут к заключительному утверждению; каждый дедуктивный шаг полагается на интуитивное оправдание, которое должно убедить читателя в его законности. И почти никогда мы не видим доказательств, выполненных формальным способом в терминах математической логики». Заинтересованных читателей отсылаем к оригинальной статье о социальных процессах и доказательствах теорем и программ Де Милло и др. [24].

Программная инженерия не должна быть бедной падчерицей традиционной инженерии. Это совсем другое. Никто в традиционной инженерии не ожидает, что мост, построенный по математическим моделям, разрушится. Точно так же «хорошее» ПО, «удовлетворяющее требованиям пользователя», не должно терпеть неудачу. Однако имеется одно «но» — все это при условии, что тем временем решительно не изменятся требования и ожидания пользователя или внешние обстоятельства.

Никто не ожидает, что мост будет перемещен на десять метров после того, как он был построен. Точно так же не следует ожидать, что программный продукт успешно выполнит различные задачи после того, как будет создан. Если это то, что нам нужно, тогда ПО создано удачно. ПО только тогда должно стать непригодным или недопустимым, когда бизнес создает новые условия или меняется внешняя среда. Если речное русло переместится на десять метров из-за недавнего наводнения, инженер-строитель не может быть обвинен, и не следует ожидать, что существующий мост можно будет легко переместить, чтобы приспособить к новому руслу.

Все сказанное означает, что разработчик ПО должен быть готов создавать ПО, которое можно приспосабливать к изменениям. Этого требует сама природа ПО. ПО должно иметь **возможность сопровождения**: понятно, ремонтопригодно и расширяемо. Это то, что отличает ПО от моста и делает программную инженерию отличной от традиционной инженерии.

Каждая система ПО уникальна, и процесс ее создания уникален. В отличие от традиционных технических дисциплин прикладной программный продукт не *производится*, он *реализуется*. Это — не автомобиль или рефрижератор. Программный продукт должен быть реализован, чтобы приспособить его к окружающей среде. Каждый случай системы ПО уникален: либо она построена на пустом месте, либо переделана из имеющегося в наличии коммерческого пакета программ (COTS — commercial off-the-shelf software — коммерческие коробочные программные продукты). Только системное ПО и инструментальные средства ПО, типа операционных систем (ОС) и текстовых процессоров, были когда-то произведены в широком масштабе. *Прикладное ПО*, которое является предметом этой книги, реализуется, а не производится.

1.1.4. Программная инженерия больше, чем программирование

В начальных страницах части 1 этой книги программная инженерия была определена как «область информатики, имеющая дело с созданием систем ПО, которые являются настолько большими или настолько сложными, что создаются коллективом или коллективами инженеров» [34]. Определение делает акцент на двух понятиях: «коллективы людей» и «сложные системы».

В **программировании** используется термин «разделение кода» — написание серий инструкций, чтобы заставить компьютер выполнить специфическую задачу. Если задача большая, для программирования может быть использован коллектив программистов, но каждый акт программирования — прежде всего персональная деятельность. Программирование — навык. Учитывая определение и спецификацию задачи, программист применяет свои навыки, чтобы выразить проблему на языке программирования.

Программная инженерия больше, чем программирование. Она обращается к сложным проблемам, которые не могут быть решены, используя одно программирование. Сложные системы должны быть разработаны прежде, чем они будут запрограммированы. Подобно строительной индустрии, над сложной системой должен поработать *архитектор*, прежде чем она будет построена. Она должна быть *разбита на модули*, используя обобщение и метод «разделяй и властвуй». Каждый модуль затем должен быть тщательно специфицирован и определены его интерфейсы к другим модулям, прежде чем его отдавать программистам для кодирования.

Программист имеет ограниченное понимание всей системы. Он/она кодирует одновременно один программный модуль — **компонент ПО**, который должен быть объединен (инженером ПО) с другими компонентами, чтобы сконфигурировать рабочую систему. (Конечно, это различие между программистом и инженером ПО приведено только для того, чтобы проиллюстрировать проблему. Практически различие может быть, а может и не быть.)

Часто инженеру ПО доступны различные **версии** одного и того же компонента. **Конфигурация** ПО выполняется объединением определенных версий различных компонентов. По этой причине можно иметь различные конфигурации одной и той же системы.

