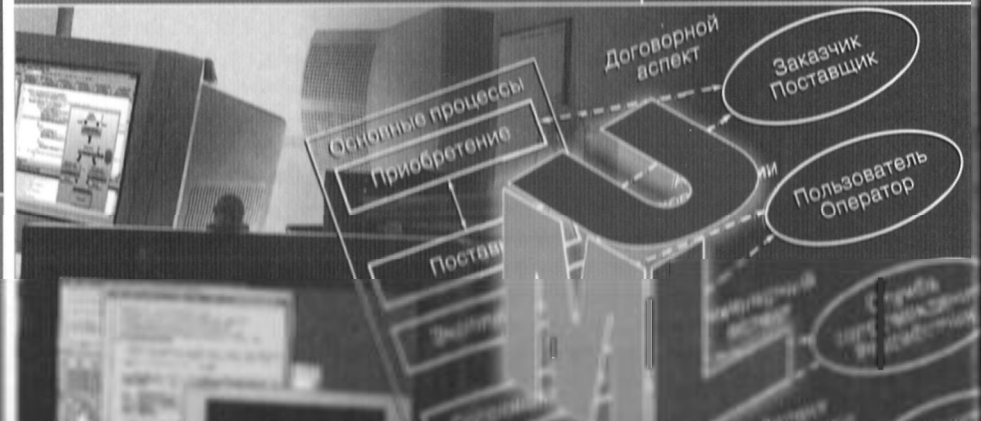
А. В. Рудаков

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНЫХ ПРОДУКТОВ

2-е издание

ИНФОРМАТИКА
И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

ACADEMA



СРЕДНЕЕ ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАНИЕ

СРЕДНЕЕ ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАНИЕ

А. В. РУДАКОВ

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНЫХ ПРОДУКТОВ

Допущено
Министерством образования Российской Федерации
в качестве учебного пособия для студентов
образовательных учреждений среднего профессионального образования

2-е издание, стереотипное

ACADEMA

Москва
Издательский центр «Академия»
2006

УДК 681.3.06(075.32)
ББК 32.973-018я723
Р83

Рецензенты:
начальник научно-исследовательского отдела ОАО «Научные приборы» *О. Е. Славянский*;
зам. директора по учебно-методической работе Санкт-Петербургской школы электроники (колледж) *Н. А. Васильева*

Рудаков А. В.

Р83 Технология разработки программных продуктов : учеб. пособие для студ. сред. проф. образования / А. В. Рудаков. — 2-е изд., стер. — М. : Издательский центр «Академия», 2006 — 208 с.

ISBN 5-7695-3281-5

Рассмотрены история возникновения, современное состояние, принципы организации, основные положения и перспективы развития технологии разработки программных продуктов.

Для студентов образовательных учреждений среднего профессионального образования.

УДК 681.3.06(075.32)
ББК 32.973-018я723

ISBN 5-7695-3281-5

ПРЕДИСЛОВИЕ

В данном учебном пособии систематизирован накопленный автором опыт по преподаванию одноименной дисциплины в Петровском колледже Санкт-Петербурга.

При отборе материала автор руководствовался Государственным образовательным стандартом среднего профессионального образования по специальности «Программное обеспечение вычислительной техники и автоматизированных систем» и личным опытом по разработке различных программных продуктов.

Учебное пособие состоит из 14 глав. В главе 1 дано описание жизненного цикла программного продукта в соответствии с принятыми международными стандартами. Рассмотрены основные процессы жизненного цикла программного продукта и взаимосвязи между ними.

Глава 2 посвящена рассмотрению основных этапов работ по созданию программного продукта. Дано краткое описание каждого этапа.

В главе 3 рассмотрены основные модели жизненного цикла разработки программного продукта. Даны их сравнительные характеристики и описаны поддерживающие процессы.

Глава 4 содержит сведения об организации процесса разработки программного продукта. В ней рассмотрены история вопроса, рекомендации по организации процесса в соответствии с моделью CMM-SEI, вопросы, связанные с управлением качеством разработки в соответствии со стандартом ISO 9001.

Глава 5 посвящена вопросам сбора, анализа и использования метрик программного продукта, проекта и процесса. Метрики в программных проектах призваны оказать помощь руководителям разного уровня, а также самим разработчикам в принятии своевременных и правильных решений.

В главе 6 рассмотрены вопросы, связанные с планированием работ по созданию программного продукта. Дано общее представление об оценке объема и сложности разрабатываемого программного продукта, технических, нетехнических и финансовых ресурсов, а также возможных рисков, связанных с ними.

Глава 7 посвящена управлению требованиями к разрабатываемому программному продукту.

В главе 8 рассмотрены вопросы проектирования разрабатываемого программного продукта. Приведены основные положения структурного и объектно-ориентированного проектирования.

Глава 9 посвящена разработке (кодированию) программного продукта. В ней также рассмотрены вопросы, связанные с разработкой документации на программный продукт, созданием справочной системы и различных версий программного продукта и его инсталляций.

В главе 10 рассмотрены вопросы, связанные с тестированием программного продукта. В ней описаны виды тестирования, типы программных ошибок, а также рекомендации по разработке тестов и тестированию программных продуктов.

Главы 11 и 12 посвящены соответственно сопровождению разработанного программного продукта и управлению поставками программного продукта.

В главе 13 рассмотрены вопросы, касающиеся надежности программных продуктов. Надежность считается ключевым показателем качества программного продукта. Особое внимание обеспечению надежности программных продуктов обусловлено тем, что этот показатель имеет определяющее значение для конечного пользователя, а факторы качества, связанные с изменением программного продукта и разработкой его новых версий, имеют определяющее значение для разработчиков программного продукта и групп технической поддержки.

Глава 14 содержит сведения о назначении и основных компонентах языка моделирования UML.

ВВЕДЕНИЕ

Современное общество невозможно представить без компьютера. Они настолько широко и глубоко внедрились в нашу жизнь, что очень трудно назвать какую-либо сферу деятельности человека, где бы они не использовались. В связи с этим серьезные требования предъявляются и к аппаратной части современных компьютеров, и к используемому программному обеспечению. В основном именно программное обеспечение, или, иными словами, программные продукты, обеспечивают возможность широкого использования компьютеров. Стоит нам переустановить программное обеспечение компьютера или добавить какой-либо новый программный продукт, и мы сможем решать на этом компьютере совершенно новые задачи. Следовательно, используемые программные продукты должны соответствовать определенным критериям, обеспечивающим надежность работы компьютера и удобство работы пользователя.

Если аппаратура компьютера, даже самые простейшие ее компоненты, с самого начала разрабатывались и выпускались в соответствии с установленным технологическим процессом, то какой-то определенной технологии разработки программных продуктов первое время не существовало. Разработчики опирались в основном на свой личный опыт, используя кустарные способы разработки. Такой подход не мог не отразиться на качестве разрабатываемых программных продуктов, сроках их разработки и, следовательно, на их стоимости. Данная ситуация была названа кризисом программирования.

Чтобы выйти из кризиса, необходимо было создать индустриальные способы разработки программных продуктов, т.е. технологию их разработки, которая включала бы в себя различные передовые инженерные методы и средства создания программных продуктов. В дальнейшем эти методы и средства были объединены общим понятием «программная инженерия» (software engineering).

Создание указанной технологии в совокупности с системой оценки ее использования при разработке программных продуктов позволило повысить надежность программных продуктов и качество их разработки, а также облегчило заказчикам выбор организации для разработки необходимого им программного продукта.

ГЛАВА 1

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ПРОДУКТА

1.1. Понятие жизненного цикла программного продукта

Программный продукт (ПП) представляет собой набор компьютерных программ, процедур и связанной с ними документации и данных.

Жизненный цикл программного продукта — это период времени, начинающийся с момента принятия решения о необходимости создания ПП и заканчивающийся в момент его полного изъятия из эксплуатации.

Структуру жизненного цикла ПП, состав процессов, действия и задачи, которые должны быть выполнены во время создания ПП, определяет и регламентирует международный стандарт ISO/IEC 12207: 1995 «Information Technology — Software Life Cycle Processes» (ISO — International Organization for Standardization — Международная организация по стандартизации; IEC — International Electrotechnical Commission — Международная комиссия по электротехнике; название стандарта «Информационные технологии — Процессы жизненного цикла программ»).

Под *процессом* понимают совокупность взаимосвязанных действий, преобразующих входные данные в выходные. Каждый процесс характеризуется определенными задачами и методами их решения, а также исходными данными, полученными от других процессов, и результатами.

Каждый процесс разделен на набор действий, каждое действие — на набор задач. Запуск и выполнение процесса, действия или задачи осуществляются другими процессами.

В России, начиная с 1970-х годов, создание ПП регламентировалось стандартами ЕСПД (Единая система программной документации — серия ГОСТ 19.XXX), которые были ориентированы на класс относительно простых программ небольшого объема, создаваемых отдельными программистами. В настоящее время указанные стандарты устарели концептуально и по форме, их сроки действия закончились и дальнейшее использование этих стандартов нецелесообразно. В результате для каждого серьезного проекта приходится создавать комплекты нормативных и методических документов, регламентирующих процессы создания конкретного прикладного ПП, поэтому в отечественных разработках целесообразно использовать современные международные стандарты.

В соответствии со стандартом ISO/IEC 12207 все процессы жизненного цикла ПП разделены на три базовые группы:

основные процессы;

вспомогательные (поддерживающие) процессы;

организационные процессы.

1.2. Основные процессы жизненного цикла программного продукта

Основные процессы включают в себя набор определенных действий и связанных с ними задач, которые должны быть выполнены в течение жизненного цикла ПП.

К основным относятся процессы приобретения, поставки, разработки, эксплуатации и сопровождения.

Процесс приобретения (***acquisition process***) охватывает действия заказчика по приобретению ПП. К этим действиям относятся:

инициирование приобретения;

подготовка заявочных предложений;

подготовка и корректировка договора;

надзор за деятельностью поставщика;

приемка и завершение работ.

Инициирование приобретения включает в себя следующие задачи:

определение заказчиком своих потребностей в приобретении, разработке или усовершенствовании системы, ПП или услуг;

анализ требований к системе;

принятие решения относительно приобретения, разработки или усовершенствования существующего ПП;

проверку наличия необходимой документации, гарантий, сертификатов, лицензий и поддержки в случае приобретения ПП;

подготовку и утверждение плана приобретения, включающего в себя требования к системе, тип договора, ответственность сторон и т.д.

Согласно нормативным документам понятие «система» можно интерпретировать двояко. В одном случае под системой понимают совокупность аппаратных, программных, материальных и людских ресурсов, услуг и данных, одним словом, все то, что потребует разработки или покупки.

В другом случае система — это совокупность конечных продуктов, которые будут действовать совместно, и вспомогательных продуктов, необходимых для разработки, поставки, обучения и т.д.

Подготовка заявочных предложений подразумевает разработку и составление предложений, которые должны содержать:

требования к разрабатываемой или покупаемой системе;

перечень необходимых ПП;
условия и соглашения;
технические ограничения (например, указание конкретной среды функционирования системы).

Заявочные предложения направляются выбранному поставщику (или нескольким поставщикам в случае проведения тендера). Поставщиком является организация, которая заключает договор с заказчиком на поставку системы, ПП или программной услуги на условиях, оговоренных в договоре.

Подготовка и корректировка договора включают в себя следующие задачи:
определение заказчиком процедуры выбора поставщика, содержащей критерии оценки предложений возможных поставщиков;
выбор конкретного поставщика на основе анализа предложений;
подготовку и заключение договора с поставщиком;
внесение изменений (при необходимости) в договор в процессе его выполнения.

Надзор за деятельностью поставщика осуществляется в соответствии с действиями, предусмотренными в процессах совместной оценки и аудита (см. подразд. 1.3).

В процессе *приемки* подготавливаются и выполняются необходимые тесты. *Завершение работ* по договору осуществляется в случае удовлетворения всем условиям приемки.

Процесс поставки (***supply process***) охватывает действия и задачи поставщика при снабжении заказчика ПП или услугой. К этим действиям относятся:
инициирование поставки;
подготовка ответа на заявочные предложения;
подготовка договора;
планирование;
выполнение и контроль;
проверка и оценка;
поставка и завершение работ.

Инициирование поставки заключается в рассмотрении поставщиком заявочных предложений и принятии решения согласиться с выставленными требованиями и условиями или предложить свои.

Подготовка ответа на заявочные предложения выполняется в соответствии с принятыми решениями в результате инициирования поставки.

Подготовка договора осуществляется после выбора заказчиком конкретного поставщика.

Планирование выполняется после заключения договора и включает в себя следующие задачи:
принятие решения поставщиком относительно выполнения работ своими силами или с привлечением субподрядчика;

разработку поставщиком плана управления проектом, содержащего организационную структуру проекта, разграничение ответственности, технические требования к среде разработки и ресурсам, управление субподрядчиками и т.д.

Субподрядчик — это организация, индивидуум или корпорация, заключившие договор с поставщиком на исполнение части работ, которые поставщик должен выполнить по договору с заказчиком.

Выполнение и контроль включают в себя задачи, связанные с выполнением поставщиком взятых на себя обязательств по поставке, разработке или усовершенствованию системы, ПП или услуг и контролем за этим выполнением.

Проверка и оценка выполняются в соответствии с действиями, предусмотренными в процессах совместной оценки и аудита (см. подразд. 1.3).

Поставка и завершение работ выполняются в соответствии с оговоренными в процессе инициирования действиями по приемке и завершению работ.

Процесс разработки (***development process***) охватывает действия и задачи разработчика и предусматривает следующие основные направления работ:
создание ПП и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации;
подготовку материалов, необходимых для проверки работоспособности и качества ПП;
подготовку материалов, необходимых для организации обучения персонала, и т.д.

Более подробно процесс разработки рассмотрен в гл. 9.

Процесс эксплуатации (***operation process***) охватывает действия и задачи оператора — организации, занимающейся эксплуатацией разработанного ПП или системы. К этим действиям относятся:
подготовительная работа;
эксплуатационное тестирование;
эксплуатация системы;
поддержка пользователей.

Подготовительная работа предполагает выполнение оператором следующих задач:
планирование работ, выполняемых в процессе эксплуатации, и установку эксплуатационных стандартов;
определение процедур локализации и разрешения проблем, возникающих в процессе эксплуатации.

Эксплуатационное тестирование выполняется для каждой очередной версии ПП, после чего она передается в эксплуатацию.

Эксплуатация системы осуществляется в предназначенной для этого среде в соответствии с пользовательской документацией.

Поддержка пользователей заключается в оказании помощи и консультациях при обнаружении ошибок в процессе эксплуатации ПП.

Процесс сопровождения (***maintenance process***) охватывает действия и задачи сопровождающей организации (службы сопровождения). Данный процесс активизируется при изменениях (модификациях) ПП и соответствующей документации, вызванных возникшими проблемами или потребностями в модернизации либо адаптации ПП. В соответствии со стандартом IEEE-90 (IEEE — Institute of Electrical and Electronics Engineers — Институт инженеров по электротехнике и электронике) под сопровождением понимается внесение изменений в ПП в целях исправления ошибок, повышения производительности либо адаптации к изменившимся условиям работы или требованиям. Более подробно процесс сопровождения рассмотрен в гл. 11.

1.3. Вспомогательные (поддерживающие) процессы жизненного цикла программного продукта

Основной целью вспомогательных (поддерживающих) процессов является создание надежного, полностью удовлетворяющего требованиям заказчика ПП в установленные договором сроки. К вспомогательным относятся процессы документирования, управления конфигурацией, обеспечения качества, верификации, аттестации, совместной оценки, аудита, разрешения проблем.

Процесс документирования (***documentation process***) предусматривает формализованное описание информации, созданной в течение жизненного цикла ПП. Данный процесс состоит из набора действий, с помощью которых планируют, проектируют, разрабатывают, выпускают, редактируют, распространяют и сопровождают документы, необходимые для всех заинтересованных лиц, таких как руководство, технические специалисты и пользователи системы.

Процесс документирования включает в себя:

подготовительную работу;

проектирование и разработку документации;

выпуск документации;

сопровождение.

Подготовительная работа требуется для определения и согласования необходимого перечня документов и документируемых процедур, выполняемых в процессе жизненного цикла ПП.

Проектирование и разработка документации выполняются в процессе работы над ПП и завершаются одновременно с завершением его жизненного цикла.

Выпуск документации осуществляется по мере ее готовности в ходе разработки ПП и его дальнейшего сопровождения.

Сопровождение включает в себя действия по корректировке и обновлению документации в процессе жизненного цикла ПП.

Процесс управления конфигурацией (***configuration management process***) предполагает применение административных и технических процедур на всем протяжении жизненного цикла ПП для выполнения следующих действий:

определение состояния компонентов ПП;

управление модификациями ПП;

описание и подготовка отчетов о состоянии компонентов ПП и запросов на модификацию;

обеспечение полноты, совместимости и корректности компонентов ПП;

управление хранением и поставкой ПП.

Согласно стандарту IEEE-90 под *конфигурацией программного продукта* понимается совокупность его функциональных и физических характеристик, установленных в технической документации и реализованных в ПП.

Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПП на всех стадиях жизненного цикла. Общие принципы и рекомендации по управлению конфигурацией ПП отражены в стандарте ISO/IEC CD 12207-2: 1995 «Information Technology — Software Life Cycle Processes. Part 2. Configuration Management for Software» («Информационные технологии — Процессы жизненного цикла программ. Часть 2. Управление конфигурацией программ»).

Процесс управления конфигурацией включает в себя:

подготовительную работу;

идентификацию конфигурации;

контроль конфигурации;

учет состояния конфигурации;

оценку конфигурации;

управление выпуском и поставку.

Подготовительная работа заключается в планировании управления конфигурацией.

Идентификация конфигурации устанавливает правила, с помощью которых можно однозначно идентифицировать и различать компоненты ПП и их версии. Кроме того, каждому компоненту и его версиям соответствует однозначно обозначаемый комплект документации. В результате создается база для однозначного выбора версий компонентов ПП и манипулирования ими, которая использует ограниченную и упорядоченную систему символов, идентифицирующих различные версии программного продукта.

Контроль конфигурации предназначен для систематической оценки предполагаемых модификаций ПП и координированной их реализации с учетом эффективности каждой модификации и затрат на ее выполнение. При этом обеспечивается контроль состояния и

развития компонентов ПП и их версий, а также адекватность реально изменяющихся компонентов и их комплектной документации.

Учет состояния конфигурации представляет собой регистрацию состояния компонентов ПП, подготовку отчетов обо всех реализованных и отвергнутых модификациях версий компонентов ПП. Совокупность отчетов обеспечивает однозначное отражение текущего состояния системы и ее компонентов, а также ведение истории модификации.

Оценка конфигурации заключается в оценке функциональной полноты компонентов ПП, а также соответствия их физического состояния текущему техническому описанию.

Управление выпуском и поставка включают в себя изготовление эталонных копий программ и документации, их хранение и поставку пользователям в соответствии с порядком (процессом), принятым в организации.

Процесс обеспечения качества (***quality assurance process***) обеспечивает соответствующие гарантии того, что ПП и процессы его жизненного цикла соответствуют заданным требованиям и утвержденным планам. Под качеством ПП понимается совокупность свойств, которые характеризуют способность ПП удовлетворять заданным требованиям.

Для получения достоверных оценок создаваемого ПП процесс обеспечения его качества должен происходить независимо от субъектов, непосредственно связанных с разработкой ПП. При этом могут использоваться результаты других вспомогательных процессов, таких как верификация, аттестация, совместная оценка, аудит и разрешение проблем.

Процесс обеспечения качества включает в себя:

подготовительную работу;

обеспечение качества продукта;

обеспечение качества процесса;

обеспечение прочих показателей качества системы.

Подготовительная работа заключается в координации с другими вспомогательными процессами и планировании самого процесса обеспечения качества с учетом используемых стандартов, методов, процедур и средств.

Обеспечение качества продукта подразумевает гарантирование полного соответствия ПП и его документации требованиям заказчика, предусмотренным в договоре.

Обеспечение качества процесса предполагает гарантирование соответствия процессов жизненного цикла ПП, методов разработки, среды разработки и квалификации персонала условиям договора, установленным стандартам и процедурам.

Обеспечение прочих показателей качества системы осуществляется в соответствии с условиями договора и стандартом качества ISO-9001.

Процесс верификации (***verification process***) состоит в доказательстве того, что ПП, являющиеся результатами некоторого действия, полностью удовлетворяют требованиям или условиям, зависящим от предшествующих действий.

Верификация может проводиться самим исполнителем или другим специалистом данной организации, а также специалистом сторонней организации. Тут возможны различные вариации, в соответствии с которыми можно говорить о различной степени независимости верификации. Если процесс верификации осуществляется организацией, не зависящей от поставщика, разработчика, оператора или службы сопровождения, то он называется процессом независимой верификации.

Процесс верификации включает в себя:

подготовительную работу;

собственно верификацию.

Подготовительная работа заключается в координации с другими вспомогательными процессами и планировании самого процесса верификации с учетом используемых стандартов, методов, процедур и средств. Для повышения эффективности верификация должна как можно раньше интегрироваться с использующими ее процессами (такими как поставка, разработка, эксплуатация или сопровождение).

Верификация в узком смысле означает формальное доказательство правильности ПП. Данный процесс может включать в себя анализ, оценку и тестирование.

В процессе верификации проверяются:

непротиворечивость требований к системе и степень учета потребностей пользователей;

возможности поставщика выполнить заданные требования;

соответствие выбранных процессов жизненного цикла ПП условиям договора;

адекватность стандартов, процедур и среды разработки процессам жизненного цикла ПП;

соответствие проектных спецификаций ПП заданным требованиям;

корректность описания в проектных спецификациях входных и выходных данных, последовательности событий, интерфейсов, логики и т.д.;

соответствие кода проектным спецификациям и требованиям;

тестируемость и корректность кода, его соответствие принятым стандартам кодирования;

корректность интеграции компонентов ПП в систему;

адекватность, полнота и непротиворечивость документации.

Процесс аттестации (***validation process***) предусматривает определение полноты соответствия заданных требований к создаваемой системе или ПП функциональному назначению последних. Под

аттестацией обычно понимают подтверждение и оценку достоверности проведенного тестирования ПП. Аттестация должна гарантировать полное соответствие ПП спецификациям, требованиям и документации, а также возможность его безопасного и надежного применения пользователем. Аттестацию рекомендуется выполнять путем тестирования во всех возможных ситуациях и использовать при этом независимых специалистов. Аттестация может проводиться на начальных стадиях жизненного цикла программного продукта или как часть работы по приемке программного продукта.

Аттестация так же, как и верификация, может осуществляться с различными степенями независимости. Если процесс аттестации выполняется организацией, не зависящей от поставщика, разработчика, оператора или службы сопровождения, то он называется процессом независимой аттестации.

Процесс аттестации включает в себя:

подготовительную работу;

собственно аттестацию.

Подготовительная работа заключается в координации с другими вспомогательными процессами и планировании самого процесса аттестации с учетом используемых стандартов, методов, процедур и средств.

Аттестация позволяет определить полноту соответствия разработанных требований к создаваемому ПП или системе функциональному назначению последних.

Процесс совместной оценки (***joint review process***) предназначен для оценки состояния работ по проекту и ПП, создаваемому при выполнении данных работ. Он заключается в основном в контроле за планированием и управлением ресурсами, персоналом, аппаратурой и инструментальными средствами проекта.

Оценка выполняется как на уровне управления проектом, так и на уровне его технической реализации и проводится в течение всего срока действия договора. Данный процесс может выполняться двумя любыми сторонами, участвующими в договоре, при этом одна сторона проверяет другую.

Процесс совместной оценки включает в себя:

подготовительную работу;

оценку управления проектом;

техническую оценку.

Подготовительная работа заключается в координации с другими вспомогательными процессами и планировании самого процесса оценки с учетом используемых стандартов, методов, процедур и средств.

Оценка управления проектом позволяет определить текущее состояние хода выполнения работ по оцениваемому проекту.

Техническая оценка проекта позволяет определить текущее состояние хода выполнения работ по технической реализации проекта.

Процесс аудита (***audit process***) представляет собой определение соответствия требованиям, планам и условиям договора как хода выполнения работ по созданию ПП, так и самого ПП. Аудит может выполняться двумя любыми сторонами, участвующими в договоре, когда одна сторона проверяет другую.

Аудит служит для установления соответствия реальных работ и отчетов требованиям, планам и контракту. Аудиторы (ревизоры) не должны иметь прямой зависимости от разработчиков ПП. Они оценивают состояние работ, использование ресурсов, соответствие документации спецификациям и стандартам, корректность проводимого тестирования.

Процесс аудита включает в себя:

подготовительную работу;

собственно аудит.

Подготовительная работа заключается в координации с другими вспомогательными процессами и планировании проведения аудиторских проверок с учетом установленных требований, планов, условий договора и контракта.

Аудит — это ревизия (проверка), проводимая компетентным органом (лицом) в целях обеспечения независимой оценки степени соответствия ПП или проводимых работ установленным требованиям, планам, условиям договора и контракта.

Процесс разрешения проблем (***problem resolution process***) предусматривает анализ и решение проблем (включая выявленные несоответствия), обнаруженных в ходе разработки, эксплуатации, сопровождения и других процессов, независимо от их происхождения или источника. Каждая обнаруженная проблема должна быть идентифицирована, описана, проанализирована и разрешена.

Процесс разрешения проблем включает в себя:

подготовительную работу;

собственно разрешение проблем.

Подготовительная работа заключается в координации с другими вспомогательными процессами и планировании работ по выявлению, анализу и решению возникших проблем.

Разрешение проблем проводится на протяжении всего жизненного цикла ПП и включает в себя действия по выявлению различных проблем или несоответствий их анализу и устранению.

1.4. Организационные процессы жизненного цикла программного продукта

Основной целью организационных процессов является организация процесса разработки надежного, полностью удовлетворяющего требованиям заказчика ПП в установленные договором

сроки и управление этим процессом. К организационным относятся процессы управления, создания инфраструктуры, усовершенствования, обучения.

Процесс управления (***management process***) состоит из действий и задач, которые могут выполняться любой стороной, управляющей своими процессами. Данная сторона (менеджер) отвечает за управление выпуском продукта, проектом и задачами соответствующих процессов, таких как приобретение, поставка, разработка, эксплуатация, сопровождение и др.

Процесс управления включает в себя:

инициирование и определение области управления;

планирование;

управление работами по созданию ПП и контроль за их выполнением;

проверку и оценку;

завершение работ.

При *инициировании и определении области управления* менеджер должен определить необходимые для управления ресурсы (персонал, оборудование и технология) и убедиться, что они имеются в его распоряжении, причем в достаточном количестве.

Планирование подразумевает выполнение, как минимум, следующих задач: составление графиков выполнения работ; оценку затрат; выделение требуемых ресурсов; распределение ответственности; оценку рисков, связанных с конкретными задачами; создание инфраструктуры управления.

Задачи планирования подробно рассмотрены в гл. 6.

Управление работами по созданию ПП и *контроль за их выполнением* осуществляются в соответствии с результатами планирования.

В ходе выполнения работ обязательно должны выполняться регулярная *проверка* их выполнения и *оценка* достигнутых результатов. При необходимости по результатам проверки и оценки могут быть внесены корректировки в ход выполнения работ.

Завершение работ происходит после выполнения всех обязательств, взятых поставщиком перед заказчиком в соответствии с заранее оговоренными процедурами.

Процесс создания инфраструктуры (***infrastructure process***) охватывает выбор и поддержку (сопровождение) технологии, стандартов и инструментальных средств, выбор и установку аппаратных и программных средств, используемых для разработки, эксплуатации или сопровождения ПП. Инфраструктура должна модифицироваться и сопровождаться в соответствии с изменениями требований к соответствующим процессам. Инфраструктура, в свою очередь, является одним из объектов управления конфигурацией.

Процесс создания инфраструктуры включает в себя:

подготовительную работу;

создание инфраструктуры;

сопровождение инфраструктуры.

Подготовительная работа заключается в координации с другими организационными процессами и планировании работ по созданию инфраструктуры с учетом выбранных технологий, стандартов, инструментальных, программных и аппаратных средств.

Создание инфраструктуры включает в себя все действия по разработке в соответствии с выбранной концепцией и планом инфраструктуры для выполнения работ по созданию ПП.

Сопровождение инфраструктуры вызвано необходимостью сопровождения ПП и возможными модификациями продукта в соответствии с изменившимися требованиями к нему.

Процесс усовершенствования (***improvement process***) предусматривает оценку, измерение, контроль и усовершенствование процессов жизненного цикла ПП. Данный процесс включает в себя:

создание процесса;

оценку процесса;

усовершенствование процессов жизненного цикла ПП.

Создание процесса усовершенствования процессов жизненного цикла ПП позволяет на основе контроля за ходом выполнения процессов жизненного цикла, измерения характеристик и оценки полученных результатов существенно улучшить качество разрабатываемого ПП и сократить сроки его создания.

Оценка процесса разработки ПП позволяет выявить его сильные и слабые стороны и на основе полученных результатов провести необходимые улучшения.

Усовершенствование процессов жизненного цикла ПП направлено на повышение производительности труда всех участвующих в них специалистов за счет совершенствования используемой технологии, методов управления, выбора инструментальных средств и обучения персонала. Усовершенствование основано на анализе достоинств и недостатков каждого процесса. Такому анализу в большой степени способствует накопление в организации исторической, технической, экономической и иной информации по реализованным проектам.

Процесс обучения (***training process***) охватывает первоначальное обучение и последующее постоянное повышение квалификации персонала. Приобретение, поставка, разработка, эксплуатация и сопровождение программного продукта в значительной степени зависят от уровня знаний и квалификации персонала. Например, разработчики ПП должны пройти необходимое обучение методам и средствам программной инженерии. Содержание процесса обучения определяется требованиями к проекту. Для этого процесса должны быть запланированы необходимые ресурсы и технические средства обучения. Кроме того, должны быть разработаны и пред-

ставлены методические материалы, необходимые для обучения пользователей в соответствии с учебным планом.

Процесс обучения включает в себя:

подготовительную работу;

разработку учебных материалов;

реализацию плана обучения.

Подготовительная работа заключается в координации с другими организационными процессами и планировании работ по созданию плана обучения и повышения квалификации.

Разработка учебных материалов является неотъемлемой частью процесса обучения, так как позволяет существенно повысить его эффективность и качество.

Реализация плана обучения должна осуществляться непрерывно в течение всего времени, для которого этот план разработан.

1.5. Взаимосвязь между процессами жизненного цикла программного продукта

Процессы жизненного цикла ПП, регламентируемые стандартом ISO/IEC 12207, могут использоваться различными организациями в конкретных проектах самым различным образом. Тем не менее, стандарт предлагает некоторый базовый набор взаимосвязей между процессами с различных точек зрения, или в различных аспектах (договорном, управления, эксплуатации, инженерном, поддержки), который показан на рис. 1.1. Штриховые стрелки показывают связь действующих лиц процессов (заказчик, поставщик и т.д.) с конкретными процессами, а сплошные стрелки — связь процессов или групп процессов между собой.

В *договорном аспекте* заказчик и поставщик вступают в договорные отношения и реализуют соответственно процессы приобретения и поставки.

В *аспекте управления* заказчик, поставщик, разработчик, оператор, служба сопровождения и другие стороны, участвующие в жизненном цикле ПП, управляют выполнением своих процессов. Менеджер является связующим звеном между организационными и основными процессами.

В *аспекте эксплуатации* оператор, эксплуатирующий систему, предоставляет необходимые услуги пользователям.

В *инженерном аспекте* разработчик или служба сопровождения решают соответствующие технические задачи, разрабатывая или модифицируя ПП.

В *аспекте поддержки* службы, реализующие вспомогательные процессы, предоставляют необходимые услуги всем остальным участникам работ. В рамках аспекта поддержки можно выделить аспект управления качеством ПП.

Рис. 1.1. Связь между процессами жизненного цикла программного продукта

Организационные процессы выполняются на корпоративном уровне, т.е. на уровне всей организации в целом, создавая базу для реализации и постоянного совершенствования остальных процессов жизненного цикла ПП.

Процессы и реализующие их организации (или стороны) связаны между собой чисто функционально. При этом внутренняя структура и статус организаций никак не регламентируются. Одна и та же организация может выполнять различные роли (поставщика, разработчика и др.), и, наоборот, одна и та же роль может выполняться несколькими организациями.

Взаимосвязи между процессами, описанные в стандарте, носят статический характер. Более важные динамические связи между процессами и реализующими их сторонами устанавливаются в реальных проектах.

Контрольные вопросы

1. Сформулируйте определение понятия «жизненный цикл программного продукта».

2. Какими документами регламентируется жизненный цикл программного продукта?

3. Какими стандартами регламентировалось прежде создание программного продукта в России?

4. На какие группы можно разделить процессы жизненного цикла программного продукта?

5. Какие процессы включены в состав каждой группы?

6. Какие действия входят в состав процесса приобретения и каково их назначение?

7. Какие действия входят в состав процесса поставки и каково их назначение?

8. Какие действия и задачи выполняются в ходе процесса разработки?

9. Какие действия входят в состав процесса эксплуатации и каково их назначение?

10. Что понимается под процессом сопровождения?

11. Какие действия входят в состав процесса документирования и каково их назначение?

12. Какие действия входят в состав процесса управления конфигурацией и каково их назначение?

13. Какие действия входят в состав процесса обеспечения качества и каково их назначение?

14. В чем отличие процесса верификации от процесса аттестации?

15. Какие условия проверяются в ходе процесса верификации?

16. Что подразумевается под процессом независимой аттестации?

17. В чем отличие процесса совместной оценки от процесса аудита?

18. Какие задачи выполняются в процессе разрешения проблем?

19. Какие действия входят в состав процесса управления и каково их назначение?

20. Какие задачи выполняются в процессе создания инфраструктуры?

21. Какие цели преследует процесс обучения?

22. Перечислите основные аспекты взаимодействия между различными процессами жизненного цикла программного продукта.

23. Чем, по вашему мнению, объясняется наличие взаимосвязей между различными процессами жизненного цикла программного продукта?

ГЛАВА 2

ОСНОВНЫЕ ЭТАПЫ РАБОТЫ ПО СОЗДАНИЮ ПРОГРАММНОГО ПРОДУКТА

2.1. Длительность основных этапов

К существенному повышению эффективности работы и качества создаваемого ПП приводит использование единых правил и технологии. Единый процесс создания ПП должен включать в себя методологию, методы, стандарты и инструментальные средства, которые необходимо использовать при выполнении всех работ по созданию высококачественного ПП.

При создании ПП можно выделить шесть основных этапов работы:

1) планирование программного проекта;
2) составление требований заказчика;
3) проектирование ПП;
4) разработка ПП;
5) тестирование ПП;
6) сопровождение ПП.

Характерная длительность каждого из этапов жизненного цикла ПП показана на рис. 2.1.

Первые два этапа создания ПП начинаются практически одновременно, при этом этап планирования программного проекта (*1*) заканчивается всегда раньше, чем этап составления требований заказчика (*2*). На этих двух этапах определяют содержание и сроки работы по созданию будущего ПП. Большая длительность этих этапов объясняется тем, что в процессе работы над ПП при-

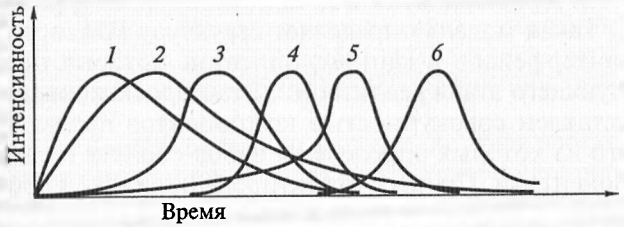


Рис. 2.1. Длительность этапов жизненного цикла программного продукта:
1—6 — номера этапов

ходится вносить коррективы в план работ, а также, возможно, и в требования к ПП.

Этап тестирования (*5*) начинается практически одновременно с этапами *1* и *2*. Такое раннее начало тестирования позволяет выявить ошибки на первых стадиях, что в дальнейшем дает возможность сэкономить время и средства на устранение ошибок. На ранних стадиях тестируется не сам ПП, а разрабатываемая проектная документация.

2.2. Характеристика основных этапов

Планирование программного проекта. В течение этапа планирования определяются все основные задачи, которые должны быть выполнены в процессе разработки, производится оценка финансовых, людских, технических и нетехнических ресурсов, объемов и сложности разрабатываемого ПП, определяются методы тестирования и критерии приемки ПП, методы и технология выполнения работы, строятся временные графики выполнения работ.

Составление требований заказчика. В течение этого этапа разработчики анализируют требования к ПП (форма представления информации, необходимые функции, желательные интерфейсы, существующие ограничения и т.д.).

Данный этап служит для выработки взаимопонимания между разработчиками и заказчиком относительно требований к ПП, для устранения неопределенности требований, четкого, однозначного понимания и определения всех деталей, касающихся будущего ПП, и его тестируемости.

Требования тестируемы в такой степени, в какой разработчик тестов может построить ясный тест, дающий однозначный ответ «да» или «нет» и определяющий соответствие разрабатываемого ПП данной спецификации требований. Для тестируемости спецификация должна быть очень конкретной, недвусмысленной и обладать по возможности количественными характеристиками.

Проектирование программного продукта. Этап проектирования предназначен для выработки и детализации модели разрабатываемого ПП. Такая модель определяет структуру ПП, организацию модулей, интерфейсов и данных, описание которых необходимо для последующего этапа реализации. Этап проектирования может быть представлен совокупностью компонентов проектирования, для каждого из которых определены набор свойств и связи с другими компонентами. Процесс проектирования должен проводиться в соответствии с теми методами и технологией разработки, которые были определены в плане проекта. Проектирование может состоять из двух частей: высокоуровневого и низкоуровневого (детального) проектирования.

Разработка программного продукта. В процессе выполнения этого этапа разработчики преобразуют результаты этапа проектирования в коды программ на используемом языке программирования в соответствии со стандартами кодирования. Разработчики также взаимодействуют с инженером по тестированию ПП для создания надлежащих условий для тестирования. Кроме этого, разработчики ведут работу по созданию технической документации и начинают планировать и выполнять интеграцию ПП.

Тестирование программного продукта. Этап тестирования не имеет четко определенного начала, но чем раньше он начинается, тем больше уверенности, что разрабатываемый ПП будет точно соответствовать требованиям заказчика. Все действия по тестированию должны выполняться специальным работником — тестировщиком. Он разрабатывает тесты, выполняет процедуру тестирования и составляет отчеты о результатах тестирования.

Общий набор тестов должен обеспечивать максимальный охват тестируемого ПП, т.е. тестированию должны быть подвергнуты как отдельные его модули, так и весь продукт в целом, включая его системные свойства в соответствии с зафиксированными в спецификации требованиями.

Сопровождение программного продукта. На этапе сопровождения основное внимание уделяется внесению изменений в ПП. Эти изменения могут быть связаны с устранением ошибок, дополнительными пожеланиями заказчика, появившимися в результате работы с ПП, изменением среды окружения и функционирования.

Если изменение признается необходимым, то следует запланировать работу по внесению данного изменения, задокументировать ее, выполнить, после чего произвести обзор результатов работы по изменению ПП.

Контрольные вопросы

1. Перечислите основные этапы работ по созданию программного продукта в порядке их выполнения.
2. Укажите характерные соотношения длительностей этапов работ по созданию программного продукта.
3. На каком этапе работ выполняется оценка необходимых ресурсов, объемов и сложности разрабатываемого программного продукта?
4. Какие критерии предъявляются к спецификации требований при разработке тестов?
5. Какая цель преследуется при выполнении этапа проектирования?
6. На какие части может быть разбит этап проектирования?
7. На каком этапе производится преобразование результатов проектирования в программный продукт?
8. Чем может быть вызвана необходимость внесения изменений в программный продукт, находящийся в эксплуатации, и на каком этапе эта работа выполняется?

ГЛАВА 3

МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА

3.1. Понятие модели жизненного цикла разработки программного продукта. Обзор существующих моделей

Под моделью жизненного цикла разработки ПП понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач, выполняемых на протяжении жизненного цикла разработки ПП. Модель жизненного цикла зависит от специфики и сложности выполняемого проекта, а также от условий, в которых создается и будет функционировать ПП.

Стандарт ISO/IEC 12207 не предлагает конкретные модель жизненного цикла и методы разработки ПП. Положения стандарта являются общими для любых моделей жизненного цикла, методов и технологий разработки ПП.

Стандарт описывает структуру процессов жизненного цикла ПП, но не уточняет, как выполнить действия и задачи, включенные в эти процессы.

Модель жизненного цикла любого конкретного ПП определяет характер процесса его создания, который представляет собой совокупность упорядоченных во времени, взаимосвязанных и объединенных в этапы работ, выполнение которых необходимо и достаточно для создания ПП, соответствующего заданным требованиям. Под этапом разработки ПП понимается часть процесса создания ПП, ограниченная некоторыми временными рамками и заканчивающаяся выпуском конкретного продукта (моделей ПП, программных компонентов, документации), определяемого заданными для данной стадии требованиями. Этапы создания ПП выделяются по соображениям рационального планирования и организации работ, заканчивающихся заданными результатами (см. гл. 2).

Наибольшее распространение получили следующие модели жизненного цикла разработки ПП:

каскадная модель, или «водопад» (Waterfall model);

V-образная модель (V-shaped model);

модель прототипирования (Prototype model);

модель быстрой разработки приложений, или RAD-модель (RAD — Rapid Application Development model);

Таблица 3.1

Модели жизненного цикла разработки программного продукта

Название	Характеристики
Каскадная модель	Прямолинейная и простая в использовании. Необходим постоянный жесткий контроль за ходом работы. Разрабатываемое программное обеспечение не доступно для изменений
V-образная модель	Простая в использовании. Особое значение придается тестированию и сравнению результатов фаз тестирования и проектирования
Модель прототипирования	Создается «быстрая» частичная реализация системы до составления окончательных требований. Обеспечивается обратная связь между пользователями и разработчиками в процессе выполнения проекта. Используемые требования не полные
Модель быстрой разработки приложений	Проектные группы небольшие (3...7 человек) и составлены из высококвалифицированных специалистов. Уменьшенное время цикла разработки (до 3 мес) и улучшенная производительность. Повторное использование кода и автоматизация процесса разработки
Многопроходная модель	Быстро создается работающая система. Уменьшается возможность внесения изменений в процессе разработки. Невозможен переход от текущей реализации к новой версии в течение построения текущей частичной реализации
Спиральная модель	Охватывает каскадную модель. Расчленяет фазы на меньшие части. Позволяет гибко выполнять проектирование. Анализирует риски и управляет ими. Пользователи знакомятся с ПП на более раннем этапе благодаря прототипам

многопроходная модель (Incremental model);

спиральная модель (Spiral model).

Краткие характеристики каждой из перечисленных моделей приведены в табл. 3.1.

3.2. Каскадная модель

В однородных информационных системах 1970-х и 1980-х годов прикладные ПП представляли собой единое целое. Для разработки такого типа ПП применялась каскадная модель, или «водопад» (waterfall) (рис. 3.1).

Принципиальная особенность каскадного подхода: переход на следующий этап осуществляется только после того, как будет полностью завершена работа на текущем этапе, и возвратов на пройденные этапы не предусматривается. Каждый этап заканчивается получением некоторых результатов, которые служат исходными данными для следующего этапа. Требования к разрабатываемому ПП, определенные на этапе формирования требований, строго документируются в виде технического задания и фиксируются на все время разработки проекта. Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков. Критерием качества разработки при таком подходе является точность выполнения спецификаций технического задания. При этом основное внимание разработчиков сосредоточивается на достижении оптимальных значений технических характеристик разрабатываемого ПП — производительности, объема занимаемой памяти и др.

Преимущества каскадного способа:

на каждой стадии формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;

выполняемые в логичной последовательности стадии работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадный подход хорошо зарекомендовал себя при построении информационных систем, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования с целью предоставить разработчикам свободу реализовать их технически как можно лучше. В эту категорию попадают сложные системы с большим числом задач вычислительного характера, системы реального времени и др.

Рис. 3.1. Каскадная модель

В то же время данный подход обладает рядом существенных недостатков, обусловленных прежде всего тем, что реальный процесс разработки ПП никогда полностью не укладывается в такую жесткую схему. Этот процесс носит, как правило, итерационный характер: результаты очередного этапа часто вызывают изменения в проектных решениях, выработанных на более ранних стадиях. Таким образом, постоянно возникает потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс разработки принимает иной вид (рис. 3.2).

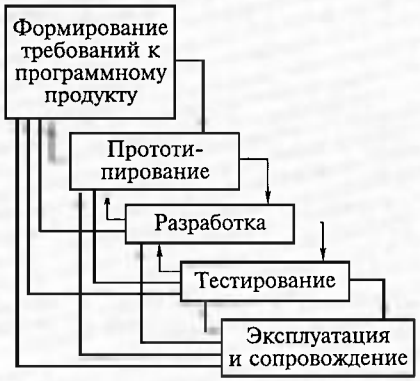


Рис. 3.2. Схема реального процесса разработки программного продукта

Изображенную на рис. 3.2 схему часто относят к отдельной так называемой модели с промежуточным контролем, в которой межстадийные корректировки обеспечивают большую надежность по сравнению с каскадной моделью, хотя и увеличивают весь период разработки.

Основными недостатками каскадного подхода являются существенное запаздывание с получением результатов и, как следствие, достаточно высокий риск создания системы, не удовлетворяющей изменившимся потребностям пользователей.

Практика показывает, что на начальной стадии проекта полностью и точно сформулировать все требования к будущей системе не удается.

Это объясняется следующими причинами:

пользователи не в состоянии сразу изложить все свои требования и не могут предвидеть, как они изменятся в ходе разработки;

за время разработки могут произойти изменения во внешней среде, которые повлияют на требования к системе.

В рамках каскадного подхода требования к ПП фиксируются в виде технического задания на все время создания, а согласование получаемых результатов с пользователями производится только после завершения каждого этапа (при этом возможна корректировка результатов по замечаниям пользователей, если они не затрагивают требования, изложенные в техническом задании). Таким образом, пользователи могут внести существенные замечания только после того, как работа над системой будет полностью завершена. В случае неточного изложения требований или их изменения в течение длительного периода создания ПП пользователи получают систему, не удовлетворяющую их потребностям. В результате приходится начинать новый проект, который может постигнуть та же участь.

3.3. V-образная модель

Эта модель (рис. 3.3) была разработана как разновидность каскадной модели, в которой особое внимание уделяется верификации и аттестации ПП. Модель показывает, что тестирование продукта обсуждается, проектируется и планируется, начиная с ранних этапов жизненного цикла разработки (на рис. 3.3 этот процесс обозначен штриховыми стрелками).

От каскадной модели V-образная модель унаследовала последовательную структуру, в соответствии с которой каждая последующая фаза начинается только после успешного завершения предыдущей фазы.

Данная модель основана на систематическом подходе к проблеме, для решения которой определены четыре базовых шага: анализ, проектирование, разработка и обзор. При выполнении анализа осуществляются планирование проекта и составление требований. Проектирование разделяется на высокоуровневое и детальное (низкоуровневое). Разработка включает в себя кодирование, а обзор — различные виды тестирования.

На модели хорошо просматриваются взаимосвязи между аналитическими фазами и фазами проектирования, которые предшествуют кодированию и тестированию. Штриховые стрелки показывают, что эти фазы надо рассматривать параллельно.

Рис. 3.3. V-образная модель

Модель включает в себя следующие фазы:

составление требований к проекту и планирование — определяются системные требования и выполняется планирование работ;

составление требований к продукту и их анализ — составляется полная спецификация требований к программному продукту;

высокоуровневое проектирование — определяются структура ПП, взаимосвязи между основными его компонентами и реализуемые ими функции;

детальное проектирование — определяется алгоритм работы каждого компонента;

кодирование — выполняется преобразование алгоритмов в готовое программное обеспечение;

модульное тестирование — выполняется проверка каждого компонента или модуля ПП;

интеграционное тестирование — осуществляются интеграция ПП и его тестирование;

системное тестирование — выполняется проверка функционирования ПП после помещения его в аппаратную среду в соответствии со спецификацией требований;

эксплуатация и сопровождение — запуск ПП в производство. На этой фазе в ПП могут вноситься поправки и может выполняться его модернизация.

ПреимуществаV-образной модели:

большая роль придается верификации и аттестации ПП, начиная с ранних стадий его разработки, все действия планируются;

предполагаются аттестация и верификация не только самого ПП, но и всех полученных внутренних и внешних данных;

ход выполнения работы может легко отслеживаться, так как завершение каждой фазы является контрольной точкой.

Кроме перечисленных достоинств модель обладает и рядом недостатков:

не учитываются итерации между фазами;

нельзя вносить изменения на разных этапах жизненного цикла;

тестирование требований происходит слишком поздно, поэтому внесение изменений влияет на выполнение графика работ.

Данную модель целесообразно использовать при разработке программных продуктов, главным требованием для которых является высокая надежность.

3.4. Модель прототипирования

Модель прототипирования (рис. 3.4) позволяет создать прототип ПП до или в течение этапа составления требований к ПП.

Рис. 3.4. Модель прототипирования

Потенциальные пользователи работают с этим прототипом определяя его сильные и слабые стороны, о результатах сообщают разработчикам ПП. Таким образом, обеспечивается обратная связь между пользователями и разработчиками, которая используется для изменения или корректировки спецификации требований к ПП. В результате такой работы продукт будет отражать реальные потребности пользователей.

Жизненный цикл разработки ПП начинается с разработки плана проекта (на рис. 3.4 этапу планирования соответствует центр эллипса), затем выполняется быстрый анализ, после чего создаются база данных (если, конечно, она используется в ПП), пользовательский интерфейс и выполняется разработка необходимых функций. В результате этой работы получается документ, содержащий частичную спецификацию требований к ПП. Данный документ в дальнейшем является основой для итерационного цикла быстрого прототипирования.

В результате прототипирования разработчик демонстрирует пользователям готовый прототип, а пользователи оценивают его функционирование. После этого определяются проблемы, над устранением которых совместно работают пользователи и разработчики. Этот процесс продолжается до тех пор, пока пользователи не будут удовлетворены степенью соответствия ПП, поставленным перед ним требованиям. Затем прототип демонстрируют пользователям с целью получения предложений по его усовершенствованию, которые включаются в последовательные итерации до тех пор, пока рабочая модель не окажется удовлетворительной. После этого получают от пользователей официальное

одобрение (утверждение) функциональных возможностей прототипа и выполняют его окончательное преобразование в готовый ПП.

Модель прототипирования обладает целым рядом преимуществ:

взаимодействие заказчика с разрабатываемой системой начинается на раннем этапе;

благодаря реакции заказчика на прототип сводится к минимуму число неточностей в требованиях;

снижается вероятность возникновения путаницы, искажения информации или недоразумений при определении требований к ПП, что приводит к созданию более качественного ПП;

в процессе разработки всегда можно учесть новые, даже неожиданные требования заказчика;

прототип представляет собой формальную спецификацию, воплощенную в ПП;

прототип позволяет очень гибко выполнять проектирование и разработку, включая несколько итераций на всех фазах жизненного цикла разработки;

заказчик всегда видит прогресс в процессе разработки ПП;

возможность возникновения противоречий между разработчиками и заказчиками сведена к минимуму;

уменьшается число доработок, что снижает стоимость разработки:

возникающие проблемы решаются на ранних стадиях, что резко сокращает расходы на их устранение;

заказчики принимают участие в процессе разработки на протяжении всего жизненного цикла и в конечном итоге в большей степени довольны результатами работы.

Кроме указанных достоинств модели прототипирования при- и целый ряд недостатков:

решение сложных задач может отодвигаться на будущее;

заказчик может предпочесть получить прототип, а не законченную полную версию ПП;

прототипирование может неоправданно затянуться;

перед началом работы неизвестно, сколько итераций придется выполнить.

Модель прототипирования рекомендуется применять в следующих случаях:

требования к ПП заранее неизвестны,

требования не постоянны или неудачно сформулированы;

требования необходимо уточнить;

нужна проверка концепции;

существует потребность в пользовательском интерфейсе;

выполняется новая, не имеющая аналогов разработка;

разработчики не уверены в том, какое решение следует выбрать.

3.5. Модель быстрой разработки приложений (RAD-модель)

В RAD-модели (рис. 3.5) конечный пользователь играет решающую роль. В тесном взаимодействии с разработчиками он участвует в формировании требований и апробации их на работающих прототипах. Таким образом, в начале жизненного цикла на конечного пользователя выпадает большая часть работы, но в результате этого создаваемая система формируется более быстро.

В традиционном жизненном цикле разработки большую часть работы составляют программирование и тестирование. При автоматизации программирования и повторном использовании кода, применяемых в RAD-модели, большую часть работы составляют планирование и проектирование.

На рис. 3.5, поясняющем принцип RAD-модели, указаны этапы процесса разработки и отображено участие заказчиков (штриховая линия) на каждом из них.

Модель включает в себя следующий фазы:

составление требований и планирование — осуществляются с использованием так называемого метода совместного планирования требований (планирование работ по созданию ПП и составление требований к ПП выполняются одновременно), который заключается в структурном анализе и обсуждении решаемых задач;

описание пользователя — проектирование ПП, выполняемое при непосредственном участии заказчика;

создание — детальное проектирование, кодирование и тестирование ПП, а также поставка его заказчику;

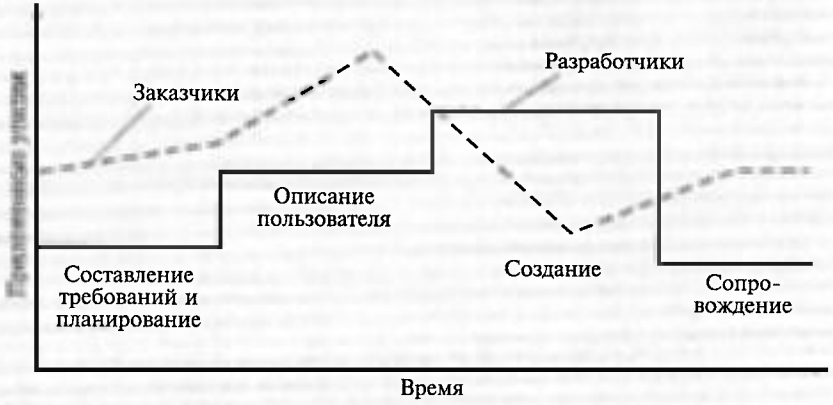


Рис. 3.5. RAD-модель

сопровождение — приемочные испытания, установка ПП и обучение пользователей.

Модель обладает следующими достоинствами:

использование современных инструментальных средств позволяет сократить время цикла разработки;

привлечение к работе заказчика сводит к минимуму риск того, что он останется недоволен готовым ПП;

повторно используются компоненты уже существующих программ.

В то же время ей присущи и недостатки:

если заказчики не могут постоянно участвовать в процессе разработки, то это может негативно сказаться на ПП;

для работы нужны высококвалифицированные кадры, умеющие пользоваться современными инструментальными средствами;

существует риск, что работа над ПП никогда не будет завершена, так как может быть зациклена, поэтому всегда надо вовремя остановиться.

Рассмотренную RAD-модель можно применять при разработке ПП, которые хорошо поддаются моделированию, когда требования к ПП хорошо известны, а заказчик может принять непосредственное участие в процессе разработки.

3.6. Многопроходная модель

Многопроходная модель (рис. 3.6) — это несколько итераций процесса построения прототипа ПП с добавлением на каждой следующей итерации новых функциональных возможностей или повышением эффективности ПП.

Предполагается, что на ранних этапах жизненного цикла разработки (планирование, анализ требований и разработка проекта) выполняется конструирование ПП в целом. Тогда же определяется и число необходимых инкрементов и относящихся к ним функций. Каждый инкремент затем проходит через оставшиеся фазы жизненного цикла (кодирование и тестирование). Сначала выполняются конструирование, тестирование и реализация базовых функций, составляющих основу ПП. Последующие итерации направлены на улучшение функциональных возможностей ПП.

Преимущества многопроходной модели:

в начале разработки требуются средства только для разработки и реализации основных функций ПП;

после каждого инкремента получается функциональный продукт;

снижается риск неудачи и изменения требований;

улучшается понимание как разработчиками, так и пользователями ПП требований для более поздних итераций;

Рис. 3.6. Многопроходная модель

инкременты функциональных возможностей легко поддаются тестированию.

Недостатки многопроходной модели:

не предусмотрены итерации внутри каждого инкремента;

определение полной функциональности должно быть осуществлено в самом начале жизненного цикла разработки;

может возникнуть тенденция оттягивания решения трудных задач;

общие затраты на создание ПП не будут снижены по сравнению с другими моделями;

обязательным условием является наличие хорошего планирования и проектирования.

Многопроходная модель может быть применена, если большинство требований к ПП будут сформулированы заранее, а для выполнения проекта будет выделен большой период времени.