Прежде чем система будет разработана, инженер ПО должен разобраться с требованиями к ней. Это означает, что должен быть сделан и определен на некотором языке моделирования анализ требований. Стандартный язык моделирования в современной практике — UML (**Unified Modeling Language** — унифицированный язык моделирования). И анализ, и синтез моделей выполняются в UML.

Инженер ПО создает такую UML-модель системы, по которой может быть создан исходный код программы. Программисты могут начинать работу с этого момента, но инженер ПО остается ответственным за циклическое проектирование между проектом и кодом. **Циклическое проектирование** — итеративный процесс, представляющий как прямое (от проекта к коду), так и обратное (от кода к проекту) проектирование.

Наконец, программная инженерия — работа коллектива. Коллективом нужно управлять. Следовательно, программная инженерия требует **управле-**

ния проектом и воздействует на него. Это руководство включает планирование, составление бюджета и разработку графика, управление качеством и управление рисками, управление конфигурацией и изменениями.

Резюмируя все, можно сказать, что программная инженерия связана с обеспечением структурного решения системы, с проектированием структурных компонентов, с объединением компонентов в рабочую систему, с прямым и обратным проектированием, с руководством проектом и т. д. Программная инженерия — сложный процесс, в пределах которого программирование является полезным ремеслом.

1.1.5. Программная инженерия напоминает моделирование

Программная инженерия напоминает моделирование [56]. **Модели** — абстракции реального мира. Они являются абстрактными представлениями действительности. Так что же такое компьютерная программа — модель или действительность? Ну, хорошо, можно утверждать, что программа, находящаяся в памяти компьютера или напечатанная на бумаге — действительность. Однако цель программирования — не код программы сам по себе; скорее эта цель — функциональные возможности, которые он обеспечивает. Является ли футбольная игра, сыгранная на компьютере, реальностью? Что, действительно, это реальность? Ясно, что это риторические вопросы.

Абстракция — мощная технология в программной инженерии. Позволяя концентрироваться на важных аспектах проблемы и игнорировать аспекты, которые являются в настоящее время несущественными, абстракция позволяет систематически справляться со сложностью проблемы (раздел 1.1.6). Абстракция применяется также и к программным продуктам и процессу создания ПО. **Модель процесса создания ПО** является абстрактным представлением этого процесса. В практических терминах модель процесса создания ПО определяет стадии жизненного цикла и то, как они взаимодействуют (раздел 1.2). **Модель программного продукта** — это его абстрактное представление. Она определяет дискретный продукт в дискретные стадии жизненного цикла.

Модель процесса создания ПО определяет, какие программные продукты, требуемые для обеспечения стадий жизненного цикла, создавать на различных уровнях абстракции. Далее приведен список моделей базовых программных продуктов:

- **Модель требований** — сравнительно неформальная модель, которая охватывает требования пользователя и описывает систему в терминах ее бизнес-ценности.
- **Модель спецификаций** — модель, которая определяет требования к более формальному использованию терминов, применяя язык моделирования типа UML.
- **Структурная модель** — модель, которая определяет желаемую структуру системы.
- **Детальная модель проекта** — модель, которая определяет характеристики программного/аппаратного обеспечения, необходимые для реализации программирования.

- **Программная модель** — конструктивная модель, которая представляет окончательную выполнимую модель ПО.

Каждая из этих моделей программного продукта может быть разделена далее для ее детализации. Например, детальная модель проекта может включать модель пользовательского интерфейса, модель БД, модель программной логики и т. д.

Наконец, подход к программной инженерии, используемый при создании системы, влияет на абстракции моделирования. Два главных подхода — в старом стиле функциональной (процедурной, командной, структурной) разработки и в современном стиле объектно-ориентированной разработки.

Функциональный подход разбивает сложную систему до управляемых единиц, используя прием, известный как функциональная декомпозиция. Для этой цели используется технология, называемая моделированием потока данных. Модель ПО последовательно делится на процессы (при уменьшении уровня абстракции), связанные с потоками данных.

Объектно-ориентированный подход разбивает систему на пакеты/компоненты классов, связанные различными отношениями. Абстракция может применяться, формируя вложенные структуры, то есть пакет/компонент может содержать многие уровни других пакетов/компонентов. Это книжная квинтэссенция объектно-ориентированной программной инженерии.

1.1.6. Система ПО сложна

Системы ПО *сложны*. В прошлом ПО было монолитно и процедурно по своей природе. Типичная программа прошлого, написанная на КОБОЛе, была единственной сущностью, использующей подпрограммы, вызываемые по мере необходимости. Логика программы была последовательна и предсказуема. Сложность такого ПО была просто следствием его размера.

Современное объектно-ориентированное ПО является распределенным (оно может находиться на многих узлах компьютерной сети) и его выполнение случайно и непредсказуемо. Размер современного ПО — сумма размеров его компонентов. Каждый компонент разработан так, чтобы быть ограниченного управляемого размера. В результате размер не является главным фактором в сложности современного ПО.

Сложность современного ПО заключается в «проводах», то есть в связях и коммуникационных путях между компонентами. Связи между компонентами создают зависимости между распределенными компонентами, которые могут быть сложны для понимания и управления. Трудность усугубляется тем, что компоненты часто разрабатываются и управляются людьми и коллективами, даже не известными друг другу.

Рис. 1.4 представляет возможную объектно-ориентированную систему, в которой объекты различных пакетов связываются без разбора. Это создает сеть внутренней связи объектов. В диаграмме сложность в пределах отдельных пакетов (компонентов) все еще управляема из-за ограниченного размера пакетов. Однако зависимости, созданные связями между пакетами, будут расти по экспоненте с добавлением новых пакетов. Кто должен обеспечивать управление такими зависимостями, не всегда ясно, поскольку обязанности обеспечения управления для пакетов остаются за различными коллективами.

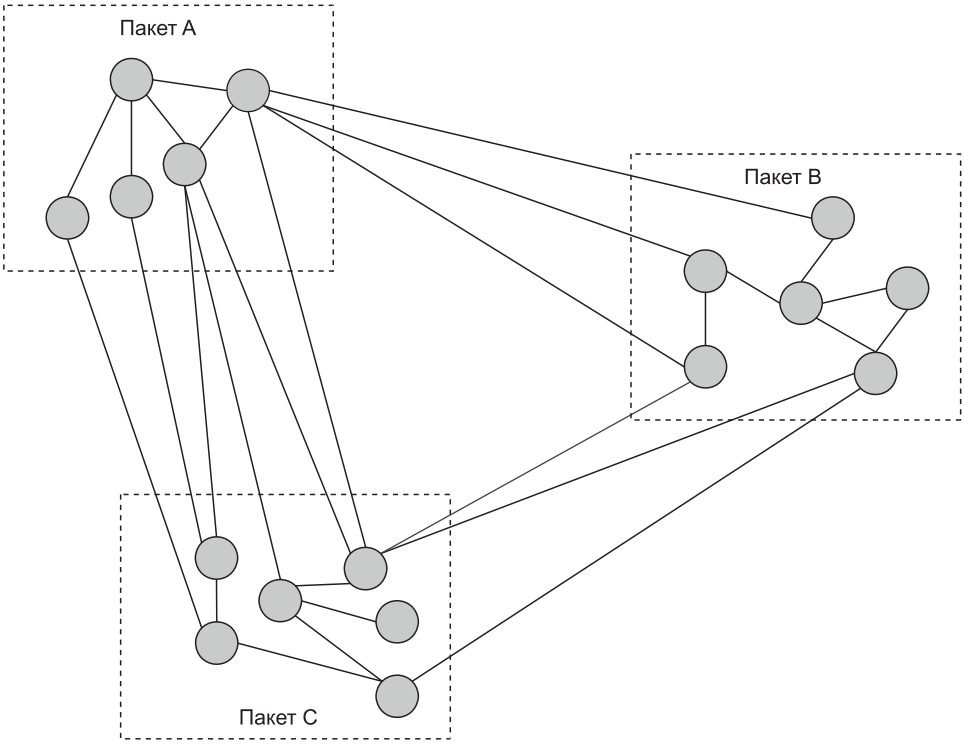


Рис. 1.4. Сложность в «проводах»

Более важен тот факт, что любой объект в одном пакете может связываться с любым объектом в другом пакете; это создает *потенциальные зависимости* между всеми объектами в системе. Такой факт означает, что изменение какого-то объекта может потенциально повлиять (вызвать «*эффект ряби*») на любой другой объект в системе.