3.7. Спиральная модель

Для преодоления проблем, связанных с использованием многопроходной модели, в середине 1980-х годов была предложена спиральная модель жизненного цикла (рис. 3.7). Ее принципиальная особенность заключается в том, что прикладной ПП создается не сразу, как в случае каскадного подхода, а по частям с использованием метода прототипирования. Под прототипом понимается действующий программный компонент, реализующий отдельные функции и внешние интерфейсы разрабатываемого ПП. Создание прототипов осуществляется за несколько итераций, или витков спирали. Каждая итерация соответствует созданию фрагмента, или версии ПП, на ней уточняются цели и характеристики проекта, оценивается качество полученных результатов и планируются работы следующей итерации. На каждой итерации производится тщательная оценка риска превышения сроков и стоимости проекта с целью определения необходимости выполнения еще одной итерации, степени полноты и точности понимания требований к системе, а также целесообразности прекращения проекта. Спиральная модель избавляет пользователей и разработчиков ПП от полного и точного формулирования требований к системе на начальной стадии, поскольку они уточняются на каждой итерации. Таким образом, углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который доводится до реализации.

Разработка итерациями отражает объективно существующий спиральный цикл создания системы, позволяя переходить на следующую стадию, не дожидаясь полного завершения работы на текущей стадии, поскольку при итеративном способе разработки недостающую работу можно выполнить на следующей итерации. Главная задача такой разработки — как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Спиральная модель не исключает использования каскадного подхода на завершающих стадиях проекта в тех случаях, когда требования к системе оказываются полностью определенными.

Основная проблема спирального цикла — определение момента перехода на следующую стадию. Для ее решения необходимо ввести временные ограничения на каждую из стадий жизненного

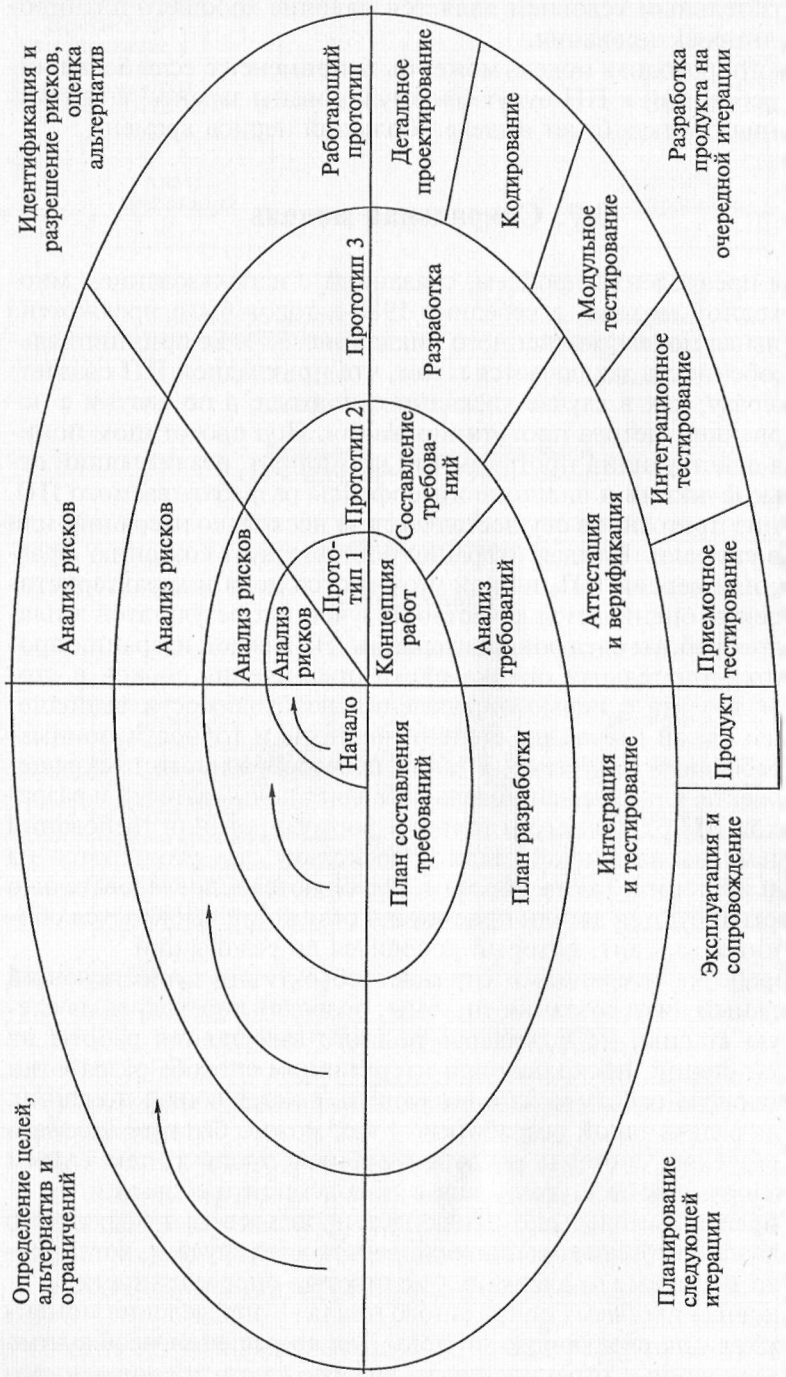


Рис 3.7. Спиральная модель

икла. Переход осуществляется в соответствии с планом, даже сли не вся запланированная работа закончена. План составляется а основе статистических данных, полученных в предыдущих роектах, и личного опыта разработчиков.

Спиральная модель обладает следующими достоинствами:

заказчик имеет возможность увидеть разрабатываемый ПП на ранних стадиях разработки;

заказчики принимают активное участие в разработке ПП;

в модели воплощены преимущества каскадной и многопроходной моделей.

Недостатки спиральной модели:

усложненная структура;

спираль может продолжаться до бесконечности, так как каждая ответная реакция заказчика может породить новый цикл.

В качестве модели жизненного цикла разработки программного продукта большое распространение получила улучшенная спиральная модель, показанная на рис. 3.8. В отличие от ранее рассмотренной спиральной модели эта модель использует каскадный подход на завершающих этапах разработки ПП.

Использование спиральной модели целесообразно, если существует хотя бы одна из следующих причин:

целесообразно создание прототипа;

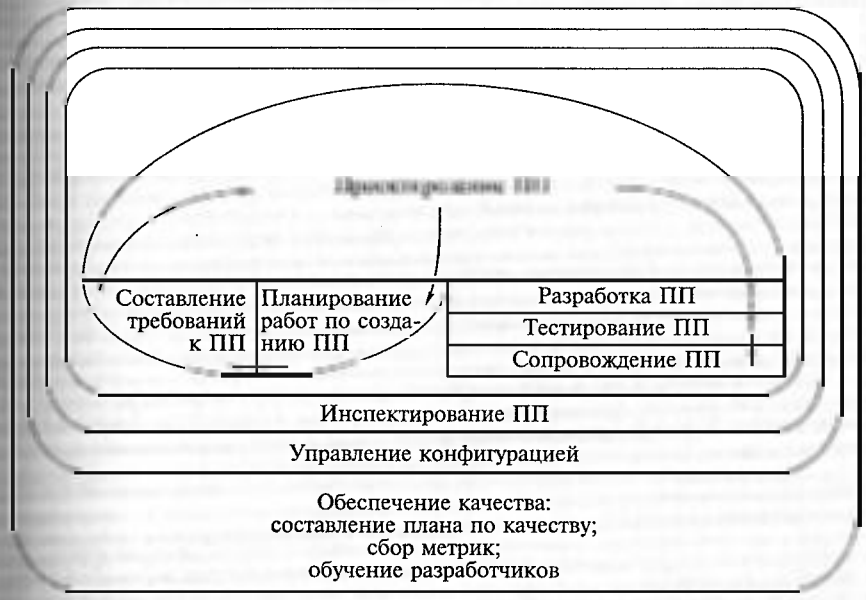


Рис. 3.8. Улучшенная спиральная модель с указанием вспомогательных процессов

организация обладает навыками, требуемыми для адаптации модели;

требуется выполнять проекты со средней и высокой степенями риска;

заказчики не уверены в своих потребностях;

требования слишком сложные;

проект очень большой.

3.8. Вспомогательные (поддерживающие) процессы

Как видно на рис. 3.8, процесс создания ПП базируется на поддерживающих процессах: инспектирования ПП; управления конфигурацией; обеспечения качества.

Основной целью поддерживающих процессов является создание надежного, полностью удовлетворяющего требованиям заказчика программного продукта в кратчайшие сроки.

Инспектирование программного продукта. Основной целью инспектирования ПП является повышение качества разрабатываемого ПП и производительности разработчиков. Различные виды инспектирования предназначены для оказания помощи разработчикам в выявлении дефектов на всех этапах работы по созданию ПП. Кроме того, периодическое проведение мероприятий по инспектированию способствует налаживанию сотрудничества между представителями различных проектных групп внутри компании.

Различают следующие виды инспектирования: высшим руководством; группой качества; представителями других проектов; представителями другого проекта; дружеская инспекция.

Инспектирование высшим руководством должно проводиться регулярно, через установленные в процессе компании промежутки времени (например, раз в квартал).

При проведении этого инспектирования каждый из сотрудников проекта докладывает о проделанной им работе, возникших проблемах и, возможно, найденных интересных решениях. Руководитель проекта рассказывает о ходе выполнения работ в целом, также отмечая полученные результаты и возникшие проблемы.

Инспектирование группой качества обычно проводится после завершения каждого этапа по созданию ПП. При этом проверяются наличие отчетной и проектной документации, соответствие этой документации и выполненной работы стандартам и процедурам компании.

Инспектирование представителями других проектов осуществляется на общем организационном собрании компании, которое обычно проводится каждую неделю. На нем выступают представители всех проектов и докладывают о ходе выполнения своих работ, достигнутых результатах и возникших трудностях, делятся

редовым опытом или перенимают его у других. В случае возникновения проблем в каком-то из проектов совместно ищут пути их шения.

При внедрении нового инструментального или прикладного ограммного обеспечения в процесс разработки ПП обычно обтку новшества проводят на одном из проектов (пилотном). На щем собрании компании представители этого проекта рассказывают о результатах внедрения новшества и рекомендуют его для еобщего использования или нет.

Инспектирование представителями другого проекта проводится просьбе представителей инспектируемого проекта. Такой вид спекции позволяет совместно решать возникшие проблемы и ксимально использовать знания и опыт наиболее квалифицированных специалистов.

Дружеская инспекция проводится между сотрудниками одного оекта. Любой сотрудник проекта может попросить другого сотрудника этого же проекта посмотреть и проверить результаты своей боты. Это иногда позволяет легче находить ошибки.

По результатам всех инспекций обязательно составляется от-т, в котором указываются обнаруженные проблемы, способы и оки их устранения, а также ответственные за их устранение.

Несмотря на то что инспектирование не решает всех проблем устранению дефектов, их своевременное выявление полезно, устранение требует гораздо меньших усилий и средств.

Основные цели инспектирования:

выявление ошибок на ранних этапах работы над ПП;

проверка соответствия хода работы установленным стандартам процедурам компании;

контроль за своевременностью выполнения основных и прометочных этапов в процессе работы по созданию ПП;

проверка формального завершения некоторой технической задачи;

проверка согласованности в работе сотрудников;

выявление путей дальнейшего улучшения работы и т.д.

Управление конфигурацией. Подробно вопрос управления конигурацией был рассмотрен в подразд. 1.2. К сказанному можно бавить только несколько замечаний.

Управление конфигурацией является процессом поддержки лостности ПП на протяжении всего его жизненного цикла.

Каждая проектная группа должна иметь ответственного за управление конфигурацией и свой план управления конфигурацией составе общего плана управления конфигурацией компании.

Проектный план управления конфигурацией составляется в ответствии с общим указанным планом.

Обеспечение качества. Подробно вопрос обеспечения качества л рассмотрен в подразд. 1.2. К сказанному можно добавить тольнесколько замечаний.

Обеспечение качества ПП заключается в проверке исполнения всеми сотрудниками принятых в компании стандартов и процедур. Работы по обеспечению качества ПП должны обеспечивать:

проверку выполнения требований заказчика;

взаимосвязь с заказчиком по вопросам качества ПП;

разработку и исполнение процедур, повышающих качество ПП;

повышение уровня квалификации разработчиков.

Каждая проектная группа должна иметь ответственного за обеспечение качества и составлять свой план по обеспечению качества. На основе планов отдельных проектных групп составляется план обеспечения качества всей компании, который устанавливает действия по повышению качества и ответственных за эти действия.

В процессе выполнения работ по созданию ПП, независимо от этапа разработки, регулярно собираются различные статистические данные (метрики), которые позволяют количественно оценивать ход выполнения работ и вовремя обнаруживать нарушения при их выполнении. Собранные метрики также позволяют в дальнейшем более точно спланировать работу над проектом, определить узкие места и принять меры к их устранению.

Метрики являются количественной оценкой степени соответствия организационного процесса по созданию ПП, отдельного проекта по созданию программного продукта или самого ПП некоторому определенному атрибуту.

В работе используются метрики продукта, проекта и процесса.

Метрики процесса применяются для отслеживания исполнения и совершенствования организационного процесса компании, метрики проекта — для отслеживания и улучшения работы над проектом, метрики продукта — для совершенствования качества ПП (подробнее см. гл 5).

Контрольные вопросы

1. Что понимают под моделью жизненного цикла разработки программного продукта?
2. Что представляет собой процесс создания программного продукта?
3. Что понимают под этапом в модели жизненного цикла разработки программного продукта?
4. Какие этапы входят обычно в состав жизненного цикла разработки программного продукта?
5. Какие основные модели жизненного цикла разработки программного продукта вы знаете?
6. В чем заключаются принципиальные отличия этих моделей?
7. Объясните и охарактеризуйте модель: а) V-образную; б) RAD-модель; в) многопроходную; г) прототипирования; д) каскадную; е) спиральную.

8. В чем отличие между моделями:
а) с промежуточным контролем и каскадной;
б) спиральной и каскадной;
в) традиционной спиральной и усовершенствованной, или измененной, спиральной?
9. Каковы основная цель и назначение процессов:
а) вспомогательных;
б) инспектирования программного продукта;
в) управления конфигурацией программного продукта;
г) обеспечения качества?
10. Перечислите и охарактеризуйте различные виды инспектирования.
11. Какова роль метрик в процессе разработки программного продукта?
12. Объясните назначение метрик процесса, проекта и продукта.

ГЛАВА 4

ОРГАНИЗАЦИЯ ПРОЦЕССА РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА

4.1. Кризис программирования и способ выхода из него

Программы не появляются сами по себе, даже основываясь на самых лучших методологиях и образцах. Программа — это результат деятельности людей, и от того, как организован труд этих людей, в огромной степени зависит качество разрабатываемого ПП.

В начале 1970-х годов в США был отмечен кризис программирования (software crisis). Это выражалось в том, что большие проекты стали выполняться с отставанием от графика или с превышением сметы расходов, разработанный ПП не обладал требуемыми функциональными возможностями, производительность его была низка, качество получаемого программного обеспечения не устраивало потребителей.

Аналитические исследования и обзоры, выполнявшиеся в последние годы ведущими зарубежными аналитиками, дают не слишком обнадеживающие результаты. Например, в 1995 г. компания Standish Group проанализировала работу 364 американских корпораций, а также итоги выполнения более 23 тыс. проектов, связанных с разработкой ПП, и сделала следующие выводы:

только 16,2 % проектов завершились в срок, не превысили запланированный бюджет и обеспечили реализацию всех требуемых функций и возможностей;

52,7 % проектов завершились с опозданием, расходы превысили запланированный бюджет, требуемые функции не были реализованы в полном объеме;

31,1 % проектов были аннулированы до завершения.

Для проектов, которые завершились с опозданием или были аннулированы до завершения, бюджет в среднем оказался превышен на 89 %, а срок выполнения — на 122 %.

В 1998 г. процентное соотношение проектов лишь немного изменилось в лучшую сторону (26, 46 и 28 % соответственно).

В числе причин возможных неудач фигурируют: нечеткая и неполная формулировка требований к ПП, недостаточное вовлечение пользователей в работу над проектом, отсутствие необходимых ресурсов, неудовлетворительное планирование, частое изме-

нение требований и спецификаций, новизна используемой технологии для организации, отсутствие грамотного управления проектом, недостаточная поддержка со стороны высшего руководства.

В последнее время ведущие зарубежные аналитики отмечают как одну из причин многих неудач тот факт, что большое число проектов выполняется в экстремальных условиях. В англоязычной литературе с легкой руки Эдварда Иордана, одного из ведущих мировых специалистов в области программной инженерии, утвердилось выражение «death march» — буквально «смертельный марш». Под ним понимается такой проект, параметры которого отклоняются от нормальных значений, по крайней мере, на 50 %. По отношению к проектам создания ПП это означает наличие, как минимум, одного из следующих ограничений:

план проекта сжат более чем на половину по сравнению с нормальным расчетным планом, т.е. работа, требующая в нормальных условиях 12 мес, должна быть выполнена за 6 мес или менее. Жесткая конкуренция на мировом рынке делает такую ситуацию наиболее распространенной;

число разработчиков уменьшено более чем на половину по сравнению с действительно необходимым для проекта данного масштаба, как правило, по причине сокращения штатов компании в результате кризиса, реорганизации и т.д.;

бюджет и связанные с ним ресурсы урезаны наполовину (результат сокращения компании и других противозатратных мер или конкурентной борьбы за выгодный контракт), что влечет за собой уменьшение числа нанимаемых разработчиков или привлечение малооплачиваемых неопытных молодых работников;

требования к функциям, производительности и другим техническим характеристикам вдвое превышают значения, которые они могли бы иметь в нормальных условиях.

Потребность контролировать процесс разработки ПП, прогнозировать и гарантировать стоимость разработки, сроки и качество результатов привела в конце 1970-х годов к необходимости перехода от кустарных к индустриальным способам создания ПП и появлению совокупности инженерных методов и средств создания ПП, объединенных общим названием «программная инженерия» (software engineering). Впервые этот термин прозвучал в качестве названия темы конференции, проводившейся под эгидой НАТО в 1968 г. Спустя семь лет, в 1975 г., в Вашингтоне была проведена первая международная конференция по программной инженерии. Тогда же появилось первое издание, посвященное программной инженерии, — «IEEE. Transactions on Software Engineering» («IEEE. Труды по программной инженерии»).

В процессе становления и развития программной инженерии можно выделить два этапа: 1970-е и 1980-е годы — систематиза-

ция и стандартизация процессов создания ПП (на основе структурного подхода); 1990-е годы — начало перехода к сборочному, индустриальному способу создания ПП (на основе объектно-ориентированного подхода).

В основе программной инженерии лежит одна фундаментальная идея: проектирование ПП является формальным процессом, который можно изучать и совершенствовать. Освоение и правильное применение методов и средств создания ПП позволяют повысить его качество, обеспечивают управляемость процесса проектирования ПП и увеличивают срок его жизни.

При любых изменениях в процессе разработки ПП необходимо иметь критерии, позволяющие оценить эффект от проведенных изменений. Другими словами, необходимы методы, позволяющие измерять качество и эффективность работы организации. Под организацией понимается совокупность всех групп разработчиков проектов (каждый проект направлен на создание своего ПП) и администрации, связанных одним процессом.

В настоящее время существуют две общепринятые методики оценки состояния процесса разработки программного обеспечения: международный стандарт ISO 9001 и модель CMM-SEI — Capability Maturity Model (модель оценки зрелости технологических процессов в организации), разработанная в Software Engineering Institute (Институт программной инженерии).

4.2. Модель CMM-SEI

В 1986 г. институт SEI (подразделение университета Карнеги—Меллона) с помощью корпорации Mitre начал разработку основ модели эффективного процесса изготовления программ. Характеризуя такой процесс, авторы модели употребляют понятие «зрелость», которое означает не только эффективность, но и устойчивость, надежность процесса.

Первоначальная версия модели, которая получила название «CMM», была выпущена в конце 1987 г. После этого модель несколько раз перерабатывалась, и в настоящее время готовится очередная ее версия.

В соответствии с этой моделью организация может находиться на одном из пяти уровней зрелости (рис. 4.1).

Для первого, или *начального*, уровня характерен спонтанный и иногда хаотический процесс разработки программ. Процедуры разработки не определены, и успех зависит от индивидуальных усилий работников.

Второй уровень, названный *повторяемым*, характеризуется тем, что на нем используются основные процессы управления, позволяющие отслеживать как функциональные характеристики разрабатываемого ПП, так и график работ, а также их стоимость. Организация способна повторить успешную разработку нового проекта с аналогичными возможностями.

Третий уровень называется *определенным*. Организации, находящиеся на этом уровне, документируют, стандартизуют и интегрируют в общий процесс управления все управленческие и инженерные задачи разработки ПП, тем самым вырабатывая стандартный процесс организации. Все проекты в организации используют утвержденные методы разработки и поддержки программ, адаптированные к конкретному проекту.

Рис. 4.1. Уровни зрелости организации

Четвертый уровень называется *управляемым*. У организаций этого уровня имеются способы детального измерения качества процесса и разрабатываемого ПП. Количественные характеристики процесса разработки и разрабатываемых ПП хорошо изучены и управляемы, процесс предсказуем.

Пятый, высший уровень зрелости организации называется *оптимизированным*. На нем осуществляется непрерывное улучшение процесса разработки, основанное на количественных характеристиках выполненных и выполняемых проектов и на внедрении новых идей и технологий.

Повышение зрелости идет от первого к пятому уровню. Подавляющее большинство организаций, занимающихся разработкой ПП, начинают с первого уровня (многие на нем и остаются). Незрелая организация не может предсказать ни срока завершения разработки, ни качества конечного продукта. Успех зависит от опытности непосредственного руководителя и квалификации и таланта разработчиков. Процесс управления, если и создается, то только в ходе выполнения проекта, ему не следуют, особенно во время часто случающихся кризисов и авралов.

Не стоит думать, что такая организация не способна успешно разработать ПП, хотя, скорее всего, график разработки не будет выполняться и система будет стоить больше, чем планировалось

сначала. Повторение успеха возможно только в том случае, если в разработке принимают участие те же самые сотрудники.

Для достижения второго и более высоких уровней организации должны разработать свой процесс, основанный и включающий в себя определенные рекомендуемые ключевые процессы.

Ключевые процессы на оптимизированном уровне:

управление изменениями процесса;

использование современных новейших технологий;

предотвращение ошибок.

Ключевые процессы на управляемом уровне:

управление качеством процесса;

измерение и анализ процесса.

Ключевые процессы на определенном уровне:

рецензирование и обсуждение с коллегами результатов работы;

координация и взаимодействие между проектами;

повышение квалификации сотрудников;

определение организационных процессов;

сосредоточение особого внимания на организационных процессах;

индустриальный подход к проектированию и разработке ПП;

интегрированное управление всеми проектами, базирующееся на стандартном процессе организации.

Ключевые процессы на повторяемом уровне:

управление конфигурацией (версиями) ПП;

обеспечение качества ПП;

управление работой субподрядчиков;

контроль за выполнением программного проекта;

планирование программного проекта;

управление требованиями к ПП.

На начальном уровне ключевых процессов нет.

Любой ключевой процесс подразумевает выполнение организацией некоторого набора ключевых действий, связанных с этим процессом, что, в свою очередь, позволяет организации достичь определенных целей, реализуемых данным ключевым процессом. Ключевые действия не только дают возможность реализовать данный ключевой процесс, но и являются своего рода инструкцией к его реализации. Выполнение или невыполнение различных ключевых действий является своеобразным ключевым показателем, по которому можно судить об уровне зрелости организации. Для определения этого уровня в модели CMM предусмотрен целый перечень вопросов, которые могут быть заданы сотрудникам организации. Анализ ответов на эти вопросы позволяет сделать вывод о достигнутом уровне зрелости. Пример структуры CMM для ключевого процесса «Планирование программного проекта» второго уровня зрелости организации приведен на рис. 4.2.

Рис. 4.2. Пример структуры CMM

Модель CMM специально обходит вопрос о подборе персонала. Она рассматривает разработку ПП как чисто производственный процесс. Организационная структура и процессы управления должны помогать наиболее эффективно проявляться способностям разработчиков.

Сами процессы в различных организациях могут быть различны, как и используемые методы и технологии. Общим требованием ко всем процессам является создание основы для системы управления разработкой ПП.

4.3. Управление качеством разработки программного продукта с помощью системы стандартов ISO 9001

Международная организация по стандартизации (МОС) разработала систему стандартов ISO 9001, которые регламентируют

вопросы управления качеством. Эти стандарты применимы практически ко всем областям производства товаров и услуг, в частности, и к ПП.

Взаимосвязь между моделью CMM и системой стандартов ISO 9001 следующая: ISO 9001 содержит перечень требований, а CMM определяет детали требований к процессу разработки для включения их в документы по управлению качеством. Россия, являясь членом МОС, приняла систему стандартов ISO 9001 как свой национальный стандарт. ВНИИ стандартизации выпустил его перевод на русский язык.

Целью ISO 9001 является построение системы сквозного управления качеством (TQM — Total Quality Management), которая должна обеспечивать качество на всех этапах разработки.

В ISO 9001 и CMM-SEI приведены процедуры сертификации организаций на соответствие указанным системе стандартов и модели. Пройти такую сертификацию достаточно сложно, однако многие компании стремятся сделать это. Официальная сертификация на соответствие системе стандартов и модели дает существенные конкурентные преимущества перед конкурентами. При выборе исполнителя заказчик будет уверен в высоком качестве выполнения работ. Некоторые крупные компании при проведении конкурсов и тендеров на выполнение работ требуют предоставить информацию о том, в какой степени участвующие в конкурсе компании соответствуют системе стандартов ISO 9001 и модели CMM-SEI.

Система стандартов ISO 9001 определяет минимальный набор требований к управлению качеством. Условно этот набор разбивается на три части: требования к менеджменту компании, контролю продукции и процессу разработки.

Эффективная система качества невозможна, если менеджмент компании не осознает ее значения и не ставит цель ее построить. От руководства компании требуется подписать формальное письмо, подтверждающее приверженность поддержанию высокого качества продуктов и услуг. В письме указываются основные документы, на которые следует опираться при контроле качества. Руководство компании создает подразделение компании, обеспечивающее контроль качества. Кроме того, определяются процедуры периодических проверок и обсуждений эффективности системы управления качеством.

Управление продукцией включает в себя контроль за версиями систем, приобретением готовых пакетов и программ, продукцией, которая в настоящий момент не отвечает требованиям качества. Многие положения стандарта в разделе управления продукцией не относятся к программному обеспечению или являются второстепенными (например, положения, относящиеся к упаковке и хранению).

Управление процессом разработки — важнейшая часть ISO 9001 для организаций, занятых программированием. Она включает в себя требования к построению и документированию всего процесса разработки ПП — от заключения контракта до распространения готового продукта (на этом этапе управление разработкой переходит в управление продукцией, упоминавшееся ранее).

По классификации ISO 9001 разработка программ относится к так называемым специальным процессам, т.е. процессам, дефекты продукта которых могут быть незаметны до тех пор, пока им не начнут пользоваться.

Система стандартов ISO 9001 не регламентирует сам процесс разработки, который может быть совершенно разным в различных организациях. Она стандартизует критерии соответствия процесса требованиям сквозного контроля качества. Первым необходимым условием является наличие документации, регламентирующей конкретный процесс в конкретной организации.

4.4. Примерная структура процесса и организации, занимающейся разработкой программных продуктов

Для организации предсказуемого и управляемого процесса компании необходимы организационные, технические и нетехнические средства (рис. 4.3).

Организационные средства включают в себя определенный перечень различных должностей и иерархию подчинения сотрудников вышестоящему руководству.

Общее управление работой компании выполняет генеральный директор. Вопросы, связанные с ходом выполнения различных проектов, курирует исполнительный директор, а вопросы, связанные с организацией и обеспечением процесса компании (т.е. свода правил, процедур, рекомендаций и других руководящих документов, в соответствии с которыми компания действует) и работы по обеспечению качества ПП, — заместитель генерального директора. Такое распределение работ лишний раз подчеркивает важность создания в компании процесса и проведения работ по обеспечению качества ПП.

При необходимости вместо двух групп (группы процесса и группы обеспечения качества) в компании может быть только одна группа процесса, но при этом она должна также выполнять все действия по обеспечению качества ПП. Кроме этого, в каждом проекте должен быть выбран ответственный за качество ПП. Обычно это руководитель проекта или один из ведущих инженеров. Ответственный за качество является представителем групп процесса и обеспечения качества (если эти группы существуют самостоятельно) в своем проекте и отвечает за выполнение всех дей-

Рис. 4.3. Примерная структура процесса и организации, занимающейся разработкой программных продуктов

ствий, связанных с процессом компании и обеспечением качества.

Независимый тестировщик, как видно из рис. 4.3, участвует в работе над проектом, но не зависит от руководителя проекта. Это позволяет проводить независимое объективное тестирование документации и ПП, разрабатываемого в данном проекте. Часто бывает так, что тестировщик одновременно принимает участие в нескольких проектах, особенно если их текущие этапы не совпадают. Нередко создают отдельную группу тестирования, куда входят все тестировщики компании.

Технические средства предназначены для организации соответствующих условий работы над проектами и поддержанию процесса компании, а также работ по обеспечению качества программного продукта. Например, автоматизированное рабочее место (АРМ) программиста позволяет повысить производительность его работ и качество разрабатываемого ПП, а компьютерная сеть — обеспечить электронный документооборот в компании и связь между сотрудниками. База данных дает возможность хранить всю информацию, связанную с ходом выполнения как текущих проектов, так и выполненных ранее.

Нетехнические средства включают в себя разработанные или принятые к использованию стандарты и планы, а также книгу процесса, которая содержит подробное описание процесса компании. По метрикам процесса оценивают его основные характеристики (ключевые процессы) и результаты оценки заносят в паспорт процесса. Этот паспорт позволяет отслеживать соблюдение процесса, а также планировать действия по его совершенствованию.

Контрольные вопросы

1. Чем был вызван кризис программирования?
2. Какой существует выход из кризиса программирования?
3. Какой проект можно определить как «смертельный марш»?
4. Какие методики оценки состояния процесса разработки программного продукта вам известны?
5. Перечислите и охарактеризуйте основные уровни зрелости организации согласно модели CMM.
6. Какие ключевые процессы должны выполняться на каждом из пяти уровней модели CMM?
7. Какова структура CMM?
8. Определите цель и назначение системы стандартов ISO 9001.
9. Перечислите и охарактеризуйте минимальный набор требований к управлению качеством согласно системе стандартов ISO 9001.
10. Какие требования предъявляются к управлению: а) компанией; б) продукцией; в) разработкой?
11. Объясните примерную структуру процесса и организации, занимающейся разработкой программных продуктов.
12. Что включают в себя средства: а) организационные; б) технические; в) нетехнические?

ГЛАВА 5

МЕТРИКИ

5.1. Роль метрик в процессе разработки программных продуктов

Измерения, выполняемые в процессе разработки ПП, помогают лучше оценить сам процесс разработки, принятый в организации, ход выполнения проекта и качество ПП. Измерения процесса производятся в целях его дальнейшего совершенствования, измерения проекта — для улучшения организации работ, а измерения ПП — для повышения его качества. В результате измерения определяется количественная характеристика какого-либо свойства объекта измерения. Путем непосредственных измерений могут определяться только опорные свойства объекта — опорные метрики. Все остальные метрики оцениваются в результате вычисления тех или иных функций от значений опорных метрик. Эти вычисления проводятся по соответствующим формулам.

В издании «IEEE. Standard Glossary of Software Engineering Terms» («IEEE. Перечень стандартных терминов, используемых в программной инженерии») метрика определена как мера степени обладания свойством, имеющая числовое значение.

Под метриками понимают количественную оценку ПП, процесса или проекта, которая используется непосредственно либо на основе которой производятся другие измерения или выполняется прогноз.

Д-р Барри У. Боэм (Dr. Barry W. Boehm), всемирно признанный эксперт в области разработки ПП, подчеркивает, что важность метрик определяется тем, в какой мере они способствуют принятию решений. Если руководитель программного проекта будет помнить об этом, то он сможет оперировать полезными и важными метриками, а не собирать их случайным образом, накапливая большие объемы информации, использование которой затруднительно.

«*Измерение при разработке программного продукта является непрерывным процессом определения, сбора и анализа данных, относящихся к программному процессу и соответствующим ему продуктам. Целью этой деятельности является получение представления о процессе, контроль над ним и программными продуктами, а также*

поддержка важной информации, которая позволит совершенствовать процесс и программные продукты» [5].

«*Измерение в ходе разработки программного продукта — количественное оценивание произвольных аспектов процесса программного инжиниринга, программного продукта или контекста; оно служит для совершенствования представления, помогает контролировать, прогнозировать и вносить улучшения в создаваемый продукт, а также в применяемые рабочие методы*» [7].

Все метрики можно разделить на три основные группы: метрики процесса; метрики проекта; метрики продукта.

Внутри каждой группы существуют следующие типы метрик: непосредственно наблюдаемые (измеряемые); прогнозируемые; вычисляемые.

Непосредственное наблюдение атрибута какого-либо объекта не требует использования в процессе измерения других атрибутов или объектов. Непосредственное наблюдение или измерение применяется при оценивании существующего объекта.

При прогнозировании используется математическая модель выбранного атрибута наравне с набором процедур прогнозирования, применяемых для определения неизвестных параметров и интерпретации результатов.

Вычисление, или косвенное измерение, означает вовлечение в процесс измерения с помощью определенной математической модели других атрибутов и объектов (всегда включает в себя вычисления с использованием, по крайней мере, двух других метрик).

В табл. 5.1 приведены примеры типов метрик, относящихся к ПП, процессу и проекту.

Метрики бывают объективными и субъективными. Субъективные измерения предполагают наличие личностного, субъективного подхода, например применение какого-либо весового коэффициента.

Измеряемые атрибуты могут быть внешними и внутренними. Внутренние атрибуты могут измеряться в терминах самого объекта, отдельно от его поведения. Внешние атрибуты оцениваются с учетом связи объекта с внешней средой. Примерами внутренних атрибутов являются показатель числа строк кода в программном продукте (LOC), продолжительность выполнения действия, величина трудозатрат, число неудачных тестовых испытаний, объем денежных затрат, уровень сложности и степень модульности. В качестве внешних атрибутов можно рассматривать время выполнения (требуются программа и компьютер), полезность и удобство представления (требуются приложение и пользователь), надежность, эффективность, тестируемость, повторную применимость, переносимость и взаимодействие между операциями.

Таблица 5.1

Примеры метрик процесса, проекта и продукта

Группа метрик	Примеры метрик		
	непосредственно наблюдаемых	прогнозируемых	вычисляемых
Метрики процесса	Обращение к процессу; охват и эффективность процесса; эффективность обучения персонала	Уровень CMM-SEI	Точность оценивания (определяется отношением предварительной оценки к действительному значению); производительность (определяется отношением числа строк кода (LOC), написанных разработчиками за месяц, к общему числу разработчиков); охват отчетом; тестирование или верификация покрытия; эффективность преподавания; вероятность риска; обзор метрических показателей; анализ требований; поддержка метрических показателей
Метрики проекта	Время разработки ПП; стоимость проекта; потребность в ресурсах	Продолжительность выполнения проекта; стоимость проекта	Обращение к графику (определяется отношением объема освоенных средств к объему запланированных средств, например отношением действительной стоимости выполненных работ к их бюджетной стоимости)
Метрики программного продукта	Число строк кода; число тестовых примеров; число неустраненных дефектов; число тестов; число дефектов, их серьезность; число компонентов системы	Число строк кода; качество программного продукта; проявление дефектов программного продукта; представление и характеристики оставшихся дефектов; надежность; число дефектов, их серьезность	Эффективность — динамическое поведение системы; эффективность — ресурсы системы; плотность дефектов (определяется отношением числа найденных дефектов к объему программного продукта, выраженному в KLOC, где 1 KLOC = 1000 LOC); измерение достигнутой степени надежности (например, значение времени наработки на отказ)

Измерения в программных проектах призваны помочь руководителям разного уровня, а также самим разработчикам принять своевременные и правильные решения в зависимости от рассматриваемых данных. Кроме того, они облегчают отслеживание прогресса в организации работ при внедрении усовершенствований в процесс, дают возможность оценить воздействия введенных прогрессивных изменений. Программные метрические показатели используются для измерения определенных атрибутов ПП, процесса или проекта. Руководитель проекта может также применять их:

для анализа ошибок и дефектов ПП;

оценки состояния ПП;

определения уровня сложности ПП;

установки основных направлений разработки;

экспериментального подтверждения лучших методик;

прогнозирования качественных показателей, графика, трудозатрат и объемов других затрат;

отслеживания прогресса в ходе выполнения проекта;

определения оптимальных сроков достижения необходимого уровня качества продукта либо процесса в целом.

Собираемые метрики позволяют улучшить и ускорить процесс принятия решений, и в результате каждый из тех, кто непосредственно выполняет измерения или пользуется результатами измерений, только выигрывает. Это может быть любой пользователь или группа лиц, интересы которых связаны с успешным завершением проекта (заказчики, спонсоры, конечные пользователи, главные менеджеры организации, руководители проек-

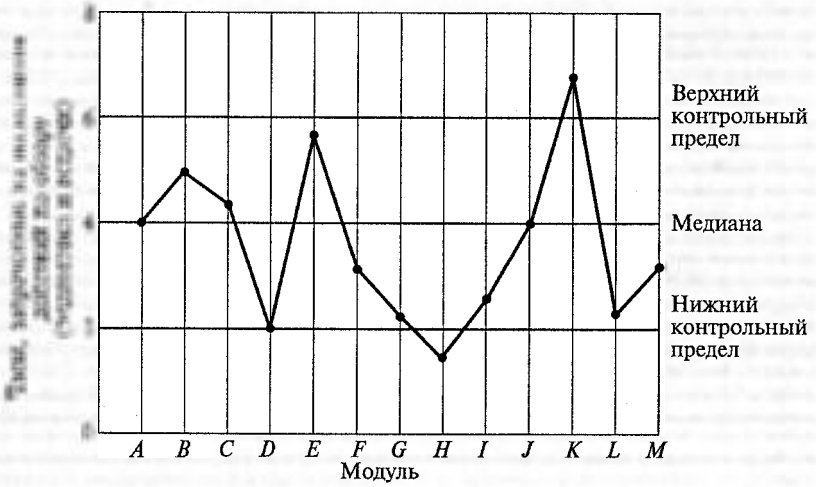


Рис. 5.1. Пример контрольного графика, применяемого командой разработчиков

та, эксперты предметной области, авторы, аналитики, разработчики, инженеры, обеспечивающие качество ПП, программисты и тестировщики).

В то время как менеджеры используют метрические параметры и определении методов управления проектами, а также при енивании изменений по улучшению процесса разработки ПП, манды разработчиков применяют их для решения следующих задач:

установление достижимых целей при анализе требований, на фазах проектирования, конструирования, тестирования и внедрения;

демонстрация потенциала при достижении этих целей;

отслеживание прогресса в достижении этих целей;

выполнение настройки процессов для коррекции предельных условий, выходящих из-под контроля (например, если обзоры являются достаточно важными, но команда не желает затрачивать них дополнительное время, то их можно отслеживать с помо- ю контрольного графика, как показано на рис. 5.1).

Заинтересованными сторонами выполняются сбор, синтез, анализ, включение в отчеты и изучение метрик. Затем происходит их постепенная сортировка по убыванию приоритета в рамках данной организации или с учетом особенностей конкретного проекта. Метрики могут не подходить для использования в угой организации. Однако после того как они получили подерждение в процессе работы и внесены в базу данных проекта и организации, их ценность многократно возрастает. Они бут играть существенную роль при прогнозировании будущих правлений и внесении усовершенствований в процесс организации или разработки.

5.2. Метрики и модель CMM-SEI

5.2.1. Второй, повторяемый, уровень модели CMM-SEI

Измерение и анализ результатов измерений являются составй частью модели зрелости программного обеспечения, предлоенной институтом SEI (см. подразд. 4.2). Каждый ключевой прос в модели CMM-SEI на любом уровне зрелости содержит мпоненты измерения и анализа. Измерение и анализ результатв измерения необходимы для уточнения состояния процесса, оекта и продукта. Действия по сбору и анализу метрик в соотстствии с уровнями модели CMM-SEI приведены в табл. 5.2.

Ключевой процесс: управление требованиями к ПП. Измерения полняются для определения действий, предпринимаемых с ю управления требованиями. Примеры метрик:

Таблица 5.2

Действия по сбору и анализу метрик на различных уровнях зрелости модели CMM-SEI

Уровень зрелости	Действия
1	Сбор данных и их анализ
2	Планирование и управление данными, используемыми в отдельных проектах
3	Применение собранных данных во всех определенных процессах. Систематическое использование данных совместно в различных проектах
4	Стандартизированные во всей организации определение и сбор данных. Использование данных для представления количественных параметров процесса и его стабилизации
5	Использование данных для оценивания и выделения усовершенствований в процессе

состояние каждого из сформулированных требований;

действия по изменению сформулированных требований;

накапливающийся итог по количеству изменений в сформулированных требованиях, включая общее число требований, среди них — предложенных, открытых, одобренных и включенных в базовую версию.

Ключевой процесс: планирование программного проекта. Измерения производятся с целью уточнения различных действий, связанных с планированием. Примеры метрик:

даты завершения основополагающих стадий для различных действий в соответствии с планированием проекта;

трудозатраты, определяемые после завершения работ; распределение средств по видам действий, связанным с планированием программного проекта.

Ключевой процесс: контроль за выполнением программного проекта. Измерения производятся с целью определения состояния процесса контроля за выполнением программного проекта.

Примеры метрик:

трудозатраты, затраты иных ресурсов при выполнении контроля;

число изменений при планировании проектов, включая изменения в оценках размеров разрабатываемых ПП, стоимости программных работ, критического объема вычислительных ресурсов и графика работ.

5.2.2. Третий, определенный, уровень модели CMM-SEI

Ключевой процесс: повышение квалификации сотрудников. Измерения производятся с целью определения состояния различных форм деятельности по реализации программы обучения сотрудников. Примеры метрик:

реальная посещаемость каждого курса обучения в сравнении с планируемой посещаемостью;

прогресс для организации и проекта, достигнутый благодаря проведенным курсам обучения;

число сбоев при проведении обучения.

Кроме того, измерения производятся для определения качества учебной программы. Примеры метрик:

результаты тестирования, проведенного после обучения;

обзоры курсов на основе студенческих опросов;

поддержка обратной связи с руководителями проектов.

Ключевой процесс: индустриальный подход к проектированию и разработке ПП. Измерения производятся с целью определения функциональных возможностей и качества ПП. Примеры метрик:

число, типы и серьезность дефектов, идентифицированных в ПП;

выяснение, классифицируются ли требования в соответствии с категориями (например, безопасность, системная конфигурация, выполнение и надежность), с учетом требований к ПП и системных тестовых испытаний.

Кроме того, измерения производятся для определения состояния различных форм деятельности по разработке ПП. Примеры метрик:

состояние каждого требования на протяжении выполнения проекта;

отчеты об имеющихся проблемах с указанием степени их важности и затрат времени;

трудозатраты на анализ предложенных изменений для каждого рассматриваемого изменения и кумулятивная оценка по всем изменениям;

объем изменений, внесенных в базовую версию ПП по категориям (например, интерфейс, безопасность, системная конфигурация, выполнение и практичность);

объем затрат при реализации и тестировании внесенных изменений, включая начальную оценку и действительное соотношение объем/затраты.

5.2.3. Четвертый, управляемый, уровень модели SEI CMM

Ключевой процесс: управление качеством процесса. Данный ключевой процесс включает в себя действия по определению целей,

измерения производительности процесса, анализ этих измерений, а также настройку и функционирование процесса в рамках допустимых ограничений. Если при выполнении процесса произойдет его стабилизация в пределах допустимых ограничений, то определенный проектом процесс, ассоциированные с ним измерения и допустимые ограничения для измерений утверждаются и применяются в качестве базы для осуществления количественного контроля над выполнением процесса.

Ключевой процесс: измерение и анализ процесса. Измерение и анализ выполняются для контроля текущего состояния процесса и его улучшения на основе полученных данных.

Примеры метрик:

точность оценивания данных о размере и стоимости ПП;

производительность при выполнении различного вида работ;

качественные показатели в соответствии с планом управления качеством;

эффективность обучения сотрудников;

тестовое покрытие и эффективность тестов;

меры по обеспечению надежности программ;

число и серьезность недостатков, обнаруженных в требованиях к ПП;

число и серьезность дефектов, обнаруженных в программном коде.

Примеры тенденций, определяющих возможности ПП:

прогнозирование появления программных дефектов и сравнение прогнозов с реальными данными;

прогнозирование распределений дефектов и их характеристик, если речь идет о дефектах, оставшихся в продукте. В качестве основы применяются данные экспертных оценок и/или тестирования.

Метрики должны быть практичными, простыми и легкими для представления, недорогими, понятными, последовательными и применяемыми на протяжении всего периода работы, удобными с точки зрения сбора и обработки данных, легкодоступными для всех заинтересованных пользователей.

Собираемые данные должны быть корректны (собраны в соответствии с правилами определения данного показателя), точными, четкими и непротиворечивыми (отсутствуют серьезные отличия для отображаемых величин даже в том случае, если заменяется измерительное устройство или специалист, производящий измерения). Большая часть данных ассоциируется с определенным действием или периодом времени. Они должны содержать аннотации и указания даты и времени с целью ознакомления всех желающих с точным местом и временем их сбора. Процедура измерения должна быть четко описана, чтобы любой исполнитель мог повторно выполнить необходимые измерения.

5.3. Парадигма Бейзили

5.3.1. Общее описание парадигмы

Парадигма д-ра Виктора Бейзили (Dr. Victor Basili) «цель, вопрос, метрика» («Goal, Question, Metric», GQM) является широко применяемым и хорошо зарекомендовавшим себя подходом к определению метрик, наиболее подходящим для контроля над

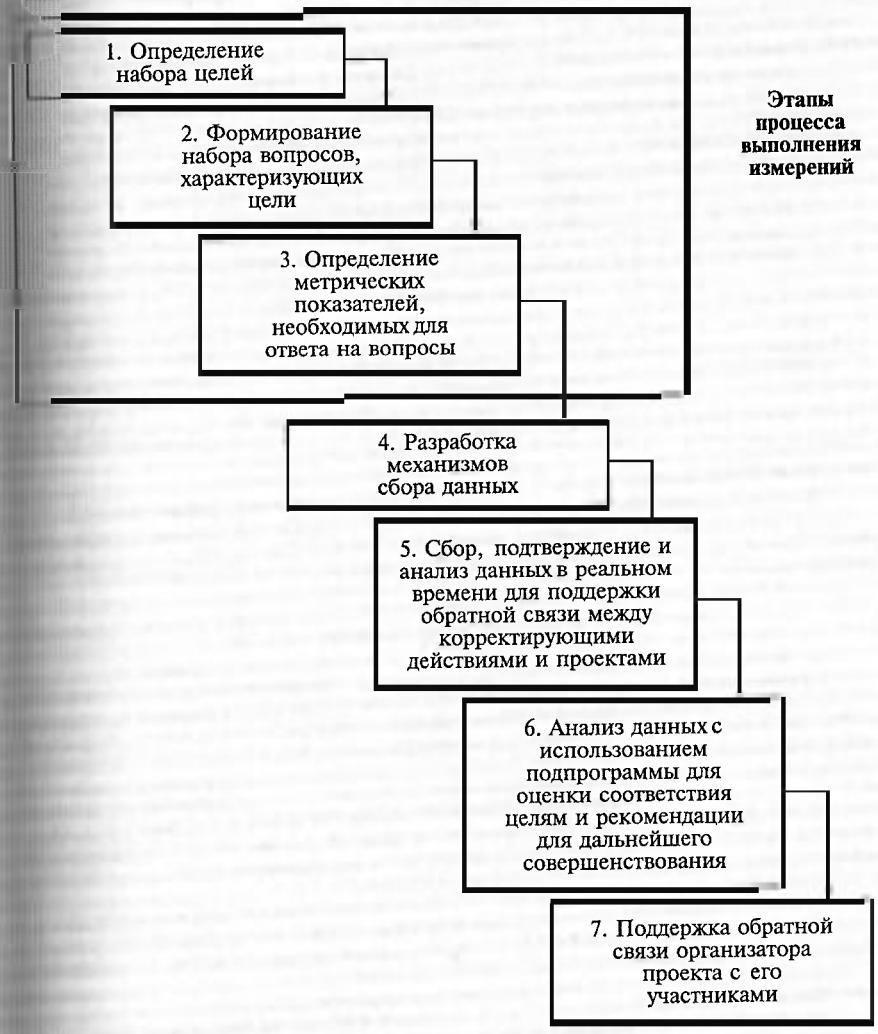


Рис. 5.2. Процесс Бейзили

проектом и формирования решения. Подход с применением GQM предполагает, что цель устанавливается еще до выбора метрик. В.Бейзили разработал методику, в основе которой лежит процесс, состоящий из семи этапов. Благодаря этому появилась возможность систематического применения измерений при разработке ПП. На рис. 5.2 представлены все этапы парадигмы. Первые три из них имеют определяющее значение для работы в рамках парадигмы GQM.

Этап 1. Определение набора целей. Под этим понимается формирование набора корпоративных целей, относящихся к работе подразделения, или проектных целей, связанных с разработкой, сопровождением, повышением производительности и качественных показателей продукта или процесса (например, удовлетворение запросов заказчиков, своевременная поставка, улучшение качества, создание повторно используемых объектов, заново применяемые опытные наработки). Часто определение целей выполняется при решении проблем методом «мозгового штурма», а также в процессе получения требований от каждого из заинтересованных лиц. Эти цели можно располагать с учетом приоритетов, а также группировать их по областям усовершенствований ПП.

Этап 2. Формирование набора вопросов, характеризующих цели. Каждая цель позволяет сформулировать вопрос, ответ на который определяет, будет ли эта цель достигнута. Данные вопросы характеризуют цели, позволяют оценить прогресс в их достижении, прогнозируют момент достижения или являются мотивацией для оценки продвижения к цели.

Этап 3. Определение метрических показателей, необходимых для ответа на вопросы. На этом этапе определяют, что необходимо измерить для получения адекватного ответа на поставленные вопросы.

5.3.2. Этап 1 GQM: определение набора целей

Чтобы определять цели, вопросы и метрические показатели, был разработан шаблон (табл. 5.3), который применяется для записи в него цели, перспективы и среды и характеризует структуру цели.

Цель — представление, оценивание, прогнозирование, мотивация процесса (или продукта, модели, метрики и т.д.), позволяющие его (ее) представить, оценить, проконтролировать, исследовать, изучить, усовершенствовать и т.д.

Представление или *характеристика* предполагает формирование модели объекта, в которой удобно представить собранные данные.

Оценивание подразумевает наличие шаблона для данных, который позволяет разрабатывать постоянную модель, основанную на доступных (или корректно оцениваемых) факторах.

Таблица 5.3

Шаблон для определения целей, вопросов и метрических показателей

Анализ (что подвергается анализу)		Объект, который измеряется, процесс, продукт, модель, метрика и т.д.
Цель	Скакой целью (зачем)?	Характеристика, оценивание, прогнозирование, мотивация, совершенствование, представление, оценивание, управление, исследование, изучение или контроль
	По отношению к чему (фокус)?	Концентрация на качестве объекта (на чем сфокусировано измерение) — объем затрат, эффективность, корректность, устранение дефектов, изменения, измерения продукта, надежность, дружественность по отношению к пользователю и т.д.
Перспектива	Перспективы (каков аспект и кто исполнитель?)	Пользователи, проводящие измерения на объекте: разработчики, заказчики, руководитель проекта, и т.д.
Среда	Контекст (характеристики)	Среда, где выполняются измерения: факторы ресурса, процесса, человеческие факторы, проблемы, методы, инструменты, ограничения и т.д.

Мотивация или *совершенствование* предполагает наличие точной модели, позволяющей правильно представить объект или позитивное качество.

Пример: оценивание процесса сопровождения с целью его улучшения.

Перспектива — проверка объема затрат, эффективности, корректности, наличия дефектов, изменений, измерений продукта с точки зрения разработчика, менеджера, заказчика и т.д.

Всегда важно учитывать такие аспекты, как доступность информации, уровень ее «дробления» и точность.

Пример: проверка объема затрат с точки зрения руководителя проекта.

Среда — определяет контекст, в котором проходит изучение. Благодаря этому облегчаются классификация текущего проекта с учетом большого разнообразия характеристик, а также выделение подходящей среды для этого проекта, обеспечивается обнаружение класса проектов, имеющих аналогичные характеристики и цели, что позволяет использовать их для сравнения. Среда включает в себя процесс, персонал, проблемные факторы, методы, инструменты и ограничения.

К проблемным факторам относятся человеческие факторы, а также факторы процесса, продукта и ресурса.

Человеческие факторы — число пользователей, уровень экспертизы программного инженеринга, вопросы организации групп, опыт в предметной области разработки приложений, опыт выполнения процесса и экспертиза инструментария.

Факторы процесса — выбор модели жизненного цикла для проектирования, методов, методик, инструментов и языков программирования.

Факторы продукта — внутренние поставки, размер системы и требуемый уровень качества (например, надежность, переносимость).

Факторы ресурса основаны на аналогиях в целевых механизмах и механизмах проектирования, календарном времени, объемах бюджетных средств и существующего программного кода, доступного для повторного использования.

Примером полностью сформированной цели является следующая формулировка: «Проанализируйте и оцените с точки зрения команды разработчиков проекта поставленный продукт с учетом сложности интерфейса и в контексте объектно-ориентированной разработки, которая применяется в проекте».

Цели также следует подвергать измерениям и руководствоваться моделями, принятыми в бизнесе.

Кроме парадигмы GQM для уточнения измеряемых целей служат и другие механизмы, например функция развертывания качества (QFD — Quality Function Deployment) и методика с применением метрик, используемых для измерения качества ПП (SQM — Software Quality Metrics).

Как указано в GQM, цели определяются для каждого объекта, для разных условий, в соответствии с различными моделями качества, с применением разных точек зрения и с учетом определенных характеристик среды.

Стратегические цели обычно устанавливаются в процессе управления. К ним относятся: поддержка основных направлений во время проведения измерений, обеспечение скоростной связи, предоставление возможности «взгляда изнутри» на процесс разработки; формирование на исторической базе плана будущих проектов.

Примером метрики, которая используется для определения продвижения по направлению к стратегической цели, является уровень зрелости модели CMM-SEI.

Типичными *тактическими целями* команды разработчиков и организаций, занимающихся разработкой ПП являются: минимизация усилий по реализации программного инженеринга, сокращение объема затрат; максимальное удовлетворение запросов заказчиков; минимизация возможности появления дефектов; управление качественными показателями и проведение тестирования.

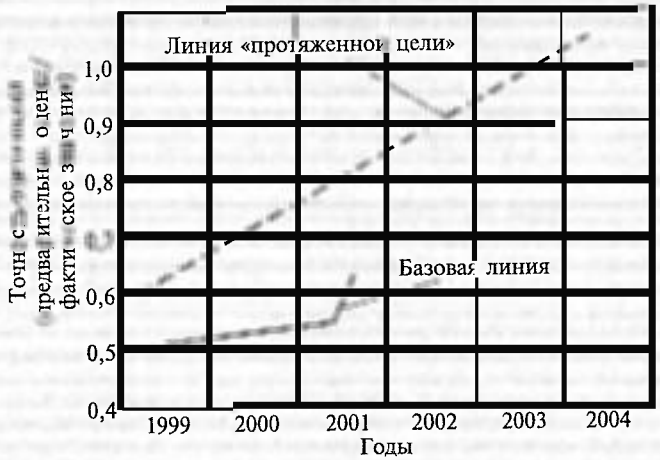


Рис. 5.3. Пример изображения «протяженной цели»

Примеры метрик для определения прогресса в продвижении к тактической цели по минимизации дефектов и управлению качеством тестирования:

среднее время фиксации служебного запроса;

суммарный объем обнаруженных дефектов;

время фиксации дефектов, соотношение оценочного и реального времени;

идентифицированные дефекты, остающиеся открытыми;

распределение источников дефектов;

среднее время между появлениями дефектов.

Как тактические, так и стратегические цели можно изобразить в виде линии «протяженной цели» на графике, где отображается достигнутый количественный уровень. Прогресс при движении по направлению к этой цели затем изображается графически на том же самом чертеже, как показано на рис. 5.3.

После установления целей процесс GQM смещается в сторону определения способов измерения прогресса, характеризующего продвижение к цели.