Более формально: совокупной мерой зависимости объектов с неограниченными межпакетными (межкомпонентными) коммуникационными связями является число различных комбинаций пар объектов. Такая мера может быть вычислена, используя метод теории вероятностей, известный как *правило комбинаторики*. Ниже приводится формула для вычисления совокупной зависимости класса (`CCD` — cumulative class dependency) в системе с `n` классами объектов. Для 5 классов `CCD` равняется 10. Для 57 классов (например) CCD равняется 1596. Такой рост сложности быстро становится *неприемлемым*.

$$_{n}CCD_{2} = \frac{n!}{2!(n-2)!}$$

Формула вычисляет наихудшую сложность, когда каждый объект связывается со всеми другими объектами. Хотя самый плохой вариант практически маловероятен, он должен быть принят при любом анализе влияния зависимости, проводимом в системе (просто потому, что реальные зависимости заранее

не известны). Если изменение в классе может потенциально воздействовать на любой другой класс, то этот факт должен быть проверен для гарантии, что изменение не вызовет неприятных последствий. Системы, допускающие произвольную сеть связи объектов, подобно изображенной на рис. 1.4, считаются неприемлемыми с точки зрения возможности программной инженерии. Они непонятны, неремонтопригодны и нерасширяемы.

Решение дилеммы находится в замене сетей объектов иерархиями (древовидными структурами) объектов. Все приемлемые сложные системы имеют *иерархическую* форму. Эта тема настолько важна, что ей посвящена отдельная глава (глава 9). На рис. 1.5 просто показано, как можно уменьшить сложность системы, допуская только единственный канал связи между двумя пакетами. Каждый пакет определяет интерфейсный объект (это может быть интерфейс Java-стиля или так называемый доминантный класс), через который осуществляется вся связь с пакетом. Несмотря на добавление трех дополнительных объектов, сложность системы, изображенной на рис. 1.5, явно уменьшена по сравнению с той же самой системой, изображенной на рис. 1.4.

Обратите также внимание на то, что проект на рис. 1.4 выбивает из рук инженера ПО основной инструмент — механизм **абстракции**. Абстракция позволяет нам рассуждать относительно отобранных частей, не учитывая несущественные детали (абстрагируясь от них). Хотя объекты на рис. 1.4 сгруп-

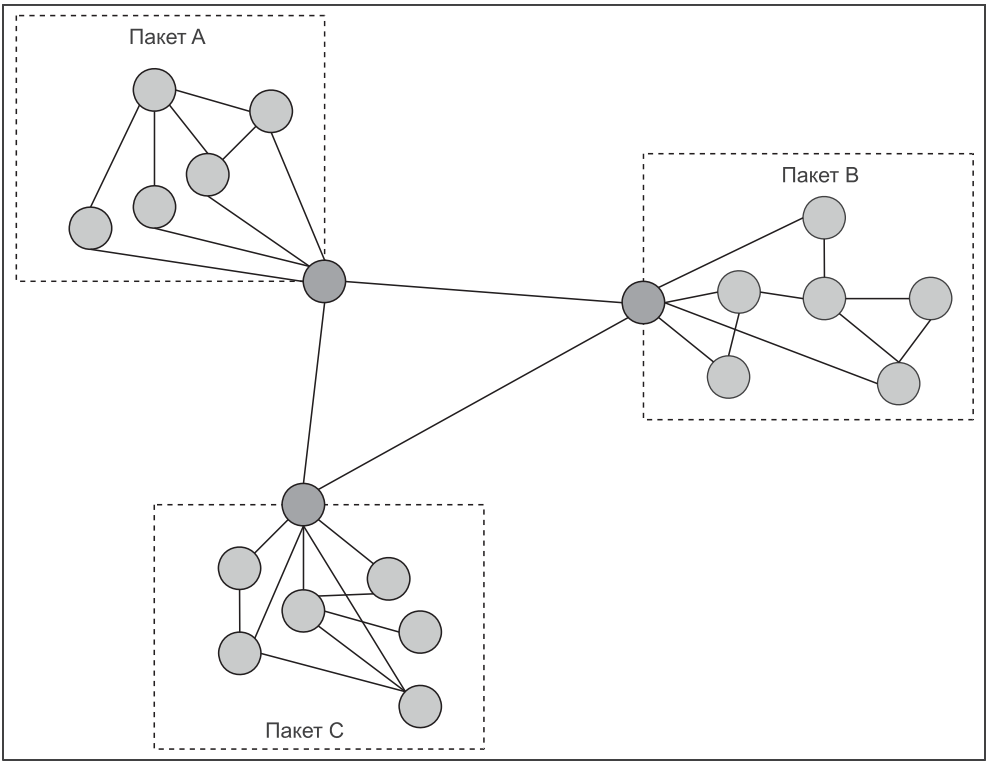


Рис. 1.5. Понижение сложности добавлением интерфейсов между пакетами

пированы в три пакета, сложность системы (измеренная как совокупная зависимость классов) та же самая, как и в подобной системе совсем без пакетов. Пакеты на рис. 1.4 не дают никакого полезного уровня абстракции. Они могут вообще отсутствовать.

1.2. Стадии жизненного цикла

Жизненный цикл ПО — это абстрактное представление процесса создания и эксплуатации ПО. Он определяет для проекта разработки ПО стадии, шаги, действия, методы, инструменты, а также то, что ожидается сдавать. Все это определяет стратегию разработки ПО.

Существует ряд полезных моделей жизненного цикла (раздел 1.3), которые находятся друг с другом в общем согласии по стадиям жизненного цикла, но отличаются важностью отдельных стадий и взаимодействиями между ними. Стадии жизненного цикла, принятые в этой книге, были определены в начале этой главы. Они следующие:

1. анализ требований;
2. проектирование системы;
3. реализация;
4. интеграция и внедрение;
5. процесс функционирования и сопровождения.

1.2.1. Анализ требований

«**Требования пользователя** — это утверждения на естественном языке плюс диаграммы, содержащие сведения, какие услуги ожидаются от системы, и ограничения, при которых система должна работать» [96]. **Анализ требований** включает действия по их определению и составлению их списка. В современной практике анализу требований помогает хорошая степень технической строгости, и поэтому эти требования иногда отождествляют с **техническими требованиями**.

Определение требований, оказывается, одна из самых больших проблем любого жизненного цикла разработки ПО. Пользователям часто неясно, что они требуют от системы. Часто они не знают реальные требования, преувеличивают их, предъявляют требования, которые противоречат требованиям коллег и т. д. Имеется также риск, как и в любом общении между людьми, что истинное значение требования будет неправильно истолковано. Разработчики часто сталкиваются с подобными анонимными заявлениями: «Я знаю, что Вы полагаете, что Вы поняли то, что Вы думаете, что я сказал, но я не уверен, что Вы сделали то, что Вы слышали, и не то, что я подразумевал».

Имеется много методов и технологий выявления требований. Они включают [64]:

- интервьюирование пользователей и экспертов в предметной области;
- анкетные опросы пользователей;
- наблюдение, как пользователи выполняют свои задачи;
- изучение существующих документов системы;

- изучение подобных систем ПО, чтобы выяснить состояние в предметной области;
- изучение опытных образцов рабочих моделей для определения и подтверждения требований;
- объединенные совещания разработчиков и клиентов по разработке приложения.

Спецификация требований следует за выявлением требований. В настоящее время UML — стандартный язык моделирования для спецификации требований (так же как и для проектирования системы). Требования определяются как в графических моделях, так и с помощью текстовых описаний. Поскольку сложную систему нельзя понять с единственной точки зрения, модели снабжаются дополнениями и при необходимости перекрестными точками зрения на систему.

И графические представления, и текст размещаются в хранилище специального CASE-средства (**computer assisted software engineering** — автоматизированная программная инженерия). Данное средство облегчает изменения в моделях, если это потребуется. Оно позволяет интегрировать различные модели с перекрывающимися концепциями. CASE-средство позволяет также выполнять преобразования между моделями анализа (где это возможно) и помогает в преобразованиях моделей проектирования.