5.3.3. Этап 2 GQM: формирование набора вопросов, характеризующих цели

Цели определяются на абстрактном уровне, а вопросы предназначаются для освещения уровня выполняемых операций. Отвечая на вопросы, необходимо уточнять, достигнута ли поставленная цель. Например, если цель процесса формулируется следую-

щим образом: «Совершенствовать способность к реагированию на проблемы заказчиков, сокращая циклическое время, предназначенное для анализа и коррекции этих проблем», то могут возникнуть следующие вопросы.

Отражен ли в документах процесс, позволяющий устранить проблемы?

Производились ли в процессе устранения проблемы обзоры и выполнялось ли тестирование?

Точны ли оценки, используемые при устранении проблемы?

Применяется ли механизм по изменению формы контроля, с которым знакомы все разработчики?

Какова фактическая продолжительность процесса обнаружения проблемы?

Какова фактическая продолжительность процесса устранения проблемы?

Каков в среднем объем трудозатрат на этапе обнаружения и устранения проблемы?

Какой процент составляют устраненные проблемы от общего числа проблем, выявленных заказчиком?

Реализация цели, направленной на усовершенствование текущей версии ПП, также может способствовать появлению отдельных вопросов. Например, цель «Упростить сложную программную систему» может инициировать появление следующих вопросов.

Каково число имеющихся программных модулей?

Какова степень соединения модулей?

Сколько времени занимает разработка модуля?

Оценивались ли возможности каждого модуля?

Каково число ошибок, обнаруженных в каждом модуле?

Каков размер каждого модуля?

Связаны ли между собой модули?

Можно ли проследить реализацию требований в модулях?

В процессе управления разработкой ПП могут возникнуть следующие принципиальные вопросы.

Можно ли выполнить это действие?

Сколько времени займет этот процесс?

Каков объем затрат?

Сколько сотрудников следует привлечь к работе?

Какова степень риска?

Каковы возможные компромиссы?

Сколько ошибок может возникнуть?

Можно ли измерить степень улучшения процесса?

После уточнения спектра вопросов, которые позволят уточнить продвижение к намеченным целям, на следующем этапе процесса GQM определяют метрические показатели, применяемые для ответа на эти вопросы.

5.3.4. Этап 3 GQM: определение метрических показателей, необходимых для ответа на вопросы

Данный этап предназначен для определения метрик, необходимых для получения ответа на поставленные вопросы, а также для отслеживания соответствия процесса и продукта поставленным целям. Обычно необходимые метрики можно легко получить, воспользовавшись ключевыми словами, содержащимися в вопросе. Например, слова «усредненные приложенные усилия» в вопросе: «Каковы усредненные приложенные усилия, необходимые для устранения проблем, отмеченных в отчетах заказчиков» указывают на очевидный метрический показатель, применяемый для оценки трудозатрат, — трудозатраты, выраженные в рабочих неделях/месяцах. Кроме того, в качестве метрики целесообразно использовать число отчетов о проблемах заказчиков, которые были закрыты в недельный/месячный период.

Рассмотрим еще примеры.

Пусть цель сформулирована так: «Охарактеризовать заключительный продукт с учетом классов дефектов». Тогда один из возможных вопросов: «Каково распределение ошибок в классе в пределах фазы уточнения дефектов?». В этом случае применяется следующая метрика: ошибки требований — их число и классификация.

Другой пример. Цель: «Проанализировать поставленный ПП с точки зрения четкости представления, с учетом эффективности повторного использования, а также с точки зрения членов команды разработчиков».

Вопрос 1: «Какой процент от общего числа составляют модули, которые не создаются изначально, а, например, используются заново целиком или в виде отдельных подсистем?».

Метрика 1.1 (для каждого программного модуля): уточнение, разработан ли модуль с самого начала (да, нет).

Метрика 1.2 (для каждого программного модуля): название подсистемы, к которой модуль имеет отношение.

Вопрос 2: «Каково процентное соотношение между повторно и полностью используемыми кодами (в завершенной системе или по подсистемам)?».

Метрика 2.1 (для каждого программного модуля): уточнение, применяется ли модуль повторно (да, нет).

Метрика 2.2 (для каждого программного модуля): название подсистемы, к которой модуль принадлежит.

Вопрос 3 (для программных модулей, где код применяется повторно): «Каково распределение модулей при повторном использовании классов модификации (полностью или по подсистемам)?».

Метрика 3.1: степень изменения кода [неизменный; приблизительный (изменено менее 20 % строк); удаленный (изменено более 20 % строк)].

Метрика 3.2: название и версия для исходного модуля, применяемого повторно.

Метрика 3.3. название подсистемы, к которой принадлежит модуль.

Вопрос 4: «Какова взаимосвязь между повторно используемым модулем и достигнутой степенью надежности?».

Метрика 4.1 (для всех модулей): уровень повторного использования.

Метрика 4.2 (для модулей, имеющих наибольшее число сбоев): уровень повторного использования.

Метрика 4.3 (для модулей, не имеющих сбоев): уровень повторного использования.

После сбора данных о модулях с определенной степенью повторного использования можно сделать следующие выводы:

повторное использование большей части модулей, результаты которых уже зафиксированы, приводит к более низкой плотности ошибок, чем изначальный процесс разработки модулей;

наличие ошибок в повторно используемых модулях лучше проверять еще до их применения, поскольку эти модули изучаются и тестируются более интенсивно, чем заново разрабатываемые модули.

Метрики обеспечивают всю количественную информацию, требующуюся для получения удовлетворительных ответов на вопросы.

С помощью лишь одной метрики можно получить ответы на несколько вопросов. Например, метрика «Возраст рассматриваемых проблем» позволяет получать ответы на следующие вопросы: «Достаточно ли собрано ресурсов для разрешения проблемы?»; «Как долго проблемы остаются нерешенными?».

С другой стороны, для ответа на один вопрос могут потребоваться несколько метрик. Например, метрики «Возраст рассматриваемых проблем» и «Возраст разрешенных проблем» следует использовать обе для ответа на вопрос: «Каким образом определить ответную реакцию на запросы заказчиков?».

После определения целей, вопросов и метрик в организации или проектной среде следует определить механизмы для сбора реальных метрических данных.

5.3.5. Этап 4 GQM: разработка механизмов сбора данных

Важно постоянно держать цель в фокусе внимания. Сбор данных, которые не относятся к целевым измерениям, не приносит пользы. В идеале собранные данные размещаются в исторической базе данных организации и постоянно используются в работе. Чтобы правильно организовать работу механизмов сбора, полезно рассмотреть следующие вопросы.

Кто ведет сбор данных? Сбор ведет тот, кто «ближе» находится к данным. Например, сбор данных о трудозатратах персонала при

разработке ПП лучше выполнять самим разработчикам. Будучи наиболее информированными, они не относятся к категории сотрудников, составляющих отчеты о проведенных измерениях, благодаря чему занимают более объективную позицию.

Когда следует выполнять сбор данных? Все зависит от того, какие данные необходимо собирать и как они будут использоваться. Однако установка «как можно раньше и как можно больше» не столь уж плоха. Если желательно вести сбор данных «по настроению», то лучше делать это ежедневно, по крайней мере — раз в неделю (что более реально). Ежемесячный сбор данных практикуется довольно редко, поскольку при этом зачастую сложно вспомнить, какой период времени эти данные характеризуют.

Каким образом организовать сбор данных наиболее точно и эффективно? Лучше всего воспользоваться автоматизированными инструментами. Механизмы по сбору данных должны быть удобными в работе.

Кто же непосредственно работает с метриками? Все зависит от «точки зрения» или от перспективы, описанной в целевом утверждении. Однако не столь важно, кто получает отчеты. Тот, кто вводит данные, должен знать, каким образом они интерпретируются и кому следует с ними ознакомиться. С другой стороны, необходимо гарантированно работать только с точными данными.

Сбор метрических данных должен быть удобным и простым. Если сбор данных и аналитический процесс имеют целевую направленность, автоматизированы и интегрированы в программный процесс, то успех гарантирован. При этом всегда необходимо следить, чтобы процесс измерения не вносил искажения в данные.

5.3.6. Этап 5 GQM: сбор, подтверждение и анализ данных в реальном времени для поддержки обратной связи между корректирующими действиями и проектами

Ведение записей вручную обычно служит причиной искажений, ошибок, пропусков и задержек. Поэтому там, где это возможно, следует обращаться к автоматизированному сбору данных. При этом следует руководствоваться следующими принципами: не усложнять процедуру сбора; избегать ненужных записей; обучить персонал вести записи в соответствии с процедурами, которые для этого используются; подтверждать все данные, собранные в исторической базе данных.

Процесс анализа может принимать разнообразные формы, отличаться применением большого числа инструментов и форм отчетности.

Графическое представление метрических данных практически всегда облегчает представление аналитических результатов. Обычно используются следующие графические представления: конт-

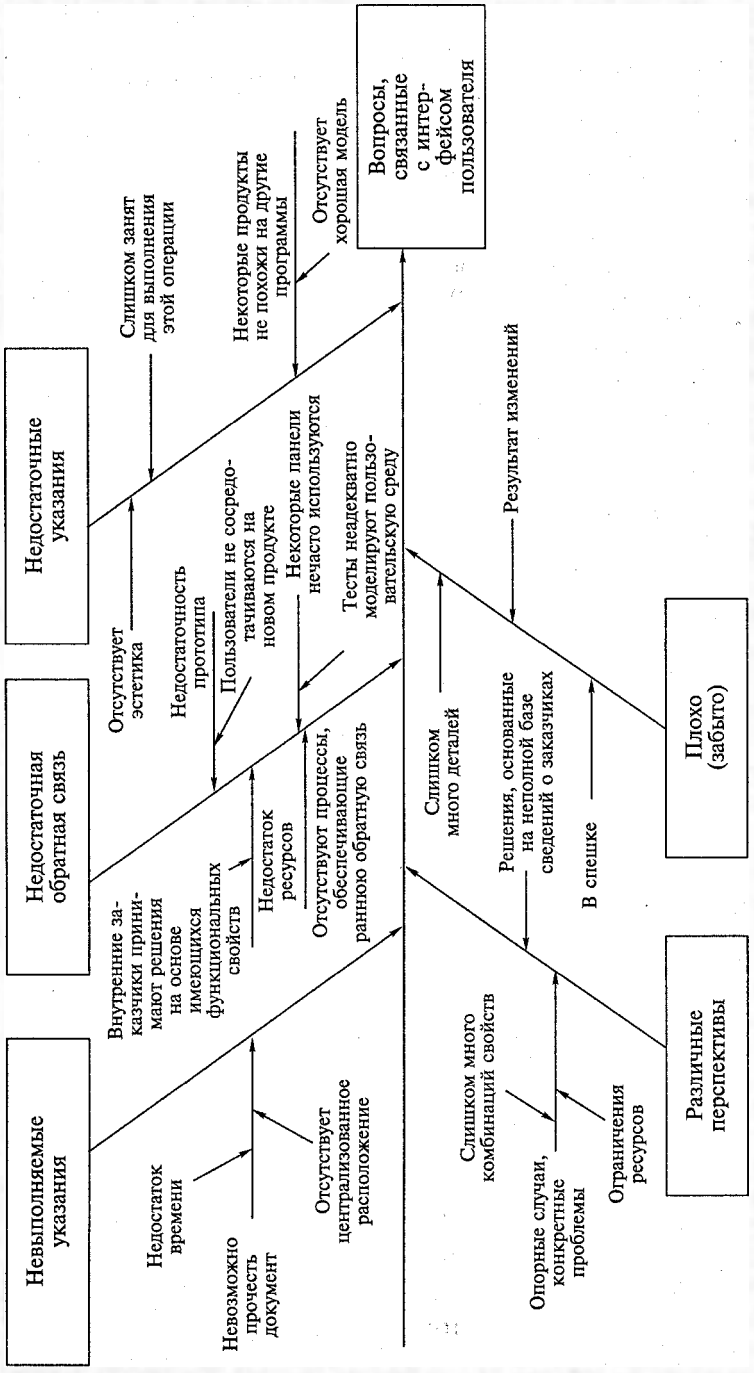


Рис. 5.4. Пример диаграммы Ишикава

рольные графики, гистограммы, диаграммы Ишикава (Ishikawa), диаграммы Парето (Pareto) и рассеянные диаграммы.

Пример диаграммы Ишикава (причина и следствие, в виде елочки) показан на рис. 5.4.

Комбинирование в одном месте (на одной странице) цели, вопроса, метрик и аналитического графика (рис. 5.5) является универсальным описательным методом для представления метрик заинтересованным лицам. Например, компания, занимающаяся разработкой ПП, может использовать следующий набор вопросов и метрик при отслеживании прогресса, направленного на достижение цели: «Усовершенствование ПП».

Вопрос 1: «Сколько новых открытых проблем возникло в течение этого месяца?».

Метрика 1: число новых проблем (NOP — New Open Problems) — равно общему числу открытых проблем в текущей версии ПП в течение месяца после ее выпуска.

Рис. 5.5. Пример расположения на одной странице цели, вопроса, метрик и аналитического графика:

ВОП — «средний возраст» открытых проблем месяца; ВЗП — ВОП — «средний возраст» проблем, закрытых в течение месяца

Вопрос 2: «Какое число актуальных (открытых и незакрытых) проблем осталось на конец месяца?».

Метрика 2: общее число новых проблем (TOP — Total Open Problems,) — равно общему числу новых проблем в версии ПП после ее выпуска, которые остались незакрытыми на конец месяца.

Вопрос 3: «Каков «средний возраст» проблем, актуальных на конец месяца?».

Метрика 3: средний возраст актуальных проблем (AOP — Mean Age of Open Problems) — равен общему времени существования проблем в версии ПП после ее выпуска, которые остались незакрытыми на конец месяца, деленному на число этих проблем.

Вопрос 4: «Каков «средний возраст» проблем, которые были закрыты в течение месяца?».

Метрика 4: средний возраст закрытых проблем (ACP — Mean Age of Closed Problems) — равен общему времени существования проблем в версии ПП после ее выпуска, закрытых в течение месяца, деленному на число актуальных проблем в версии ПП после ее выпуска, решенных в течение месяца.

5.3.7. Этап 6 GQM: анализ данных с использованием подпрограммы для оценки соответствия целям и рекомендации для дальнейшего совершенствования

Необработанные метрические данные обычно весьма трудны для понимания. Например, если получено регистрационное сообщение об ошибке, включающее в себя 150 записей, где отмечены дата, время, код ошибки, эти данные трудно интерпретировать. Однако если в течение определенного периода времени ведется подсчет числа появлений ошибки каждого типа, то эти данные наполняются определенным смыслом. Их можно использовать для принятия решения относительно того, достигнута ли цель. На рис. 5.6 показано, как с помощью гистограммы можно графически представить ошибки каждого типа.

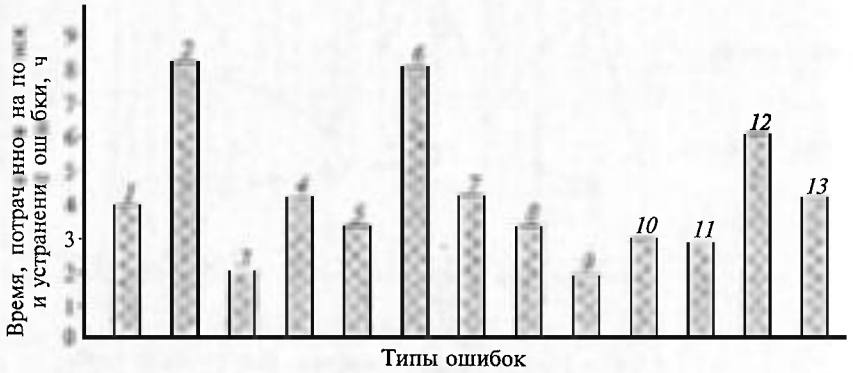


Рис. 5.6. Пример получения данных с помощью гистограммы:

1 — логические ошибки; *2* — вычислительные ошибки; *3* — ошибки, связанные с интерфейсом; *4* — ошибки, связанные с выделением интервалов времени; *5* — ошибки, связанные со взаимодействием между операциями; *6* — несоответствия стандартам; *7* — неоднозначные требования; *8* — неправильные выходные данные или их отсутствие; *9* — неправильные входные данные или их отсутствие; *10* — некорректная инициализация данных; *11* — неправильно вызванные подпрограммы; *12* — ошибки, связанные с обработкой данных; *13* — ошибки в документации

Поддержка обратной связи — это очень важный шаг в процессе выполнения измерений. Все сотрудники проекта и другие пользователи, а особенно те, кто собирал данные, должны иметь возможность просматривать результаты своей работы.

Например, если разработчики и тестировщики располагают информацией о плотности распределения дефектов в проверенных модулях (рис. 5.7), то они могут сделать определенные выводы или, по крайней мере, обратиться к целому ряду вопросов. Почему столь большое число дефектов содержится в первом модуле? Не в том ли причина, что эксперты являются новичками в данном виде работ? А может быть, программист недостаточно хорошо знаком с языком или предметной областью приложения?

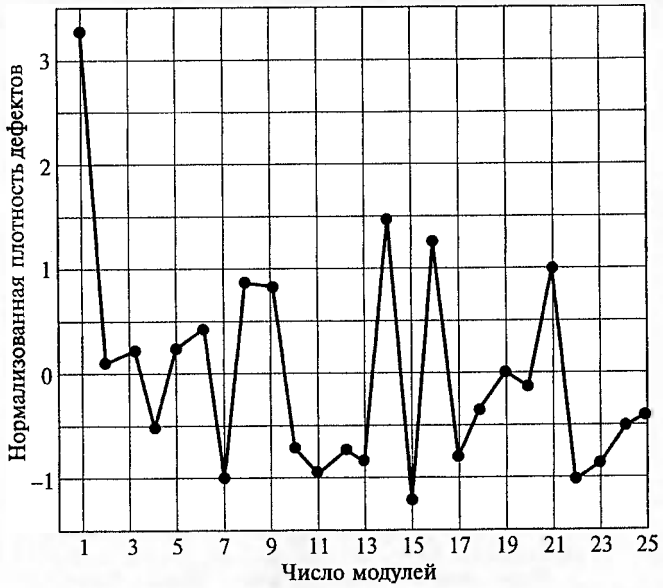


Рис. 5.7. График плотности распределения дефектов в проверенных модулях

Анализ приведенного на рис. 5.7 графика можно использовать для совершенствования процесса. Если разработчики и тестировщики, собирающие данные обзоров и тестовых испытаний, не ознакомятся с полученными результатами, они не только упустят возможность высказать свое мнение при обсуждении путей совершенствования процесса, но и, возможно, не получат мотивацию для работы с точными данными. Универсальным способом представления обратной связи является отображение на одной «странице» цели, вопросов, метрик и графических результатов, как показано на рис. 5.4. Другие примеры графических отображений механизмов обратной связи показаны на рис. 5.8—5.12.

Представления о плотности распределения дефектов весьма полезны, поскольку позволяют выделить с целью дальнейшего анализа модули, имеющие наибольшее число дефектов. Чаще всего процесс анализа начинается с выяснения основной причины. Не потому ли в модуле столь большое число дефектов, что он слишком сложен? Не относятся ли обнаруженные погрешности к ошибкам определенного типа (например, к ошибкам начального этапа проекта), которые присущи данному программисту? Если это так, не следует ли программисту посещать курсы повышения квалификации? Если один из этих модулей планируется повторно использовать в будущем, но плотность распределения дефектов в нем высока, не следует ли пересмотреть этот план?

Если известно среднее время, в течение которого должен быть исправлен основной дефект, и на основе определенных исторических данных есть возможность предсказать число ожидаемых

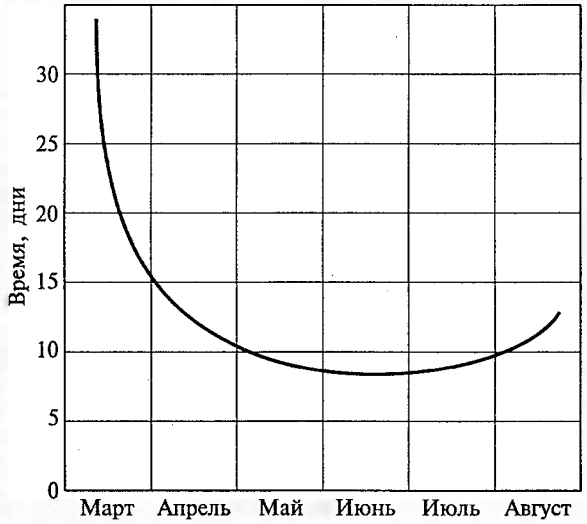


Рис. 5.8. Среднее время, затрачиваемое на коррекцию основного дефекта

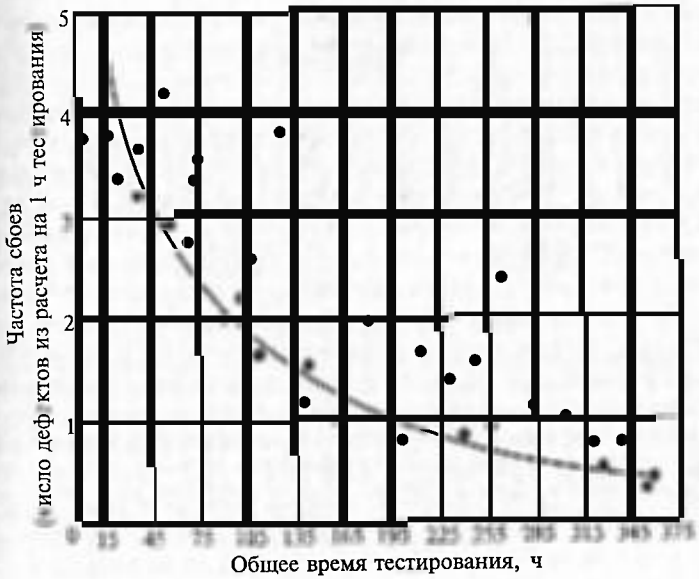


Рис. 5.9. Изменение числа сбоев с течением времени

ошибок, то можно обоснованно делать вывод об объеме средств, необходимых для исправления этих ошибок. Если же трудозатраты, направленные на коррекцию основных дефектов, сначала снижают, а потом снова начинают расти, как показано на рис. 5.8,

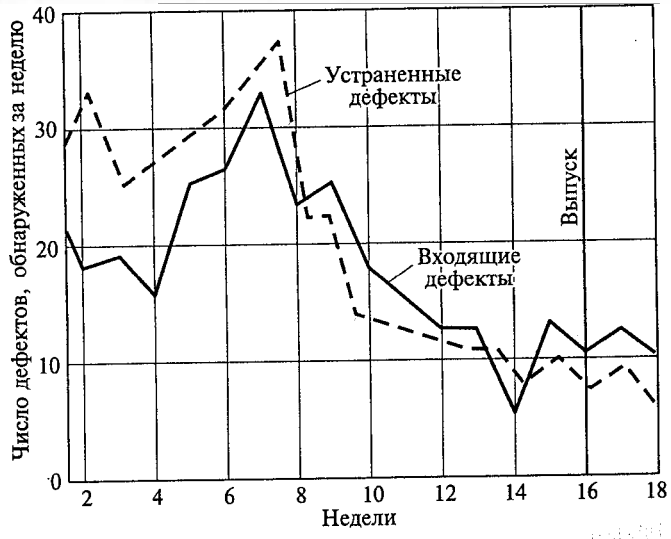


Рис. 5.10. Обнаружение дефектов и их устранение с течением времени

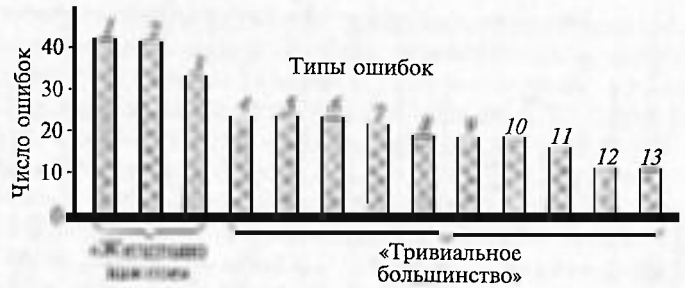


Рис. 5.11. Диаграмма Парето:

1 — вычислительные ошибки; *2* — несоответствия стандартам; *3* — ошибки, связанные с обработкой данных; *4* — ошибки, связанные с выделением интервалов времени; *5* — ошибки в документации; *6* — неоднозначные требования; *7* — логические ошибки; *8* — неправильные выходные данные или их отсутствие; *9* — ошибки, связанные со взаимодействием между операциями; *10* — некорректная инициализация данных; *11* — неправильно вызванные подпрограммы; *12* — неправильные входные данные или их отсутствие; *13* — ошибки, связанные с интерфейсом

значит, что-то делается неверно. Возможно, из-за роста энтропии структуры возросла сложность, и необходимо сделать компонент более простым.

В среде тестирования ошибки обнаруживаются и фиксируются, «на скорую руку» выполняются регрессионные тесты. Всякий раз, когда реализуется регрессионная оболочка, можно ожидать появ-

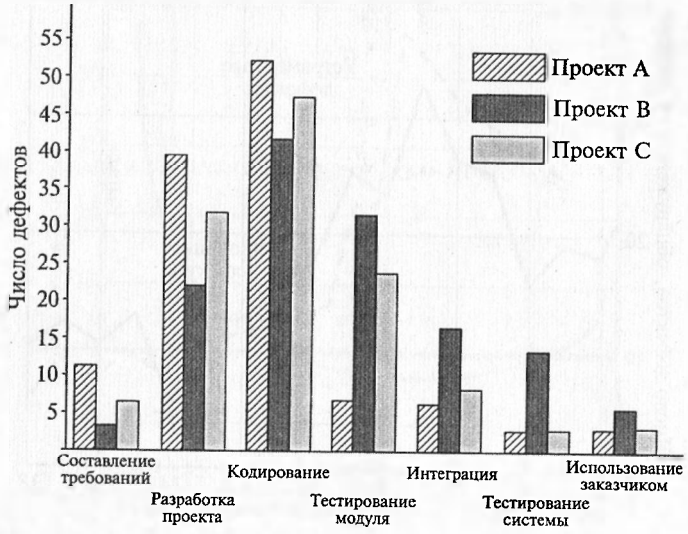


Рис. 5.12. Диаграмма, позволяющая выявить этап сосредоточения дефектов

ния «на поверхности» нескольких ошибок, как показано на рис. 5.9. Если число сбоев растет, а не уменьшается, значит, разработчики заняты только частью проблемы, а не решением ее целиком. Руководитель проекта может приостановить тестирование и проанализировать данную тенденцию. Если же число сбоев сокращается и при этом остаются все менее важные ошибки, то руководство компании, руководитель проекта и заказчик могут прийти к соглашению о выпуске данного ПП. Подобная разновидность анализа тенденций не ограничивается этапом тестирования. Как показано на рис. 5.10, отслеживание дефектов может начаться даже раньше, чем появятся первые статические обзоры проекта.

Диаграмма Парето, пример которой приведен на рис. 5.11, помогает руководителю проекта уточнить источники ошибок. Если справедливо часто встречающееся утверждение о том, что 80 % проблем скрыто в 20 % кода, тогда следует уточнить, о каких именно 20 % идет речь. Упорядоченная диаграмма Парето — прекрасный помощник в принятии подобного решения. После того как «жизненно важное» будет отделено от «тривиального большинства», дальнейший анализ позволит выявить детали.

Если известно, в какой именно фазе жизненного цикла разработки ПП сосредоточены дефекты, то особое внимание уделяют ей. Скорее всего, именно там будет производиться наибольший объем обзоров и проверок, к решению проблемы подключатся представители других проектов компании, обладающие большим опытом и знаниями. Результаты одного проекта могут рассматриваться как аномалия, но если несколько проектов укажут на один и тот же «сомнительный» этап, например на этап кодирования, как показано на рис. 5.12, то руководство компанией получит ясные представления, которые позволят выработать правильное решение.

5.4. Набор основных метрических показателей

5.4.1. Основные источники метрических показателей

При управлении программными проектами отмечают три очень обширных источника данных: трудозатраты; обзоры; запросы на изменение. Эти три базиса взаимодействуют друг с другом, образуя единое основание. Их можно использовать в качестве первого шага при формировании практической системы метрических показателей.

Управление требует точного представления, во-первых, об объемах трудозатрат в рамках проекта, а также о характеристиках этих трудозатрат; во-вторых, — о числе и типах ошибок и дефектов, обнаруженных в процессе обзора; в-третьих, — о запрашиваемых

изменениях, о том, изменяет ли запрос исходные требования системы или полученный компонент проекта изменяется вследствие открывшегося дефекта.

5.4.2. Трудозатраты

О трудозатратах больше всего говорят в связи с ресурсами, затраченными в процессе разработки ПП. Аппаратное обеспечение, телекоммуникации, программные инструменты и другое ценное имущество редко оказывают серьезное влияние на бюджет разработки ПП. При разработке программ, как правило, не требуется дорогостоящее оборудование, отсутствует потребность в станках и складских помещениях. Затраты на создание ПП существенно зависят от человеческого фактора. В этом случае требуется составить представление о том, чем занят персонал (наиболее «дорогостоящий» элемент во всей конструкции), выполняется ли обработка ценной информации, касающейся разработки проекта, каков ход выполнения проекта и каким образом распределяется бюджет на оставшуюся часть работ по проекту. Следует совершенствовать процесс разработки как для данного, так и для будущих проектов, увеличивать степень производительности разработчиков, а также составлять прогноз о стоимости будущих проектов.

Сбор данных о трудозатратах представляет важность не только для оценки следующего подобного проекта; эти сведения помогают руководству в принятии многих других решений.

Для реальной оценки точного объема затрат при создании ПП следует аккуратно отслеживать объем трудозатрат для каждой задачи, выполняющейся в процессе разработки.

Осведомленность о принципах сбора данных, отражающих трудозатраты, часто помогает руководству получить представление о том, какие фазы потребуют наибольших трудозатрат, об объемах времени, затрачиваемого на решение отдельных задач, дополнительного времени, которое необходимо для выполнения работы, а также об объемах непроизводительной дополнительной работы, которую обычно следует выполнять в обязательном порядке.

Система формирования отчетов персонала, включающих в себя сведения о трудозатратах, должна быть автоматизирована, а служащий не должен затрачивать много времени на периодическое заполнение отчета.

5.4.3. Обзоры

В процессе обзора ведут запись некоторой связанной с ним информации: о числе ошибок, типе каждой ошибки, фазе, которая породила данную ошибку, а также о той фазе, на которой эта ошибка проявилась.

Одним из наиболее ценных метрических показателей, применяемых для контроля за выполнением проекта, является время, затраченное на обзор каждого продукта. Если обзоры производятся произвольным образом, а время точно отслеживается, то можно привлекать для проведения экспертиз членов команд из других проектов. Тогда происходят обмен опытом, поскольку эксперты данной команды могут выполнять аналогичную работу на взаимовыгодных началах, взаимное обучение и поддержка. Несомненно, это благотворно сказывается на качестве конечного ПП. Ведение хронометража при данном виде работ позволяет сделать вывод, что чем сложнее процессы обзора ПП, тем меньше времени требуется для их повторной переработки. Подобные обзоры приводят к экономии средств и улучшают качество проекта.

Важно знать, на каком этапе возникли затруднения. Если проблемы проявились на поздних этапах жизненного цикла разработки, это влечет за собой повышенный объем затрат на их устранение, поскольку возрастает объем переделок. Проблемы, проявившиеся после выпуска ПП, обычно обнаруживает уже заказчик. Обзоры позволяют получить информацию о видах затруднений. В этом случае целые категории проблем можно исключить или уменьшить их влияние. В процессе выполнения обзоров идентифицируются проблемы, которые можно скорректировать относительно быстро и с небольшими издержками.

Проблемы удобно подразделять на ошибки и дефекты. *Ошибка* классифицируется как проблема, обнаруженная на этапе ее зарождения. *Дефектом* называется проблема, которая была обнаружена не на этапе ее появления, а лишь на некотором более позднем этапе жизненного цикла разработки.

Устранение ошибок недешево, но устранение дефектов обходится еще дороже. Наиболее дорогостоящим является устранение тех дефектов, которые возникли на ранних этапах жизненного цикла, особенно на этапе формулирования требований, а также тех, которые обнаружились на поздних этапах жизненного цикла, особенно на этапе тестирования системы.

При работе над проектом периодически выполняется инспектирование, проводятся экспертное оценивание, а также структурированные сквозные измерения дефектов продукта, т. е. обзоры. Обнаружение дефектов на ранних этапах разработки путем проведения обзора является тем ценным вкладом, который, несомненно, требует весомого вознаграждения со стороны организации, но при этом надо помнить, что статистика дефектов должна ассоциироваться с обзором, а не с «виновником» появления дефекта.

Информация о каждой ошибке или дефекте записывается, при этом фиксируются: момент выполнения обзора; информация об участниках процедуры; указание, что именно подлежит обзору;

согласованное мнение об ошибках и дефектах, типах ошибок и дефектов, источнике их возникновения, уровнях их серьезности. Кроме того, выполняется запись о доступности ПП или о потребности проведения повторного обзора.

Результаты обзоров помогают ответить на целый ряд важных вопросов:

Кто проводит качественное оценивание и кто какую роль в нем играет?

Какие этапы отличаются наибольшим числом проблем?

На каких этапах обнаружены наиболее серьезные проблемы?

Какие встречаются типы ошибок и дефектов?

Какой тип ошибок обнаруживается чаще всего?

5.4.4. Запросы на изменение

Этот важный источник данных для измерения формируется при изменении программной системы. После поставки ПП разработчики с помощью системы отслеживания обычно ведут запись недочетов или запросов на изменение, прилагая усилия к их устранению. Согласно материалам Национального института стандартов и технологий (NIST — National Institute of Standards and Technology) выделяют следующие типы запросов на изменение:

корректирующий — направлен на исправление сбоя (ошибки или дефекта);

адаптационный — направлен на адаптацию ПП к изменениям в системе, приспосабливание к изменившейся среде (внешним обстоятельствам) без внесения улучшений или качественных изменений;

превентивный — направлен на предупреждение сбоев до их проявления;

усовершенствованный — направлен на поддержку с целью облегчения дальнейшего сопровождения.

Проблемы могут возникать не только на этапе сопровождения после поставки ПП, но и на этапе его разработки. Они могут обнаруживаться при осуществлении обзоров (статических тестов) и выполняемых тестов (динамических тестов). Существуют и другие методы обнаружения проблем, например изучение ПП самим разработчиком или ознакомление с проблемами самого пользователя.

Необходимость внесения изменения обычно диктуется пользователем системы, но инициатива может исходить и от наблюдателя.

Обычно информация, обосновывающая необходимость внесения изменений, включает в себя следующие пункты:

указание того, кто сообщает о необходимости изменения;

описание проблемы или потребности (стандарт IEEE по аномалиям программных продуктов — IEEE Standard on Software Anomalies — содержит полный перечень категорий возможных проблем);

указание, где и когда проявилась проблема, или же сообщение об ошибке;

название системы или подсистемы (если известно);

указание о критичности данного запроса (основанной на предварительно определенных уровнях серьезности);

указание о корректности данного изменения, его месте в процессе совершенствования продукта, приспособляемости, превентивности или связанном с ним повышении качественных показателей;

сообщение о влиянии данного изменения.

Отклик на запрос об изменении сопровождается следующей информацией:

какое место в системе будет изменено или где находится проблема;

этап разработки, на котором появилась проблема;

необходимый набор тестов;

оцененное количество трудозатрат (число часов), необходимых для выполнения изменений;

когда именно (с указанием календарного времени) ожидается выполнение изменений.

После реализации изменений заключительный набор сведений включает в себя информацию об обнаружении и устранении проблемы или о произведенном усовершенствовании. При этом указываются:

дата реализации изменения;

реальный объем затраченных усилий;

где (в пределах системы) произведено изменение;

результаты тестирования.

Обычно в процессе изменения ПП преследуются следующие цели:

сокращение времени цикла для отклика на запросы по изменению ПП;

уменьшение числа изменений, которые необходимы для повышения качества программы.

С целью определения степени прогресса в достижении поставленных целей часто задают следующие вопросы.

Какова суть запрашиваемых изменений?

Изменения служат для устранения ошибок или для внесения усовершенствований?

Каковы составляющие компоненты проблемы?

Как часто изменяются требования?

Как долго проблема оставалась неустранимой?

Какие серьезные проблемы содержатся в отчетах?

На каком этапе появляется необходимость изменения?

При внесении изменений возникает потребность в планировании работы по их реализации. Планирование подразумевает уточнение необходимых изменений, выбор модулей ПП для изменения, версии ПП с учетом контроля конфигурации, определение того, какие атрибуты следует измерять после выполнения работы. Планирование также предполагает определение процедур для обработки собранных данных, анализ данных и отчет о результатах измерений, ведение документации всех процедур, а в случае необходимости и обучение всех желающих пользователей из круга заинтересованных лиц.

Контрольные вопросы

1. Какова цель измерений: а) процесса; б) программного продукта; в) проекта?
2. Дайте определение понятия «метрика».
3. Какие группы и типы метрик вы знаете?
4. Приведите примеры метрик, относящихся: а) к программному продукту; б) к процессу; в) к проекту.
5. Какие цели преследует измерение атрибутов: а) программного продукта; б) проекта; в) процесса?
6. Какие основные задачи позволяют решать разработчикам полученные метрики?
7. Где хранятся собранные метрики?
8. Какие основные действия по сбору и анализу метрик соответствуют каждому из пяти уровней модели CMM-SEI?
9. Какие измерения выполняются: а) при управлении программными требованиями; б) при планировании проекта; в) при отслеживании и контроле хода выполнения проекта; г) при реализации программы обучения; д) при программном инженеринге; е) при количественном управлении процессом?
10. В чем суть парадигмы Бейзили?
11. Какие основные этапы включает в себя методика, лежащая в основе парадигмы Бейзили?
12. Дайте краткую характеристику каждому этапу методики Бейзили.
13. Каким образом выполняются: а) определение набора целей; б) определение набора вопросов, характеризующих цели; в) определение метрических показателей, необходимых для получения ответов на поставленные вопросы; г) разработка механизма сбора данных; д) сбор, подтверждение и анализ данных, необходимых для обратной связи между корректирующими действиями и проектами; е) анализ данных для оценки соответствия целям; ж) поддержка обратной связи для руководства проектом?
14. Перечислите основные метрические показатели.
15. Дайте определения понятий «ошибка» и «дефект» и объясните разницу между ними.
16. Укажите основные типы запросов на изменение.

ГЛАВА 6

ПЛАНИРОВАНИЕ РАБОТ ПО СОЗДАНИЮ ПРОГРАММНЫХ ПРОДУКТОВ

6.1. Структура разделения работ по созданию программного продукта

Планирование работ начинается с получения первичных требований заказчика (ПТЗ), а основой планирования является выделение всех необходимых для выполнения и успешного завершения проекта задач и определение связей между ними. Результатом этого является *структура разделения работ* по созданию ПП.

Оцениваются объем и трудоемкость каждой выделенной задачи и каждого элемента структуры, определяются необходимые ресурсы и временной график реализации жизненного цикла. Процесс планирования определяется как циклический; его цикл показан на рис. 6.1.

График разработки ПП оценивается с точки зрения реальности выполнения, и в случае получения по каким-либо показателям нереального графика цикл планирования повторяется. При этом не всегда обязательно повторять выполнение всех выделенных задач этапа планирования.

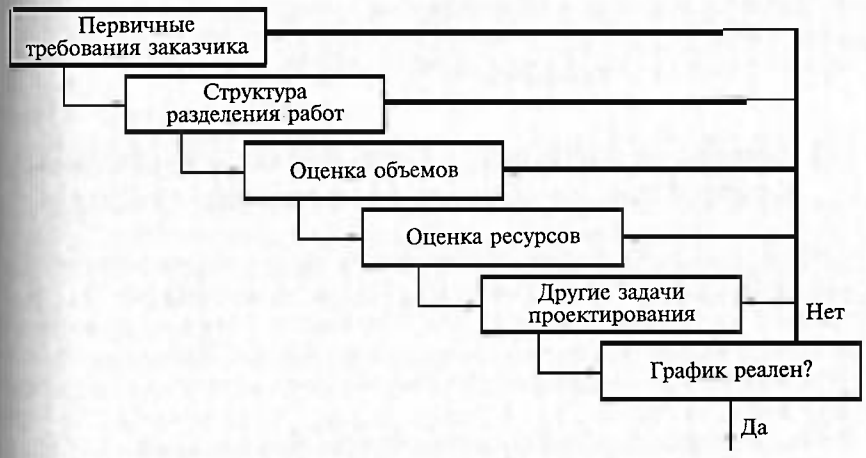


Рис. 6.1. Цикл планирования работ по созданию программного продукта

Как правило, структура разделения работ представляет собой иерархию задач.

Детализацию в иерархии задач необходимо производить до уровня, достаточного для проведения оценки сложности и объема каждой задачи. Задачи низшего уровня структуры разделения работ должны быть настолько малы и просты, чтобы любую из них мог выполнить отдельный исполнитель за достаточно короткий отрезок времени.

Структурирование желательно заканчивать построением структурной диаграммы, отражающей общую концепцию дальнейшего проектирования ПП.

6.2. Оценка объемов и сложности программного продукта

За единицу объема ПП принято число строк программного кода (LOC), а за единицу производительности — число строк эффективного программного кода (т. е. число строк программного кода в отлаженном ПП), производимых одним человеком за один месяц (LOC/чел.-мес).

Отдельные работы, не связанные с конструированием программного кода, следует измерять в человекочасах.

Объем и сложность каждого элемента структуры разделения работ определяются при помощи экспертной оценки и выражаются числом LOC и человекочасов. Рекомендуется использовать для получения каждой оценки не менее трех независимых экспертов, усредняя их показания. При этом сложность структурного элемента учитывается весовым коэффициентом сложности $K_c = 0{,}75 \ldots 1{,}25$. Для получения объема структурного элемента необходимо его экспертную оценку умножить на коэффициент сложности K_c.

6.3. Оценка технических, нетехнических и финансовых ресурсов для выполнения программного проекта

По объемам отдельных структурных элементов вычисляется общий объем работ по созданию ПП (LOC и человекочасы). В зависимости от объема кода ПП подразделяют на малые, промежуточные, средние и большие. Используя табл. 6.1, определяют необходимое число исполнителей для создания программного кода (программистов).

Для получения общего числа исполнителей проекта к числу программистов добавляют число человек, определенное по человекочасовым затратам.

Таблица 6.1

Показатели программных продуктов

Объем программного продукта	Трудоемкость, чел.-мес	Производительность, LOC/чел.-мес	Срок создания, мес	Необходимый штат, чел.
Малый (2 KLOC)	5,0	400	4,6	1,1
Промежуточный (8 KLOC)	21,3	376	8,0	2,7
Средний (32 KLOC)	91,0	352	14,0	6,5
Большой (128 KLOC)	392,0	327	24,0	16,0

По каждому выделенному структурному элементу разделения работ определяют квалификацию исполнителей, требуемые инструментальные средства (аппаратные и программные) для его выполнения, возможные дополнительные финансовые затраты. Далее при необходимости определяют последовательность использования во времени отдельных ресурсов, механизмы их разделения различными структурными элементами, ограничения по срокам разработки.

6.4. Оценка возможных рисков при выполнении программного проекта

Риски, возникающие в процессе разработки, подразделяются на связанные с ресурсным, финансовым и организационным (административным) обеспечением и связанные с большим объемом и сложностью ПП.

Ресурсные риски выявляются при анализе полученных оценок ресурсов и планировании их использования. Такие риски могут быть обусловлены нехваткой персонала нужной квалификации, недостаточной производительностью аппаратного обеспечения, несоответствием программных инструментальных средств или нехваткой финансирования на некоторые дополнительные нужды.

Финансовые риски тесно связаны с ресурсными рисками и рисками, обусловленными большим объемом и сложностью ПП, так как неправильное планирование ресурсов может вызвать превышение бюджета проекта. Кроме того, на финансовые риски влияют изменение конъюнктуры рынка ПП и состоятельность заказчика.

Организационные, или административные, риски связаны с неправильной организацией хода разработки, ошибками в планировании и распределении обязанностей, недостаточной ответственностью исполнителей.

Риски последней разновидности обусловлены неточностью предварительных оценок объемов и сложности ПП. Заниженные предварительные оценки могут привести к неправильному определению необходимых объемов ресурсов и, в конечном итоге, к срыву сроков выполнения работ.

После выявления возможных рисков производится экспертная оценка вероятности их возникновения и планируются способы их преодоления.

6.5. Составление временного графика выполнения программного проекта

Для составления временного графика выполнения проекта необходимо проанализировать и обобщить полученные ранее оценки объемов и ресурсов, запланированные объемы работ, инструментальные ресурсы и распределить персонал по фазам жизненного цикла. Такое распределение производится на основе имеющегося исторического опыта подобного планирования. Если опыт отсутствует, то можно воспользоваться табл. 6.2.

Составление временного графика проекта начинают с построения GANTT-диаграммы этапов разработки, пример которой приведен на рис. 6.2. При планировании небольших проектов это легко делать вручную. На диаграмме, получившей такое название по имени автора — Генри Ганта (Henry Gantt), хорошо видны очередность и взаимосвязь этапов, их последовательность во времени, конечный срок завершения проекта.

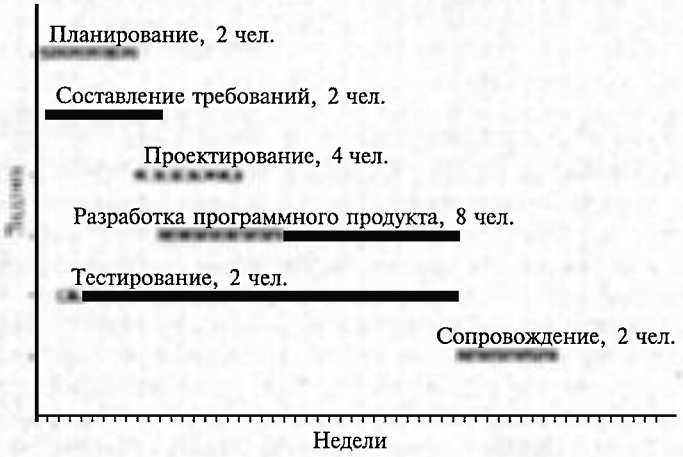


Рис. 6.2. Пример GANTT-диаграммы

Таблица 6.2

Распределение трудозатрат и временных затрат по основным этапам разработки программного продукта

Этап	Объем проекта, %			
	малого (2 KLOC)	промежуточного (8 KLOC)	среднего (32 KLOC)	большого (128 KLOC)
Трудозатраты				
Планирование, составление требований, высокоуровневое проектирование	16	16	16	16
Детальное проектирование	26	25	24	23
Разработка	42	40	38	36
Тестирование, сопровождение	16	19	22	25
Временные затраты				
Планирование, составление требований, высокоуровневое проектирование	19	19	19	19
Детальное проектирование, разработка	63	59	55	51
Тестирование, сопровождение	18	22	26	30

Для каждого этапа указывают, сколько человек занято его выполнением, какова его продолжительность, даты начала и завершения этапа.

Для больших проектов ручное построение диаграмм затруднено. В этом случае рекомендуется использовать автоматизированные средства, например Microsoft ProjectTM, что позволит глубже представить структуру распределения этапов работ и ресурсов, учесть взаимосвязь между отдельными работами, рационально распределить персонал, избежать перегрузок и простоев.

Для больших проектов рекомендуется представлять также обобщенный график выполнения проекта (формируется вручную), в котором отмечены основные фазы подобно рис. 6.2.

Допускается отсутствие в плане проекта подробной GANTT-диаграммы при условии, что полная структура разделения работ представляется таблицей.

6.6. Собираемые метрики, используемые методы, стандарты и шаблоны

На этапе планирования необходимо выполнять оценки затраченных на составление плана проекта ресурсов (время, требующееся для получения утвержденной версии плана), число человек, участвующих в планировании, время создания компьютерной версии плана и его бумажной копии, длительность процедуры согласования и утверждения плана и т.п.). По этим оценкам следует определить производительность отдельных этапов и проекта в целом.

Все полученные данные необходимо хранить в исторической базе данных (ИБД) проектной группы. Кроме того, в ИБД следует заносить запланированные объемы работ и распределение ресурсов по этапам жизненного цикла ПП, длительность этих этапов и все другие данные, которые, по мнению руководителя проекта, могут помочь в дальнейшем улучшить процесс планирования и повысить его производительность.

Используемые инструменты: система подготовки документов (например, MS Word); электронные таблицы (например, Excel); система автоматизации планирования (например, Project).

Используемые методы и стандарты: процесс организации; метрическая программа организации.

Используемые шаблоны: плана проекта; отчета по обзору; отчета о статусе проекта.

Контрольные вопросы

1. Каково назначение этапа планирования в жизненном цикле разработки программного продукта?
2. Что представляет собой цикл планирования?
3. Объясните цель и назначение структурирования работ.
4. В чем и как измеряются объем и сложность программного продукта?
5. Как выполняется оценка необходимых ресурсов для выполнения работ?
6. Какие виды рисков вы знаете и как они оцениваются?
7. Объясните назначение временного графика выполнения работ.
8. Какие метрики собирают на этапе планирования?
9. Какие инструменты, методы, стандарты и шаблоны используют при выполнении этапа планирования?

ГЛАВА 7

УПРАВЛЕНИЕ ТРЕБОВАНИЯМИ К ПРОГРАММНОМУ ПРОДУКТУ

7.1. Общие сведения об управлении требованиями

Одно из первых действий при проектировании ПП — сбор и упорядочение требований к нему. Изначально собираемые требования представляют собой первичные требования заказчика (ПТЗ), протоколы совещаний и интервью с заказчиками и пользователями, копии и оригиналы различных документов, отчеты о существующих аналогичных ПП и массу других материалов. После сбора их начинают упорядочивать и очищать от противоречий. Затем на их основе вырабатывают требования к компонентам ПП — базам данных, программным и техническим средствам. При этом приходится иметь дело с большим количеством неструктурированных, часто противоречивых требований и пожеланий, разбросанных по всевозможным соглашениям о намерениях, приложениям к договорам, протоколам рабочих совещаний, черновым материалам обследований.

Таким образом, без организованных усилий по регистрации и контролю выполнения этих требований велик риск их не учесть. Решение проблемы достаточно очевидно: следует вести учет собираемых требований и контролировать их обработку, оценку и реализацию (или отказ от реализации). Такая работа называется *работой по управлению требованиями*.

Управление требованиями (requirements management) представляет собой:

систематический подход к выявлению, организации и документированию требований к ПП;

процесс, устанавливающий соглашение между заказчиками и разработчиками относительно изменения требований к ПП.

Управление требованиями преследует следующие цели:

достижение соглашения с заказчиком и пользователями о том, что ПП должен делать;

улучшение понимания требований к ПП со стороны разработчиков;

установление границ ПП, т.е. определение технических требований к аппаратуре компьютера, операционной среде и возможностям ПП;

определение базиса для планирования.

Важность управления требованиями объясняется еще и тем, что почти во всех проектах, выполняющихся в экстремальных условиях, приходится устанавливать приоритеты, разделяя все требования на три категории: «необходимо выполнить», «следует выполнить» и «можно выполнить». При такой расстановке приоритетов в проекте очевидная стратегия будет заключаться в следующем: в первую очередь сконцентрироваться на требованиях, которые необходимо выполнить; затем, если останется время, сосредоточиться на требованиях, которые следует выполнить; и, наконец, если будет возможность, заняться требованиями, которые можно выполнить.

Если не следовать такой стратегии с самого начала проекта, то в конце можно оказаться в крайне неприятной кризисной ситуации.

Управление требованиями относится к работам, техническая поддержка которых не требует больших финансовых затрат, но эти работы способны ощутимо повысить качество создаваемого ПП.

Требование — это условие или характеристика, которой должен удовлетворять ПП. Существуют функциональные и нефункциональные требования.

Функциональные требования определяют действия, которые должен выполнять ПП, без учета ограничений, связанных с его реализацией. Другими словами, они определяют поведение ПП в процессе обработки информации.

Нефункциональные требования не определяют поведение ПП, но описывают его атрибуты или атрибуты системного окружения. Можно выделить следующие типы нефункциональных требований:

требования к применению — определяют качество пользовательского интерфейса, документации и учебных курсов;

требования к производительности — накладывают ограничения на функциональные требования, задавая необходимые эффективность использования ресурсов, пропускную способность и время реакции;

требования к реализации — предписывают использование определенных стандартов, языков программирования, операционной среды и т.д.;

требования к надежности — обусловливают допустимые частоту и воздействие сбоев на работу ПП, а также возможности восстановления ПП после сбоев;

требования к интерфейсу — определяют внешние сущности (т.е. пользователей и любые внешние устройства), с которыми может взаимодействовать система, и регламент этого взаимодействия.

7.2. Цикл формирования требований

Этап управления требованиями можно назвать этапом неформального определения архитектуры будущего ПП. Именно этот этап является наиболее сложным, так как в процессе работы над ПП изменяются представления о самом ПП и у заказчика, и у исполнителя. Поэтому на раннем этапе проекта часто трудно точно определить, какими характеристиками ПП должен обладать.

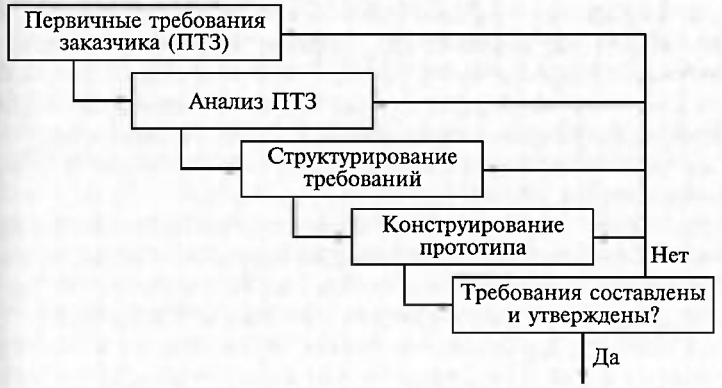


Рис. 7.1. Цикл составления требований к программному продукту

Составление требований, как и планирование, — циклический процесс (рис. 7.1), в каждом цикле (или подцикле) основным действием является выявление и устранение семантических ошибок в описании ПП. При этом процесс составления требований не имеет точно определенного окончания. Для того чтобы он закончился, на одном из шагов, руководствуясь принципом разумной достаточности, принимают решение, что составленные требования больше не имеют семантических ошибок и однозначно описывают то, что ПП должен выполнять.

7.3. Анализ и структурирование первичных требований заказчика

Обычно ПТЗ представляют собой не очень подробное описание ПП. Как правило, оно содержит мало подробностей, а описания отдельных функций не детализированы или не упоминаются вовсе.

На шаге анализа ПТЗ обязательно в сотрудничестве с заказчиком (личные встречи, телефонные разговоры, электронная почта и т.п.) необходимо выяснить, возможно ли создание предлагае-

мого ПП вообще и в указанные сроки в частности, каких деталей в описании не хватает, что еще следует прояснить. Кроме того, необходимо определить сроки и способы получения недостающей информации, способы тестирования и условия приемки готового ПП.

Работу по структурированию требований начинают с составления полного описания будущего ПП без детализации отдельных функций (часть этой работы выполняется на шаге анализа ПТЗ), т.е. с построения обобщенной структуры требований к ПП. При этом указывают лишь общее назначение ПП, особенности аппаратных средств, на которых ПП должен быть реализован, общие особенности и ограничения комплекса, в составе которого ПП должен работать, выделяют основные компоненты ПП. При их выделении необходимо стремиться к тому, чтобы между ними было как можно меньше связей, т.е. к обеспечению возможной информационной независимости.

Обобщенная структура (обобщенное описание) требований к ПП должна давать представление о глобальных связях, основных свойствах и взаимосвязях отдельных структурных элементов ПП.

Далее выполняют детализацию обобщенной структуры требований: уточняют описания взаимосвязей основных компонентов и составляют детальное описание каждого структурного элемента (строят детальную структуру требований к ПП). Для каждого структурного элемента осуществляют детализацию его функций и составляют общее описание каждой функции всех структурных элементов. После этого уточняют и описывают взаимосвязи функций внутри каждого структурного элемента и взаимосвязи функций разных структурных элементов.

При этом анализируют все требования с точки зрения возможности их выполнения в указанные сроки или отложенного выполнения в последующих версиях ПП с использованием приоритетов заказчика по значимости и приоритетов исполнителя по сложности реализации.

Детальная структура требований к ПП должна давать подробное представление о взаимосвязях структурных элементов и общее представление обо всех их функциях.

Последним действием по структурированию требований к ПП является составление описаний отдельных функций каждого структурного элемента.

Для каждой из функций должны быть указаны входные данные и место их расположения, детально описаны производимые над входными данными действия, определены выходной продукт функции и место его расположения, а также способы тестирования.