Анализ требований завершается созданием **технического задания** [64]. Большинство организаций использует некоторые шаблоны для технического задания. Шаблон определяет структуру документа и дает руководящие принципы того, как его писать. Основная часть технического задания содержит модели и описания сервисов и ограничений системы. *Сервисы системы* (то, что система должна делать) часто делятся на функциональные требования и требования к данным. *Ограничения системы* (то, чем система ограничена) включают соображения, связанные с пользовательским интерфейсом, работой, безопасностью, эксплуатационными условиями, политическими и юридическими ограничениями и т. д.

Как уже мимоходом упоминалось, результат каждой стадии жизненного цикла должен быть утвержден и проверен. Профессиональный подход к тестированию требует внутри организации создания группы по гарантии качества ПО (SQA — **software quality assurance**). Коллектив этой группы состоит из профессиональных системных испытателей. Он работает достаточно независимо от разработчиков. Чтобы обеспечить целиком всю работу процесса, именно коллектив SQA-группы, а не разработчики, является ответственным за качество программного продукта (то есть SQA отвечает за все невыявленные несоответствия и дефекты в ПО).

Тестирование абстрактных моделей затруднено, поскольку большую часть времени оно не может быть автоматизировано. *Сквозной контроль* и *инспекции* — вот две популярные и эффективные технологии. Данные технологии схожи. Это предварительные встречи разработчиков и пользователей, на которых «проходятся» по техническому заданию и документам. Обсуждение, которое происходит на встречах, вероятно, раскроет некоторые проблемы. Сущность этих технологий заключается в том, что в течение встреч идентифициру-

ются проблемы, но их решение не определено, и нет никакого «указующего перста» на людей, потенциально ответственных за эти проблемы.

1.2.2. Проектирование системы

«**Проектирование ПО** — это описание структуры ПО, которое будет реализовано, данных, которые являются частью системы, интерфейсов между компонентами системы и, иногда, используемых алгоритмов» [96]. Это определение совместимо с определением системы ПО как объединения структур данных и алгоритмов. В информационных системах предприятий структуры данных подразумевают БД. Алгоритмы не всегда полностью описываются во время проектирования, чтобы оставить некоторый уровень свободы выполнения программистам (ведь говоря прямо, проектировщики — это не программисты, и они не в состоянии выбрать умные алгоритмические решения).

Проектирование начинается там, где заканчивается анализ. Столь же истинно и тривиально утверждение, что линия, отделяющая анализ от проектирования, во многих проектах не столь уж и ясна. Теоретически проблема проста. Анализ — моделирование, не ограниченное никакой реализацией (аппаратного/программного обеспечения). Проектирование — моделирование, которое учитывает платформу, на которой должна быть реализована система.

На практике различие между анализом и проектированием размыто. Этому имеются две главные причины. Во-первых, современные модели жизненного цикла являются *итеративными* и *пошаговыми* (раздел 1.3.2). В большинстве таких моделей при разработке в любой момент времени имеются многочисленные разнообразные версии ПО. Некоторые из версий находятся в процессе анализа, другие — в процессе проектирования; некоторые в разработке, другие в производстве и т. д. Во-вторых, и это более важно, для анализа и проектирования используется один и тот же язык моделирования (UML). Переход от анализа к проектированию «по готовности» предпочтительней, чем переход между различными представлениями. Модель анализа уточняется в модели проекта простым дополнением деталей спецификации. Провести линию раздела между анализом и проектированием в таких обстоятельствах очень трудно.

Проектирование, обсужденное выше, более точно называется **детальным проектированием**, то есть проектированием, которое добавляет детали к моделям анализа. Но имеется другой аспект проектирования системы, а именно структурное проектирование. **Структурное проектирование** связано с определением структуры системы, которая должна быть детально спроектирована и которой следует твердо придерживаться, а также с принципами и образцами внутренних коммуникаций между компонентами.

Структурное проектирование задает «красоту» системы. Главная цель структурного проектирования состоит в том, чтобы получить систему, которая является *приемлемой* — понятной, ремонтопригодной и расширяемой. Детальный проект должен соответствовать структурному проекту. Из-за расплывчатой линии раздела между анализом и детальным проектированием некоторые ранние структурные решения, может быть, придется заново выбрать внутри технического задания или даже раньше (но после определения требований).

Тестирование структурного проекта двулико. Во-первых, преимущества структуры, предложенной проектировщикам, должны быть продемонстрированы. Нужно показать, что структура поддерживает сложность ПО, гарантирует возможность сопровождения, упрощает разработку и т. д. Во-вторых, тестирование структурного проекта должно подтвердить, что проект компонентов соответствует принципам и шаблонам принятой структуры.

Тестирование детального проекта также имеет два аспекта. Во-первых, чтобы быть тестируемым, должна быть возможность трассировать детальный проект. **Управление трассировкой** — целая отрасль программной инженерии, занимающаяся поддержанием связей между продуктами ПО в различных стадиях разработки. В случае детального проекта каждый продукт проекта должен быть связан с требованиями в техническом задании, которое мотивировало производство того продукта. Наличие продукта еще не подразумевает, что это требование обеспечено. Следовательно, второй аспект тестирования проекта использует сквозной контроль и инспекции, чтобы оценить качество изделия проекта.

1.2.3. Реализация

Реализация в большей мере связана с программированием. Но программирование подразумевает не только группу людей, сидящих в общем помещении и кодирующих на некотором языке программирования в соответствии со спецификацией проекта. Программирование предполагает намного более интеллектуальные требования, чем это. Как указано в предыдущих разделах, проекты будут «недоопределены» в некоторых областях, когда они попадают к программистам, особенно в области проектирования алгоритмов. Завершение спецификаций требует дополнительного проектирования прежде, чем можно будет начать кодирование. В этом смысле программист — тоже проектировщик.

Программист — *инженер, имеющий дело с компонентами*. Сегодняшнее программирование редко выполняется на пустом месте. Большая часть программирования основана на многократном использовании уже созданных компонентов. Это означает, что программист должен иметь знание о компонентах ПО и должен знать, как найти это ПО, чтобы добавить к нему новые закодированные компоненты приложения. Это трудный вопрос.

Программист — *инженер, работающий в двух направлениях*. Программирование начинается с преобразования проекта в код. Начальный код не должен программироваться вручную. Используя CASE-средства и IDE-средства (**integrated development environments** — интегрированные средства разработки), код начинает формироваться (прямое проектирование) из моделей проекта. После того как эта работа будет сделана, произведенный код должен быть скорректирован вручную, чтобы заполнить отсутствующие части (эти «части» существенны и наиболее трудны в программировании). После того как код будет модифицирован программистом, он может обратно воздействовать на модели проекта, корректируя их. Эти технические операции в прямом и обратном направлении называются **циклическим проектированием**.

Если все это звучит просто, на самом деле это не так. Циклическое проектирование несовершенно. Существующие инструментальные средства мощны и умны, но они все еще не приспособлены должным образом для выполнения некоторых видов работы. Чтобы сохранить проект и реализацию синхронными, и проектировщики, и программисты должны знать ограничения и выполнять ручные исправления и дополнения, когда это необходимо. Большая часть ответственности в этой задаче падает на менеджеров проекта. Они должны планировать и контролировать работу проектировщиков и программистов так, чтобы те не наступали на ноги друг другу. Практическое требование заключается в том, чтобы прямое и обратное проектирование не наложились по времени [62].

Во многих проектах реализация — самая длинная из стадий разработки. В некоторых моделях жизненного цикла, типа быстрой разработки ПО (раздел 1.3.2), реализация является доминирующей стадией разработки. Реализация — подверженная ошибкам деятельность. Время, потраченное на творческое написание программ, может быть меньше, чем время, потраченное на отладку программы и ее тестирование.

Отладка — это процесс удаления из ПО «блох» — ошибок в программах. Ошибки в синтаксисе программы и некоторые логические ошибки могут быть определены и исправлены коммерческими средствами отладки. Другие ошибки и дефекты должны быть обнаружены во время тестирования программы. **Тестирование** может иметь форму просмотра кода (сквозной контроль и инспекции) или может основываться на выполнении программы (наблюдение за поведением программы во время ее выполнения). Управление трассировкой поддерживает возможность использования тестирования, если программы удовлетворяют требованиям пользователя.