Глубину детализации каждого требования определяет руководитель проекта. Она должна быть такой, чтобы описания отдельных функций содержали все необходимые подробности и давали разработчику и заказчику ясное представление о том, что же он представляет ПП и что он может делать.

Описание рекомендуется составлять с использованием принятой в технической литературе терминологии.

7.4. Конструирование прототипа

Конструирование прототипа выполняют аналогично проектированию ПП. Изучение свойств прототипа осуществляется с целью уточнения или проверки требований заказчика до окончательного их утверждения.

Заказчикам программного обеспечения и конечным пользователям обычно сложно четко сформулировать требования к разрабатываемому ПП. Трудно предвидеть, как ПП будет взаимодействовать с другими программными пакетами и какие операции, выполняемые пользователями, необходимо автоматизировать. Тщательный анализ требований помогает яснее понять, что ПП должен делать. Однако реально проверить требования, прежде чем их утвердить, практически невозможно. В этой ситуации может помочь прототип системы.

Прототип является начальной версией ПП, которая используется для демонстрации концепций, заложенных в системе, проверки вариантов требований, а также поиска проблем, которые могут возникнуть как в ходе разработки, так и при эксплуатации ПП, и возможных вариантов их решения. Очень важна быстрая разработка прототипа ПП, чтобы пользователи могли начать экспериментировать с ним как можно раньше.

При работе над составлением требований к ПП прототип обеспечивает ряд преимуществ. Пользователи могут экспериментировать с прототипом, что позволяет им проверить, как будет работать ПП. Они могут определить сильные и слабые стороны ПП, в результате чего сформировать новые требования.

Прототип позволяет обнаружить ошибки и упущения в ранее принятых требованиях. Например, некоторые функции ПП, определенные в требованиях, пользователи первоначально могут считать полезными и нужными, однако в процессе применения этих функций совместно с другими функциями вполне способны изменить мнение о них. В результате требования к ПП изменятся, отражая измененное понимание пользователями функций ПП.

Прототипирование можно использовать при анализе рисков на этапе планирования. Основной опасностью при разработке ПП являются ошибки и упущения в требованиях. Затраты на устранение ошибок в требованиях на более поздних стадиях процесса разработки могут быть очень высокими. Прототипирование также

уменьшает общую стоимость разработки системы. По этим причинам оно часто используется в процессе разработки требований.

7.5. Составление спецификаций по требованиям заказчика

Этап управления требованиями заканчивается составлением спецификаций требований, которые могут быть оформлены в виде таблицы. При составлении спецификаций не следует употреблять слова и словосочетания, допускающие неоднозначное толкование.

Утверждение требований исполнителем и заказчиком определяет момент достижения соглашения между ними по всем пунктам спецификаций требований. Заказчик может потребовать предоставить ему прототип или иной пример, иллюстрирующий проверку некоторых требований к ПП, может определить множество проверочных примеров или многократно изменить параметры отдельных требований к ПП. В этом случае такие пожелания заказчика необходимо оформлять в приложении к спецификациям требований.

7.6. Собираемые метрики, используемые методы, стандарты и шаблоны

На этапе составления спецификаций требований заказчика необходимо выполнять оценки затраченных на составление спецификаций требований ресурсов (время, необходимое для получения утвержденной версии спецификаций требований, число человек, участвующих в составлении требований, время создания компьютерной версии спецификаций требований и ее бумажной копии, длительность процедуры согласования и утверждения спецификаций требований и т. п.). По этим оценкам следует определить производительность составления спецификаций требований.

Все полученные данные необходимо хранить в ИБД проектной группы. Кроме того, в нее следует заносить все другие данные, которые, по мнению руководителя проекта, могут помочь улучшить процесс составления спецификаций требований и повысить его производительность.

Используемый инструмент: система подготовки документов (например, MS Word).

Используемые методы и стандарты: процесс организации; метрическая программа организации.

Используемые шаблоны: спецификации требований; отчета по обзору; отчета о статусе проекта.

Контрольные вопросы

1. Каково назначение этапа управления требованиями к программному продукту в жизненном цикле разработки программного продукта?
2. Какие действия включает в себя работа по управлению требованиями?
3. Какие цели преследует работа по управлению требованиями?
4. На какие категории можно разделить все требования при определении их приоритетов?
5. В какой очередности следует реализовывать требования к программному продукту?
6. Какие требования относятся к функциональным и нефункциональным?
7. Какие действия включает в себя цикл формирования требований к программному продукту?
8. Какие цели преследует анализ первичных требований заказчика?
9. Какие действия выполняются при структурировании требований заказчика?
10. Какова роль прототипа при формировании требований к программному продукту?
11. Какие преимущества обеспечивает прототип?
12. Почему требования в спецификации требований должны быть однозначными?
13. Какие метрики собирают на этапе управления требованиями?
14. Какие методы, стандарты и шаблоны используют на этапе управления требованиями?

ГЛАВА 8

ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА

8.1. Общая характеристика и компоненты проектирования

Этап проектирования предназначен для выработки и детализации модели разрабатываемой программной системы. Такая модель определяет структуру программной системы, организацию модулей, интерфейсов и данных, описание которых необходимо для последующего этапа реализации. Указанный этап может быть представлен совокупностью компонент проектирования, для каждой из которых определены набор свойств и связи с другими компонентами.

Компонентой проектирования является элемент проектирования, полученный в результате декомпозиции требований заказчика к ПП. Компоненты проектирования могут быть системами, подсистемами, модулями, элементами данных, процессами и т. п. Все они обладают общими характеристиками, называемыми атрибутами компонент. Для компонент проектирования могут быть определены следующие атрибуты:

название компоненты — ее уникальное имя;

тип компоненты — ее сущность (подсистема, процедура, процесс, элемент данных и т. д.);

функция — выполняемые компонентой действия;

зависимости — описание взаимосвязей с другими компонентами;

интерфейсы — описание методов взаимодействия с другими компонентами;

ресурсы — описание необходимой аппаратной, библиотечной или другой поддержки;

обработка — описание алгоритма выполнения функции;

данные — описание внутренних структур данных.

Перечень необходимых компонент, их названия и назначения определяются в зависимости от выбранного способа построения модели ПП и, в первую очередь, от выбранной методики программирования (например, структурное программирование или объектно-ориентированное). Каждая из этих методик имеет свои способы построения модели ПП.

Результаты проектирования представляются в виде описания компонент проектирования по определенному набору атрибутов.

8.2. Эволюция разработки программного продукта

С момента появления первых электронно-вычислительных машин разработка программного обеспечения прошла большой путь. Восхищения возможностью написания хоть какой-нибудь программы сменилось осознанием того, что именно технология разработки программного обеспечения определяет прогресс в вычислительной технике.

На первых этапах развитие вычислительной техники было ориентировано на решение технических проблем. Предметом забот была прежде всего аппаратура, вычислительная машина как таковая. Казалось вполне естественным, что программы для таких машин разрабатывают в двоичных кодах. Программирование было уделом энтузиастов.

По мере того как вычислительные машины становились все мощнее и надежнее, значение программного обеспечения осознавалось все большим числом ученых и практиков. Важнейший прорыв произошел в конце 1950-х годов, когда появились языки программирования высокого уровня — Fortran, Algol и др. Их появление было обусловлено прежде всего тем, что написание программ без таких языков становилось все более сложной задачей. Решив текущие проблемы, эти языки создали новые. Ускорив и упростив программирование, они существенно расширили круг задач, решаемых с помощью ЭВМ. Появление новых задач, более сложных, чем решавшиеся раньше, привело к тому, что имеющихся средств снова стало недостаточно для успешного их решения.

Такое развитие характерно для всей истории применения компьютеров, вплоть до настоящего времени. Вычислительная техника все время используется на пределе своих возможностей. Каждое новое достижение в аппаратном либо программном обеспечении приводит к попыткам расширить сферу применения ЭВМ и тем самым ставит новые задачи, для решения которых нужны новые возможности.

В сложившейся ситуации усилия программистов были сосредоточены прежде всего на создании новых языков программирования. Появился язык Cobol, дающий возможность решать задачи обработки экономической информации. Он создавался таким образом, чтобы программа на нем выглядела почти как текст на английском языке. Появился язык PL/I, объединивший в себе все возможности, которые только могут иметь языки программирования.

Языки Fortran, Algol, PL/I поддерживают процедурный стиль программирования. Программа разрабатывается в терминах тех действий, которые она выполняет. Основной единицей программы является процедура. Процедуры вызывают другие процедуры,

все вместе они работают по определенному алгоритму, который ведет к решению задачи. Алгоритм — это точная последовательность действий для получения необходимого результата.

Появление перечисленных языков стало следствием круга задач, которые решались с помощью вычислительной техники. Прежде всего это были вычислительные задачи. Основными пользователями ЭВМ являлись ученые и инженеры, которых привлекала возможность быстро получить необходимый результат.

Применение ЭВМ для решения задач искусственного интеллекта и обработки текстов привело к созданию функциональных языков, в частности языка Lisp. Эти языки имеют также хорошо проработанное математическое основание — лямбда-исчисление. В отличие от языков типа Algol, в которых действия в основном выражаются в виде итерации (повторения какого-либо фрагмента программы несколько раз), в языке Lisp вычисления производятся с помощью рекурсии — вызова функцией самой себя, а основной структурой данных является список.

В 1960-е годы многие теоретики и практики осознали, что одно лишь создание новых, более совершенных языков программирования не может решить все проблемы разработки программ. Начались интенсивные исследования в области тестирования программ, организации процесса разработки программного обеспечения и др.

Усилия преобразовать программирование из эмпирического процесса в более упорядоченный, придать ему более «инженерный» характер привели к созданию методик программирования. Это был очень существенный шаг. Разработка методов построения программ, с одной стороны, создавала основу для массового промышленного программирования, а с другой стороны, обобщая и анализируя текущее состояние программного обеспечения, давала мощный импульс созданию новых языков программирования, операционных систем, сред программирования.

Одной из наиболее широко применяемых методик программирования стал структурный подход к программированию. Создателем его считается Э. Дейкстра (E. Dijkstra). Фактически структурный подход к программированию — это первая законченная методика программирования. Законченной ее можно назвать потому, что структурное программирование предлагает путь от задачи до ее воплощения в программе. Структурное программирование оказало огромное влияние на развитие программирования. Этот метод, который применялся очень широко в практическом программировании, и по сей день не потерял своего значения для определенного класса задач.

Структурный подход базируется на двух основополагающих принципах:

использование процедурного стиля программирования;

последовательная декомпозиция (разложение) задачи сверху вниз.

Задача решается путем выполнения некоторой последовательности действий. Первоначально она формулируется в терминах входа — выхода (на вход программы подаются исходные данные, а после выполнения программы не выход выдается ответ). Далее начинается последовательное разложение всей задачи на более простые действия. Например, если нам необходимо написать программу проверки правильности адреса, то вначале мы ее запишем следующим образом.

1. Прочитать адрес.
2. Сверить адрес с базой имеющихся адресов.
3. Если результат проверки положителен, напечатать «Да», в противном случае напечатать «Нет».

Такая запись один к одному отображается в программе на языке высокого уровня, например на Pascal или C.

Чрезвычайно важно то, что на любом этапе программу можно проверить, достаточно лишь написать заглушки — процедуры, имитирующие вход и выход процедур нижнего уровня. В приведенной выше программе можно использовать процедуру чтения адреса, которая вместо ввода с терминала просто подставляет какой-нибудь фиксированный адрес, и процедуру сверки с базой данных, которая ничего не делает, а просто всегда возвращает код корректного завершения.

Программа компонуется с заглушками и может работать. Иными словами, заглушки позволяют проверить логику верхнего уровня до реализации следующего уровня. Последовательно применяя метод заглушек, можно на каждом шаге разработки программы иметь работающий скелет, который постепенно обрастает деталями.

Структурное программирование поддерживалось языками программирования, появившимися в конце 1960-х годов (Pascal, Algol-68, Fortran и др.). Именно к тому времени было осознано значение программного обеспечения при решении задач с помощью вычислительной техники. Эти языки поддерживали разнообразные вложенные процедуры, разные способы передачи параметров.

Несмотря на то что структурное программирование ясно определило значение модульного построения программ при разработке больших проектов, языки программирования еще слабо поддерживали модульность. Единственным способом структуризации программ являлось составление ее из подпрограмм или функций. Контроль за правильностью вызова функций, в том числе за соответствием количества и типов фактических аргументов ожидаемым формальным параметрам, осуществлялся только на стадии выполнения (понятие прототипа функции появилось позже).

Объектно-ориентированное программирование родилось и получило широкое распространение ввиду осознания трех важнейших проблем программирования.

Первая проблема состояла в том, что развитие языков и методов программирования, хотя и существенно облегчило и ускорило разработку программных систем, не успевало за растущими с еще большей скоростью потребностями в программах. Единственным реальным способом резко ускорить разработки был метод многократного использования разработанного программного обеспечения. Иными словами, требовалось не строить каждый раз систему с нуля, а использовать разработанные ранее модули, которые уже прошли цикл отладки и тестирования и успешно работали.

Разумеется, процедурное программирование предоставляло способ разработки функций, которые могли служить блоками для построения программ. Однако ни гибкость этого метода, ни масштабы использования не позволяли существенно ускорить массовое программирование.

Второй проблемой, фактически связанной с первой, являлась необходимость упрощения сопровождения и модификации разработанных систем. Факт постоянного изменения требований к системе был осознан как нормальное условие развития системы, а не как неумение или недостаточно четкая организация работы разработчиков. Сопровождение и модификация систем требуют не меньших усилий, чем собственно разработка. Требовалось радикально изменить способ построения программных систем, чтобы, во-первых, локальные модификации не могли нарушить работоспособность всей системы и, во-вторых, было легче производить изменения поведения системы.

Третья проблема, возможно наиболее существенная, которую требовалось решить, — это облегчение проектирования систем. Далеко не все задачи поддаются алгоритмическому описанию и, тем более, алгоритмической декомпозиции, как того требует структурное программирование. Было необходимо приблизить структуру программ к структуре решаемых задач, сократить так называемый семантический разрыв между ними. О семантическом разрыве говорят в том случае, когда понятия, лежащие в основе языка задачи и средств ее решения, различны. Поэтому наряду с необходимостью записи самого решения требуется еще перевести одни понятия в другие. Сравните это с переводом с одного естественного языка на другой. Именно потому, что таких понятий в русском языке раньше не было, появляются слова типа «брокер», «оффшор» или «инвестор». К сожалению, в программировании заимствование слов невозможно.

Итак, упрощение проектирования, ускорение разработки за счет многократного использования готовых модулей и легкость модификации — вот три основных достоинства объектно-ориен-

рованного программирования, которые пропагандировались его сторонниками.

Объектно-ориентированный подход к программированию связывают прежде всего с языками программирования, такими как Smalltalk, C++, Java и т.д. Эта позиция имеет существенные основания. Поскольку языки являются главными инструментами объектно-ориентированного программирования, именно при их разработке появилось большинство тех идей, которые и составляют основу объектно-ориентированного метода в настоящее время.

Накопление новых идей шло в течение примерно 10 лет, начиная с конца 1960-х годов. К концу 1970-х годов переход на новый уровень абстракции стал свершившимся фактом в академических кругах. Потребовалось еще около 10 лет, чтобы новые идеи проникли в промышленность. Пожалуй, первым шагом на пути создания собственно объектной модели следует считать появление абстрактных типов данных.

Первым на необходимость структуризации систем по уровням абстракции указал Э.Дейкстра. Идея инкапсуляции (скрытия информации) была высказана Д.Парнасом. Позднее был разработан механизм абстрактных типов данных, который был дополнен С.Хоором в его теории типов и подтипов.

Считается, что первой полной реализацией абстрактных типов данных в языках программирования является язык Simula, который, в свою очередь, опирается на языки Modula, CLU, Euclid и др. Первым «настоящим» объектно-ориентированным языком программирования принято считать Smalltalk, разработанный в лаборатории компании «Ксерокс» в Паоло-Альто. (Многие по-прежнему считают его единственным настоящим объектным языком программирования.) Затем появились (и продолжают появляться) другие объектно-ориентированные языки, которые определяют современное состояние программирования. Наиболее распространенными из них стали C++, CLOS, Eiffel, Java.

Однако только языками программирования объектно-ориентированный подход не исчерпывается. Языки лишь предоставляют инструментарий, которым можно воспользоваться или не воспользоваться. На языке C++ можно писать в процедурном стиле. Объектно-ориентированное программирование предполагает единый подход к проектированию, построению и развитию системы.

Появление объектно-ориентированного метода произошло на основе всего предыдущего развития методов разработки программного обеспечения, а также многих других отраслей науки. Американский эксперт в области программирования Гради Буч (Grady Booch) помимо развития языков программирования отмечает следующие достижения, которые способствовали возникновению объектно-ориентированного подхода к проектированию систем.

1. Развитие вычислительной техники, в частности аппаратная поддержка основных концепций операционных систем и построение функционально-ориентированных систем. При проектировании вычислительных машин использование понятия объекта началось с исследований по нефоннеймановской архитектуре. Попытки сократить семантический разрыв между низкоуровневой архитектурой традиционных процессоров и высокоуровневыми понятиями операционных систем предпринимались при построении таких систем, как Барроус, Intel i432, IBM/38 и др. Тесно связанные с ними разработки объектно-ориентированных операционных систем начались, пожалуй, с проекта THE под руководством Э. Дейкстры, были продолжены в системах iMAX для Intel i432 и др. Сравнительно недавние проекты Cairo Microsoft и Taligent Pink (хотя и остановившиеся на исследовательском этапе) — это также проекты объектно-ориентированных операционных систем.

2. Достижения в методологии программирования, в частности модульное построение систем и инкапсуляция информации.

3. Создание теории построения и моделирования систем управления базами данных, которая внесла в объектное программирование идеи построения отношений между объектами. Моделирование данных с помощью отношений между объектами было впервые предложено П. Ченом (P. Chen) в методе ER-моделирования. В этом методе модель данных строится в виде объектов (сущностей), их атрибутов и отношений между ними.

4. Исследования в области искусственного интеллекта, позволившие лучше осознать механизмы абстракции. Теория фреймов, предложенная М. Минским для представления реальных объектов в системах распознавания образов, дала мощный импульс для развития не только систем искусственного интеллекта, но и механизмов абстракции в языках программирования.

5. Развитие философии и теории познания. Объектно-ориентированное построение систем — это определенный взгляд на моделируемый реальный мир. Именно в этом аспекте философия и теория познания оказали сильное влияние на объектную модель. Еще древние греки рассматривали мир в виде объектов или процессов. Декарт выдвинул предположение, что для человека естественным представляется объектно-ориентированное рассмотрение окружающего мира. М. Минский предположил, что разум проявляется как взаимодействие агентов, не умеющих по отдельности мыслить.

Теория архитектуры и строительства, выдвинувшая концепцию образцов или шаблонов, активно используется в последние годы в области объектно-ориентированного анализа (ООА) и объектно-ориентированного проектирования (ООП).

Следствием интенсивных исследований в области методов программирования явилось появление большого числа средств авто-

тизации разработки программ, или CASE-средств (Computer ided Software Engineering). Предполагалось, что после записи задачи на каком-либо высокоуровневом языке эти системы автоматически (или хотя бы с минимальным участием человека) сгенерируют готовую, правильно работающую программу, которая адекватно решит поставленную задачу.

В целом CASE-средства не смогли достичь желаемого: либо описание задачи по сложности оказывалось сравнимым с результирующей программой, либо решался крайне ограниченный круг равнительно простых задач, либо качество генерируемой программы оказывалось слишком низким.

Несмотря на провал при попытке достичь основной цели, опыт разработки CASE-средств дал очень много для развития методов программирования. Небольшая часть систем используется напрямую (например, генераторы компиляторов lex и yacc), часть идей была использована при создании современных средств быстрой разработки программ (RAD — Rapid Application Development), таких как JBuilder, Delphi, Powerbuilder и др.

Приведенный исторический экскурс развития методологии программирования, совершенно не претендующий на полноту, иллюстрирует то, что все время программирование «боролось» за возможность решать все более сложные задачи, создавать все более сложные программные системы и делать это как можно быстрее. Периодически то или иное достижение объявлялось панацеей (будь то структурное программирование, CASE-средства или что-либо еще). Однако всегда оказывалось, что широко разрекламированное средство не способно решить все проблемы.

8.3. Структурное программирование

Появление первых ЭВМ ознаменовало новый этап в развитии техники вычислений. Возникла идея, что достаточно разработать последовательность элементарных действий, каждое из которых преобразовать в понятные ЭВМ инструкции, и любая вычислительная задача может быть решена. Этот подход оказался настолько жизнеспособным, что долгое время доминировал над всеми другими в процессе разработки программ. Появились специальные языки программирования, которые позволили преобразовывать отдельные вычислительные операции в соответствующий программный код.

Основой данной методологии разработки программ стала процедурная или алгоритмическая организация структуры программного кода. Это было настолько естественно для решения вычислительных задач, что ни у кого не вызывала сомнений целесообразность такого подхода. Исходным в этой методологии являлось по-

нятие «алгоритм», под которым в общем случае подразумевается некоторое предписание выполнить точно определенную последовательность действий, направленных на достижение заданной цели или решение поставленной задачи.

С этой точки зрения вся история математики тесно связана с разработкой тех или иных алгоритмов решения актуальных для своей эпохи задач. Более того, само понятие «алгоритм» стало предметом соответствующей теории — теории алгоритмов, которая занимается изучением их общих свойств. Со временем содержание этой теории стало настолько абстрактным, что соответствующие результаты понимали только специалисты. Как дань этой традиции какой-то период времени языки программирования назывались алгоритмическими, а первое графическое средство документирования программ получило название «блок-схема алгоритма». Соответствующая система графических обозначений была зафиксирована в ГОСТ 19.701—90, который регламентировал использование условных обозначений в схемах алгоритмов, программ, данных и систем.

Однако потребности практики не всегда требовали установления вычислимости конкретных функций или разрешимости отдельных задач. В языках программирования возникло и закрепилось новое понятие — «процедура», которое конкретизировало общее понятие «алгоритм» применительно к решению задач на компьютерах. Так же, как и алгоритм, процедура представляет собой законченную последовательность действий или операций, направленных на решение отдельной задачи. В языках программирования появилась специальная синтаксическая конструкция, которая получила название «процедура».

Со временем разработка больших программ превратилась в серьезную проблему и потребовала их разбиения на более мелкие фрагменты. Основой для такого разбиения и стала процедурная декомпозиция, при которой отдельные части программы, или модули, представляли собой совокупность процедур для решения некоторой совокупности задач. Главная особенность процедурного программирования заключается в том, что программа всегда имеет начало во времени, или начальную процедуру (начальный блок), и окончание (конечный блок). При этом вся программа может быть представлена визуально в виде направленной последовательности графических примитивов, или блоков

Важным свойством таких программ является необходимость завершения всех действий предшествующей процедуры для начала действий последующей процедуры. Изменение порядка выполнения этих действий даже в пределах одной процедуры потребовало включения в языки программирования специальных условных операторов типа if-then-else и go to для реализации ветвления

вычислительного процесса в зависимости от промежуточных результатов решения задачи.

Появление и интенсивное использование условных операторов и оператора безусловного перехода стало предметом острых дискуссий среди специалистов по программированию. Дело в том, что бесконтрольное применение в программе оператора безусловного перехода go to способно серьезно осложнить понимание кода. Соответствующие программы стали сравнивать со спагетти, называя их bowl of spaghetti, имея в виду многочисленные переходы от одного фрагмента программы к другому или, что еще хуже, возврат от конечных операторов программы к ее начальным операторам.

Ситуация казалась настолько драматичной, что в литературе зазвучали призывы исключить оператор go to из языков программирования. Именно с этого времени принято считать хорошим стилем программирование без оператора go to.

Рассмотренные идеи способствовали становлению некоторой системы взглядов на процесс разработки программ и написания программных кодов, которая получила название *методология структурного программирования*. Основой данной методологии является процедурная декомпозиция программной системы и организация отдельных модулей в виде совокупности выполняемых процедур. В рамках данной методологии получило развитие нисходящее проектирование программ, или программирование «сверху вниз». Период наибольшей популярности идей структурного программирования приходится на конец 1970-х — начало 1980-х годов.

В качестве вспомогательного средства структуризации программного кода было рекомендовано использование отступов в начале каждой строки, которые должны выделять вложенные циклы и условные операторы. Все это призвано способствовать пониманию или читабельности самой программы. Данное правило со временем было реализовано в современных инструментариях разработки программ.

8.4. Объектно-ориентированное проектирование

Основными понятиями объектно-ориентированного подхода являются объект и класс.

Объект определяется как осязаемая реальность (tangible entity), т.е. предмет или явление, имеющие четко определяемое поведение. Объект характеризуется состоянием, поведением и индивидуальностью; структура и поведение схожих объектов определяют общий для них класс. Термины «экземпляр класса» и «объект» являются эквивалентными. Состояние объекта характеризуется

перечнем всех возможных (статических) свойств данного объекта и текущими (динамическими) значениями каждого из этих свойств. Поведение объекта характеризуется его воздействием на другие объекты и наоборот с точки зрения изменения состояния этих объектов и передачи сообщений. Иначе говоря, поведение объекта полностью определяется его действиями. Индивидуальность — это свойства объекта, отличающие его от всех других объектов.

Определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию называется *операцией*. Как правило, в объектных и объектно-ориентированных языках операции, выполняемые над данным объектом, называются методами и являются составной частью определения класса.

Класс — это множество объектов, связанных общностью структуры и поведения. Любой объект является экземпляром класса. Определение классов и объектов — одна из самых сложных задач объектно-ориентированного проектирования.

Следующую группу важных понятий объектно-ориентированного подхода составляют наследование и полиморфизм. *Полиморфизм* можно интерпретировать как способность класса принадлежать более чем одному типу. *Наследование* означает построение новых классов на основе существующих с возможностью добавления или переопределения данных и методов.

Объектно-ориентированная система изначально строится с учетом ее эволюции. Наследование и полиморфизм обеспечивают возможность определения новой функциональности классов с помощью создания производных классов — потомков базовых классов. Потомки наследуют характеристики родительских классов без изменения их первоначального описания и добавляют при необходимости собственные структуры данных и методы. Определение производных классов, при котором задаются только различия или уточнения, в огромной степени экономит время и усилия при производстве и использовании спецификаций и программного кода.

Принципиальное различие между структурным и объектно-ориентированным подходом заключается в способе декомпозиции системы. Объектно-ориентированный подход использует объектную декомпозицию, при этом статическая структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами. Каждый объект системы обладает своим собственным поведением, моделирующим поведение объекта реального мира.

Понятие «объект» впервые было использовано около 30 лет назад при попытках отойти от традиционной архитектуры фон Неймана и преодолеть барьер между высоким уровнем программных абстракций и низким уровнем абстрагирования. С объектно-

ориентированной архитектурой также тесно связаны объектно-ориентированные операционные системы. Наиболее значительный вклад в объектный подход был внесен объектными и объектно-ориентированными языками программирования Simula, Smalltalk, C++, Object Pascal. На объектный подход оказали влияние также развивавшиеся достаточно независимо методы моделирования баз данных, в особенности подход сущность—связь.

Концептуальной основой объектно-ориентированного подхода служит объектная модель. Основными ее элементами являются абстрагирование (abstraction), инкапсуляция (encapsulation), модульность (modularity), иерархия (hierarchy).

Кроме основных имеются еще три дополнительных элемента — типизация (typing), параллелизм (concurrency), устойчивость (persistence), не являющихся в отличие от основных строго обязательными.

Абстрагирование — это выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы для дальнейшего рассмотрения и анализа. Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности его поведения от деталей их реализации. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования.

Инкапсуляция — это процесс отделения друг от друга отдельных элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы изолировать интерфейс объекта, отражающий его внешнее поведение, от внутренней реализации объекта. Объектный подход предполагает, что собственные ресурсы, которыми могут манипулировать только методы самого класса, скрыты от внешней среды.

Абстрагирование и инкапсуляция являются взаимодополняющими операциями: абстрагирование фокусирует внимание на внешних особенностях объекта, а инкапсуляция не позволяет объектам-пользователям видеть внутреннее устройство объекта.

Модульность — это свойство системы, обусловленное возможностью ее декомпозиции на ряд внутренне связанных, но слабо объединенных между собой модулей. Инкапсуляция и модульность создают барьеры между абстракциями.