Имеются два вида *тестирования, основанного на выполнении программы*: **тестирование на основе технических требований** (тестирование черного ящика) и **тестирование на основе кода** (тестирование белого ящика). Оба вида используют ту же самую стратегию задания программе входных данных и наблюдения, тот ли выходной результат получается, который ожидался. Различие заключается в том, что при тестировании на основе технических требований программе задаются данные без какого-либо учета логики работы программы. Считается, что программа должна вести себя разумно при любых входных данных. В тестировании на основе кода используются такие входные данные, которые позволяют проверить определенные пути выполнения программы — столько путей и настолько разнообразных, насколько это возможно. Поскольку тестирование на основе технических требований и тестирование на основе кода обнаруживают различные виды ошибок и дефектов, нужно использовать их оба.

1.2.4. Интеграция и внедрение

«Целое — больше чем сумма его частей». Этот оригинальный афоризм Аристотеля (384–322 гг. до н. э.) охватывает сущность интеграции системы и ее внедрения. **Интеграция** собирает приложение из набора компонентов, предварительно созданных и проверенных. **Внедрение** — передача системы клиентам для использования в производстве.

Интеграция ПО означает переход от «программирования в малом» к «программированию в большом» [34]. Информационные системы предприятий — все достаточно большие и сложные системы (раздел 1.1.6) и для них интеграция — существенная стадия в жизненном цикле. По этому поводу есть другое высказывание: «Для каждой сложной проблемы имеется простое решение, которое не будет работать» (Х. Л. Менкен). Интеграция не может быть проигнорирована. Это отдельная стадия по праву, даже если ее иногда трудно отделить от реализации, как, например, в случае *непрерывной интеграции* в быстрой разработке (раздел 1.3.2).

Интеграцию также трудно отделить от тестирования. Фактически, стадия интеграции жизненного цикла часто упоминается и обсуждается под термином **тестирования интеграции**. При широком использовании итеративных моделей жизненного цикла (раздел 1.3.2) ПО создается как последовательность быстрых пошаговых реализаций. Каждый шаг — интеграция компонентов, до этого проверенных индивидуально, однако при этом до внедрения саму эту интеграцию системы необходимо сначала проверить.

В значительной степени интеграция определяется структурным проектом системы. В свою очередь, структура системы определяет ее компоненты и зависимости между ними. Особенно важно, чтобы структурное решение было в виде *иерархии* или древовидной структуры (раздел 1.1.6). Иерархия (древовидная структура) означает устранение любых циклических зависимостей между компонентами. В случае циклических зависимостей тестирование интеграции отдельных шагов создания ПО (конструкций) может оказаться невозможным.

Рассмотрим структуру зависимых компонентов, где компонент C_i использует (зависит от) компонент C_j, а C_j использует компонент C_k. Предположим, что C_i и C_j уже были реализованы и индивидуально протестированы, а компонент C_k должен быть еще создан. Задача состоит в том, чтобы объединить C_i и C_j. Эта задача требует программирования **испытательной заглушки** для C_k, то есть части кода, которая моделирует поведение отсутствующего компонента C_k. Заглушка обеспечивает среду интеграции для выполнения компонента C_j. Обычный путь создания заглушки — сделать ее такой, чтобы она обеспечивала те же зависимости входа/выхода, что и в окончательном компоненте C_k, и формировать результаты, ожидаемые для компонента C_j, жестким кодированием их или читая их из файла.

Все это работает, если только между C_j и C_k нет никаких циклических зависимостей. В присутствии циклических зависимостей оба компонента должны быть полностью реализованы и индивидуально проверены до интеграции. Но даже и тогда циклические зависимости создадут кошмар тестирования. С большими циклами в структурном проекте тестирование интеграции должно быть выполнено как единая операция, называемая *тестированием «одним махом»*. Тестирование «одним махом» может быть успешно выполнено только для небольших программных решений. Это не очень разумно в современной разработке ПО.

Возвращаясь к примеру, предположим теперь, что C_j и C_k были реализованы и индивидуально протестированы, но C_i должен быть еще разработан. Как мы должны интегрировать C_j и C_k, чтобы объединенный экземпляр