Иерархия — это ранжированная, или упорядоченная, система абстракций, расположение их по уровням. Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия по номенклатуре) и структура объектов (иерархия по составу). Примерами иерархии классов являются простое и множественное наследование (один класс

использует структурную или функциональную часть соответственно одного или нескольких других классов), а примером иерархии объектов — агрегация.

Типизация — это ограничение, накладываемое на класс объектов и препятствующее взаимозаменяемости различных классов (или сильно сужающее ее возможность). Типизация позволяет защититься от использования объектов одного класса вместо другого или, по крайней мере, управлять таким использованием.

Параллелизм — это свойство объектов находиться в активном или пассивном состоянии и различать активные и пассивные объекты между собой.

Устойчивость — это свойство объекта существовать во времени (независимо от процесса, породившего данный объект) и/или в пространстве (при перемещении объекта из адресного пространства, в котором он был создан).

Важным качеством объектно-ориентированного подхода является согласованность моделей деятельности организации и моделей проектируемой системы, начиная со стадии формирования требований и до стадии реализации. Требование согласованности моделей выполняется благодаря возможности применения абстрагирования, модульности, полиморфизма на всех стадиях разработки. Модели ранних стадий могут быть непосредственно подвергнуты сравнению с моделями реализации. По объектным моделям может быть прослежено отображение реальных сущностей моделируемой предметной области (организации) в объекты и классы информационной системы.

Большинство существующих методов объектно-ориентированного анализа и проектирования (ООАП) включают в себя как язык моделирования, так и описание процесса моделирования. Язык моделирования — это нотация (в основном графическая), которая используется методом для описания проектов. Графическая нотация представляет собой совокупность графических объектов, которые используются в моделях; она является синтаксисом языка моделирования. Например, нотация диаграммы классов определяет, каким образом представляются такие элементы и понятия, как класс, ассоциация и множественность. Процесс — это описание шагов, которые необходимо выполнить при разработке проекта.

Унифицированный язык моделирования UML (Unified Modeling Language) — это преемник того поколения методов ООАП, которые появились в конце 1980-х — начале 1990-х годов. Создание UML фактически началось в конце 1994 г., когда Гради Буч и Джеймс Рамбо приступили к работе по объединению методов Booch и OMT (Object Modeling Technique) с помощью А.Джекобсона и под эгидой компании Rational Software. К концу 1995 г. они создали первую спецификацию объединенного метода, названного ими Unified Method, версия 0.8. Тогда же, в 1995 г., к ним присоединился со-

здатель метода OOSE (Object-Oriented Software Engineering) Ивар Якобсон. Таким образом, UML является прямым объединением и унификацией методов Г. Буча, Д. Рамбо, А. Джекобсона и И. Якобсона, однако дополняет их новыми возможностями. Главными в разработке UML были следующие цели:

предоставить пользователям готовый к применению выразительный язык визуального моделирования, позволяющий разрабатывать осмысленные модели и обмениваться ими;

предусмотреть механизмы расширения и специализации для более точного представления моделей систем в конкретной предметной области;

обеспечить независимость от конкретных языков программирования и процессов разработки;

обеспечить формальную основу для понимания этого языка моделирования (язык должен быть одновременно точным и доступным для понимания, без лишнего формализма);

стимулировать рост рынка объектно-ориентированных инструментальных средств;

интегрировать лучший практический опыт.

Язык UML находится в процессе стандартизации, проводимом OMG (Object Management Group) — организацией по стандартизации в области объектно-ориентированных методов и технологий. В настоящее время он принят в качестве стандартного языка моделирования и получил широкую поддержку в индустрии разработки ПП. Язык UML принят на вооружение практически всеми крупнейшими компаниями — производителями ПП (Microsoft, IBM, Hewlett-Packard, Oracle, Sybase и др.). Кроме того, практически все мировые производители CASE-средств помимо Rational Software (Rational Rose) поддерживают UML в своих продуктах (Paradigm Plus 3.6, System Architec, Microsoft Visual Modeler for Visual Basic, Delphi, PowerBuilder и др.). Полное описание UML можно найти на сайтах http:// www.omg.org, http://www.rational.com и http://uml.shl.com.

Создатели UML представляют его как язык для определения, представления, проектирования и документирования программных, организационно-экономических, технических и других систем. Подробнее язык UML описан в гл. 14.

8.5. Собираемые метрики, используемые методы, стандарты и шаблоны

На этапе проектирования ПП необходимо выполнять оценки расхождений плановых сроков и объемов с фактическими, числа проведенных обзоров, выявленных ошибок и дефектов, а также средних трудозатрат и производительности проектирования.

Все полученные данные следует хранить в ИБД проектной группы.

Используемый инструмент: система подготовки документов (например, MS Word).

Используемые методы и стандарты: процесс организации; метрическая программа организации.

Используемые шаблоны: описания результатов высокоуровневого проектирования; отчета по обзору; отчета о статусе проекта.

Контрольные вопросы

1. Каково назначение этапа проектирования в жизненном цикле разработки программного продукта?
2. Что является компонентой проектирования и какие для нее могут быть определены атрибуты?
3. Дайте определение понятия «алгоритм».
4. Какие основные принципы положены в основу структурного подхода к программированию?
5. Дайте определите понятия «программная заглушка». Каково ее назначение?
6. Какие проблемы программирования способствовали широкому распространению объектно-ориентированного подхода к программированию?
7. Какие достижения технологии способствовали возникновению объектно-ориентированного подхода к программированию?
8. Для чего предназначены CASE-средства, или CASE-технологии?
9. Дайте определение понятия «объект».
10. Что такое абстрагирование?
11. Что такое инкапсуляция?
12. Дайте определение понятия «класс».
13. Каково назначение языка UML?
14. Какие метрики собирают на этапе проектирования?
15. Какие методы, стандарты и шаблоны используют на этапе проектирования?

ГЛАВА 9

ЭТАП РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА

9.1. Кодирование

На этапе разработки ПП выполняются следующие основные действия: кодирование; тестирование; разработка справочной системы ПП; создание документации пользователя; создание версии и инсталляции ПП.

Кодирование представляет собой процесс преобразования результатов высокоуровнего и низкоуровнего проектирования в готовый программный продукт. Другими словами, при кодировании происходит описание составленной модели ПП средствами выбранного языка программирования, которым может быть лю-

Пример оформления заголовка программы:

```
;*****************************************************
;*Description:                                       *
;*                                                   *
;* File Listing template for CPU32 has been developed*
;* with MASM v.5.0.                                  *
;*                                                   *
;* Creation Date: 21/04/95 From: New                 *
;*                                                   *
;* Author: IBS Russia                                *
;*                                                   *
;* Update By Modification:                           *
;*                                                   *
;* 06/05/95 AR Add changes according to R.Soja and   *
;* B.Calvert comments / letter from 06/05/95 / for   *
;* MASM 5.0                                          *
;*                                                   *
;*                                                   *
;*          **** This Revision: 1.11 ****            *
;*                                                   *
;* Last Modified: 06/05/95 By: Alexandr Rudakov      *
;*****************************************************
```

бой из существующих языков. Выбор языка осуществляется либо по желанию заказчика, либо с учетом решаемой задачи и личного опыта разработчиков.

При кодировании необходимо следовать стандарту на выбранный язык, например, для языка C — это ANSI C, а для C++ — ISO/IEC 14882 «Standart for the C++ Programming Language».

Кроме общепринятого стандарта на язык программирования в компании могут использоваться разработаны и свои дополнительные требования к правилам написания программ. В основном они касаются правил оформления текста программы.

Следование стандарту и правилам компании позволяет создать корректно работающую, легко читаемую, понятную другим разработчикам программу, содержащую сведения о разработчике, дату создания, имя и назначение, а также и необходимые данные для управления конфигурацией.

9.2. Тестирование

На этапе кодирования программист пишет программы и сам их тестирует. Такое тестирование называется модульным. Все вопросы, связанные с тестированием ПП, рассмотрены в гл. 10, здесь же описана технология тестирования, которая применяется на этапе разработки ПП. Эта технология называется тестированием «*стеклянного ящика*» (*glass box*); иногда ее еще называют тестированием «*белого ящика*» (*white box*) в противоположность классическому понятию «черного ящика» (black box).

При тестировании «черного ящика» программа рассматривается как объект, внутренняя структура которого неизвестна. Тестировщик вводит данные и анализирует результат, но он не знает, как именно работает программа. Подбирая тесты, специалист ищет интересные с его точки зрения входные данные и условия, которые могут привести к нестандартным результатам. Интересны для него прежде всего те представители каждого класса входных данных, при которых с наибольшей вероятностью могут проявиться ошибки тестируемой программы.

При тестировании «стеклянного ящика» ситуация совершенно иная. Тестировщик (в данном случае сам программист) разрабатывает тесты, основываясь на знании исходного кода, к которому он имеет полный доступ. В результате он получает следующие преимущества.

1. Направленность тестирования. Программист может тестировать программу по частям, разрабатывать специальные тестовые подпрограммки, которые вызывают тестируемый модуль и передают ему интересующие программиста данные. Отдельный модуль гораздо легче протестировать именно как «стеклянный ящик».

2. Полный охват кода. Программист всегда может определить, какие именно фрагменты кода работают в каждом тесте. Он видит, какие еще ветви кода остались непротестированными, и может подобрать условия, в которых они будут протестированы. Ниже описано, как отслеживать степень охвата программного кода проведенными тестами.

3. Возможность управления потоком команд. Программист всегда знает, какая функция должна выполняться в программе следующей и каким должно быть ее текущее состояние. Чтобы выяснить, работает ли программа так, как он думает, программист может включить в нее отладочные команды, отображающие информацию о ходе ее выполнения, или воспользоваться для этого специальным программным средством, называемым отладчиком. Отладчик может делать очень много полезных вещей: отслеживать и менять последовательность выполнения команд программы, показывать содержимое ее переменных и их адреса в памяти и др.

4. Возможность отслеживания целостности данных. Программисту известно, какая часть программы должна изменять каждый элемент данных. Отслеживая состояние данных (с помощью того же отладчика), он может выявить такие ошибки, как изменение данных не теми модулями, их неверная интерпретация или неудачная организация. Программист может и самостоятельно автоматизировать тестирование.

5. Видение внутренних граничных точек. В исходном коде видны те граничные точки программы, которые скрыты от взгляда извне. Например, для выполнения определенного действия может быть использовано несколько совершенно различных алгоритмов, и, не заглянув в код, невозможно определить, какой из них выбрал программист. Еще одним типичным примером может быть проблема переполнения буфера, используемого для временного хранения входных данных. Программист сразу может сказать, при каком количестве данных буфер переполнится, и ему не нужно при этом проводить тысячи тестов.

6. Возможность тестирования, определяемого выбранным алгоритмом. Для тестирования обработки данных, использующей очень сложные вычислительные алгоритмы, могут понадобиться специальные технологии. В качестве классических примеров можно привести преобразование матрицы и сортировку данных. Тестировщику, в отличие от программиста, нужно точно знать, какие алгоритмы используются, поэтому приходится обращаться к специальной литературе.

Тестирование «стеклянного ящика» — часть процесса программирования. Программисты выполняют эту работу постоянно, они тестируют каждый модуль после его написания, а затем еще раз после интеграции его в систему.

При выполнении модульного тестирования можно использовать технологию либо структурного, либо функционального тестирования или и ту, и другую.

Структурное тестирование является одним из видов тестирования «стеклянного ящика». Его главной идеей является правильный выбор тестируемого программного пути. В противоположность ему *функциональное* тестирование относится к категории тестирования «черного ящика». Каждая функция программы тестируется путем ввода ее входных данных и анализа выходных. При этом внутренняя структура программы учитывается очень редко.

Хотя структурное тестирование имеет гораздо более мощную теоретическую основу, большинство тестировщиков предпочитают функциональное тестирование. Структурное тестирование лучше поддается математическому моделированию, но это совсем не означает, что оно эффективнее. Каждая из технологий позволяет выявить ошибки, пропускаемые в случае использования другой. С этой точки зрения их можно назвать одинаково эффективными.

Объектом тестирования может быть не только полный путь программы (последовательность команд, которые она выполняет от старта до завершения), но и его отдельные участки. Протестировать все возможные пути выполнения программы абсолютно нереально. Поэтому специалисты по тестированию выделяют из всех возможных путей те группы, которые нужно протестировать обязательно. Для отбора они пользуются специальными критериями, называемыми *критериями охвата* (*coverage criteria*), которые определяют вполне реальное (пусть даже и достаточно большое) число тестов. Данные критерии иногда называют *логическими критериями охвата*, или *критериями полноты*.

Рассмотрим три критерия, которые используются тестировщиками чаще всего: охвата строк, охвата ветвлений и охвата условий. Когда тестирование организовано в соответствии с этими критериями, о нем говорят как о тестировании путей.

Критерий охвата строк — наиболее слабый из трех. Он требует, чтобы каждая строка кода была выполнена хотя бы один раз. И хотя многие программисты не удосуживаются соблюсти и это требование, для серьезного тестирования программы его далеко не достаточно. Если строка содержит оператор принятия решения, должны быть проверены все управляющие решением значения. Для примера рассмотрим следующий фрагмент кода:

```
IF (A < B and C = 5)
THEN Сделать НЕЧТО
SET D = 5
```

Чтобы проверить этот код, нужно проанализировать следующие четыре варианта.

а) A < B и C = 5 (НЕЧТО выполняется, затем D присваивается значение 5)

б) A < B и C ≠ 5 (НЕЧТО не выполняется, D присваивается значение 5)

в) A ≥ B и C = 5 (НЕЧТО не выполняется, D присваивается значение 5)

г) A > B и C ≠ 5 (НЕЧТО не выполняется, D присваивается 5).

Для выполнения всех трех строк кода достаточно проверить вариант а).

При более основательном способе тестирования — по критерию охвата ветвлений — программист проверяет вариант а) и еще один из трех остальных вариантов. Смысл этого способа в том, что проверяются действия программы при выполненном условии оператора IF и при невыполненном. Таким образом, программа проходит не только все строки кода, но и все возможные ветви.

Иногда охват ветвлений называют полным охватом кода. Но термин этот некорректен: охват ветвлений не может претендовать на полноту тестирования, поскольку в лучшем случае позволяет обнаружить половину имеющихся в программе ошибок.

Еще более строгим является критерий охвата условий. По этому критерию следует проверить все составляющие каждого логического условия. В приведенном примере это означает проверку всех четырех перечисленных вариантов.

Тестирование путей программы считается завершенным, когда выбранный критерий охвата полностью выполнен. Для автоматизации этого процесса разработаны программы, анализирующие программный код и вычисляющие число подлежащих тестированию путей, а затем подсчитывающие, сколько их уже проверено. Такие программы называют средствами мониторинга охвата (execution coverage monitors).

Как правило, в процессе тестирования путей не поощряется проверка одного и того же пути при различных данных. Хотя варианты б), в) и г) приведенного примера могут оказаться важными, большинство средств мониторинга охвата посчитают их проверку пустой тратой времени. Все три начинаются с одного и того же оператора и приводят к выполнению одной и той же последовательности команд, поэтому проверять их все означает трижды проверять один и тот же путь.

Хотя критерии охвата очень полезны, одного только тестирования путей недостаточно для эффективного выявления ошибок. Так, И.Гуденаф (Goodenough) и К.Герхарт (Gerhart) привели классический пример того, что проход строки кода еще не означает выявления имеющейся в ней ошибки. Рассмотрим следующую строку программы: SET A = B/C

Она вполне успешно выполняется, если C ≠ 0. Если C = 0, то программа или сообщит об ошибке и прекратит работу, или «за-

виснет». Разница между этими двумя вариантами не в пути, а в данных.

Де Милло (De Millo) называет программный путь чувствительным к ошибкам, если при его прохождении ошибки могут проявиться. Если же ошибки обязательно проявятся при прохождении данного пути, то такой путь называется обнаруживающим ошибки. Любой путь, проходящий через приведенную строку программы, чувствителен к ошибкам, ошибка же проявляется только при нулевом значении переменной С. Для выявления подобных ошибок технология тестирования «черного ящика» подходит лучше, чем анализ программного кода.

Формальное описание технологии тестирования программных путей можно найти у П. Реппса (Rapps) и С. Вейакера (Weyuker), более подробное исследование проблемы — у Р. Бейзера (Beizer), а особенно детальное обсуждение критериев охвата — у Д. Майерса (Myers).

Любая система разрабатывается по частям — как набор процессов или модулей. Можно ее так и тестировать, т. е. сначала проверять отдельные части, а потом уже их взаимодействие. Такое тестирование называется *восходящим* (*bottom-up*).

Выяснив, что отдельные элементы программы в порядке, специалист приступает к тестированию их совместной работы. И тут может оказаться, что вместе они работать отказываются. Например, если программист случайно поменяет местами параметры вызываемой функции, то при выполнении вызова произойдет ошибка. И выявлена она будет только при проверке совместной работы обеих функций — вызывающей и вызываемой.

Тестирование совместной работы программных модулей называют *интеграционным*. В ходе такого тестирования модули сначала объединяют в пары, потом в большие блоки, пока, наконец, все модули не будут объединены в единую систему. Интеграционное тестирование всей системы выполняет тестировщик, но интеграционное тестирование какой-либо подзадачи — разрабатывающий ее программист.

Восходящее тестирование — это прекрасный способ локализации ошибок. Если ошибка обнаружена при тестировании единственного модуля, то очевидно, что она содержится именно в нем, и для поиска ее источника не нужно анализировать код всей системы. Если же ошибка проявляется при совместной работе двух предварительно протестированных модулей, значит, дело в их интерфейсе. Еще одним преимуществом восходящего тестирования является то, что выполняющий его программист концентрируется на очень узкой области (единственном модуле, передаче данных между парой модулей и т. п.). Благодаря этому тестирование проводится более тщательно и с большей вероятностью выявляет ошибки.

Главным недостатком восходящего тестирования является необходимость написания специального кода-оболочки, вызывающего тестируемый модуль. Если тот, в свою очередь, вызывает другой модуль, то для него нужно написать заглушку. *Заглушка* — это имитация вызываемой функции, возвращающая те же данные, но ничего больше не делающая.

Написание оболочек и заглушек замедляет работу, а для конечного продукта они абсолютно бесполезны, но созданные однажды, эти элементы могут использоваться повторно при каждом изменении программы. Хороший набор оболочек и заглушек — это очень эффективный инструмент тестирования.

В противоположность стратегии восходящего тестирования стратегия *целостного* тестирования предполагает, что до полной интеграции системы ее отдельные модули не проходят особо тщательного тестирования. Преимуществом такой стратегии является отсутствие необходимости написания дополнительного кода. Многие программисты выбирают этот способ тестирования из соображений экономии времени, считая, что лучше разработать один обширный набор тестов и с его помощью за один раз проверить всю систему. Но такое представление совершенно ошибочно по следующим причинам.

1. Очень трудно выявить источник ошибки. Это главная проблема. Поскольку ни один из модулей не проверен как следует, в большинстве из них есть ошибки. Поэтому вопрос не столько в том, в каком модуле произошла обнаруженная ошибка, сколько в том, какая из ошибок во всех вовлеченных в процесс модулях привела к полученному результату. Когда накладываются ошибки нескольких модулей, бывает гораздо труднее локализовать источник ошибки.

Кроме того, ошибка в одном из модулей может блокировать тестирование другого. Как протестировать функцию, если вызывающий ее модуль не работает? Если не написать для этой функции программу-оболочку, то придется ждать отладки модуля, а это может занять много времени.

2. Трудно организовать исправление ошибок. Если программу пишут несколько программистов (а именно так и бывает в случае больших систем) и при этом неизвестно, в каком модуле ошибка, кто же должен ее искать и исправлять? Первый программист будет указывать на второго, тот, выяснив, что его код ни при чем, переложит ответственность на первого, а в результате сильно снизится скорость разработки.

3. Процесс тестирования трудно автоматизировать. То, что на первый взгляд кажется преимуществом целостного тестирования, а именно отсутствие необходимости создавать оболочки и заглушки, на самом деле оборачивается его недостатком. В процессе разработки программа ежедневно меняется и ее приходится тестиро-

вать снова и снова, а оболочки и заглушки помогают автоматизи ровать этот однообразный труд.

Когда кто-нибудь из программистов выбирает целостное тес тирование, можно предположить, что он видит одному ему ведо мые преимущества этого подхода. Есть и такие программисты, которые вообще не слишком заботятся об эффективности тести рования.

Главное для них — как можно скорее отрапортовать начальству о завершении работ, даже если на самом деле ничего не работает. Если после этого с проектом возникнут проблемы, то они будут обвинять законы Мэрфи, тестировщиков, невезение, но только не самих себя, собственную часть работы они будут считать выполненной успешно и в срок.

Существует еще один принцип организации тестирования, когда программа, как и при восходящем способе, тестируется не целиком, а по частям, только направление движения меняется — сначала тестируется самый верхний уровень иерархии модулей, а от него тестировщик постепенно спускается вниз. Такое тестирование называется *нисходящим* (*top-down*). И нисходящее, и восходящее тестирования называют также *инкрементальными*.

При нисходящем тестировании отпадает необходимость в написании оболочек, но заглушки остаются. По мере тестирования заглушки по очереди заменяются на реальные модули.

Мнения специалистов о том, какое из двух инкрементальных тестирований более эффективно, сильно расходятся. Р. Иордан (Yourdon) доказывает, что гораздо лучше нисходящее тестирование, а Д. Майерс (Myers) утверждает, что, хотя у обоих подходов есть свои преимущества и недостатки, в целом восходящее тестирование лучше. По мнению же Д. Данна (Dunn), эти способы примерно эквивалентны.

На практике вопрос выбора стратегии тестирования обычно решается просто: каждый модуль по возможности тестируется сразу после его написания, в результате последовательность тестирования одних частей программы может оказаться восходящей, а других — нисходящей.

При *статическом* тестировании программный код вообще не выполняется — он тестируется только путем логического анализа.

Для статического анализа существует множество инструментальных средств. Самое известное из них — компилятор. Встретив синтаксическую ошибку или недопустимую операцию, компилятор выдает соответствующее сообщение. Ряд полезных сообщений (о повторяющихся именах переменных и других объектов, ссылках на необъявленные переменные и функции) выдает и компоновщик. Статический анализ программы может выполняться и людьми. Они читают исходный код, возможно, обсуждают его, и, как правило, находят достаточно много ошибок.

9.3. Разработка справочной системы программного продукта. Создание документации пользователя

Целесообразно одного из сотрудников проекта назначать техническим редактором документации. Этот сотрудник может выполнять и другую работу, но главной его задачей должен быть анализ документации, даже если ее разрабатывают и другие сотрудники.

Часто бывает так, что над созданием ПП работают несколько человек, но никто из них не несет полной ответственности за его качество. В результате ПП не только не выигрывает от того, что его разрабатывает большее число людей, но еще и проигрывает, поскольку каждый подсознательно перекладывает ответственность на другого и ожидает, что ту или иную часть работы выполнят его коллеги. Эту проблему и решает назначение редактора, несущего полную ответственность за качество и точность технической документации.

Справочная система ПП формируется на основе материала, разработанного для руководства пользователя. Формирует и создает ее ответственный за выполнение этой работы. Им может быть как технический редактор, так и один из разработчиков совместно с техническим редактором.

У хорошо документированного ПП имеются следующие преимущества.

1. Легкость использования. Если ПП хорошо документирован, то его гораздо легче применять. Пользователи его быстрее изучают, делают меньше ошибок, а в результате быстрее и эффективнее выполняют свою работу.

2. Меньшая стоимость технической поддержки. Когда пользователь не может разобраться, как выполнить необходимые ему действия, он звонит производителю ПП в службу технической поддержки. Содержание такой службы обходится очень дорого. Хорошее же руководство помогает пользователям решать возникающие проблемы самостоятельно и меньше обращаться в группу технической поддержки.

3. Высокая надежность. Непонятная или неаккуратная документация делает ПП менее надежным, поскольку его пользователи чаще допускают ошибки, им трудно разобраться, в чем их причина и как справиться с их последствиями.

4. Легкость сопровождения. Огромное количество денег и времени тратится на анализ проблем, которые порождены ошибками пользователей. Изменения, вносимые в новые выпуски ПП, зачастую являются просто сменой интерфейса старых функций. Они вносятся для того, чтобы пользователи, наконец, разобрались, как применять ПП, и перестали звонить в службу технической поддержки. Хорошее руководство в значительной степени

решает эту проблему, плохое же, наоборот, усложняет ее еще больше.

5. Упрощенная установка. После покупки ПП продукта пользователь должен установить его на свой компьютер. Даже если этот процесс полностью автоматизирован, пользователю предстоит ответить на ряд вопросов и принять решения относительно набора и расположения компонентов ПП и настройки его функций. А ведь у пользователя может не быть опыта по установке ПП и некоторые вопросы программы могут поставить его в тупик. Для компании это опять же будет связано с затратами на техническую поддержку. Поэтому четкие и понятные инструкции по установке ПП являются одной из наиболее важных составляющих его документации.

Кроме инструкций по установке программное обеспечение должно сопровождаться и инструкциями по удалению ПП из системы. В документации также должно поясняться, как изменить параметры настройки, добавить или удалить компоненты ПП и выполнить установку его новой версии поверх предыдущей.

6. Коммерческий успех. Качество документации является одним из факторов, определяющих коммерческий успех ПП. Дилерам, вооруженным хорошей документацией, легче демонстрировать ПП покупателям и рассказывать о его возможностях. Во многих обзорах программного обеспечения, печатаемых в профессиональной прессе, документации уделяется значительное внимание.

7. Достоверность информации. В документации не должно быть неверной информации, вводящей пользователей в заблуждение и заставляющей их тратить лишнее время и усилия.

9.4. Создание версии и инсталляции программного продукта

Создание инсталляции ПП. Это действие позволяет автоматизировать процесс установки ПП на компьютеры пользователей, предоставляя им при этом возможность выбора различных сценариев установки и обеспечивая корректность его дальнейшей работы. При инсталляции ПП как бы «погружается» в то программное окружение, в котором он должен работать. Кроме того, у разработчиков ПП появляется прекрасная возможность запретить несанкционированные «пиратские» установки (например, с помощью проверки серийного номера ПП). Процесс инсталляции ПП обязательно должен быть описан в руководстве пользователя либо в отдельной брошюре.

В процессе инсталляции ПП могут проявиться различные проблемы. Наиболее часто встречаются следующие.

1. Окружение, в котором инсталлируется ПП, не совпадает с окружением, для которого он спроектирован. Например, ПП может использовать функции, которые предоставляет только определенная версия операционной системы. Однако версия операционной системы либо сама операционная система, определяющая текущее окружение инсталлируемого ПП, может не обладать этими функциями. В этом случае после инсталляции ПП может не работать совсем либо некоторые его функции окажутся не реализованными. Для устранения этой проблемы в руководстве пользователя обязательно должны быть перечислены либо версии операционных систем, либо сами операционные системы, с которыми ПП будет работать корректно.

2. Новый ПП может сосуществовать со старым до тех пор, пока в организации, где он инсталлируется, не убедятся, что новый ПП работает так, как требуется. Инсталляция нового ПП при оставленном старом может привести к определенным проблемам, особенно, если новый и старый ПП не являются полностью независимыми, а имеют некоторые общие компоненты. Случаются ситуации, когда новый ПП вообще невозможно внедрить без удаления старого. При этом испытания нового ПП можно провести только тогда, когда старый ПП не функционирует. Если такая ситуация вероятна, то она также должна быть описана в руководстве пользователя с конкретными рекомендациями по ее устранению.

Управление созданием версий и поставками ПП. Такое управление необходимо для идентификации всех версий и поставок ПП и слежения за ними. Специально выделенный сотрудник проекта (ответственный за управление конфигурацией), отвечающий за управление версиями и поставками ПП, выполняет процедуры поиска старых и создания новых версий или поставок ПП, а также следит за тем, чтобы изменения не осуществлялись произвольно. Только в том случае, когда информация об изменениях в версиях вносится исключительно ответственным за управление конфигурацией, можно гарантировать согласованность версий.

Версией ПП называют экземпляр ПП, имеющий определенные отличия от других экземпляров этого же ПП. Новые версии могут отличаться функциональными возможностями, эффективностью или отсутствием ошибок, имевшихся в старых версиях. Некоторые версии имеют одинаковые функциональные возможности, однако разработаны под различные конфигурации аппаратного или программного обеспечения. Если отличия между версиями незначительны, то они называются *вариантами* одной версии.

Выходная версия (*release*), или *поставка ПП*, — это та версия, которая поставляется заказчику. В каждой выходной версии либо обязательно присутствуют новые функциональные возможности, либо она разработана под новую платформу. Число версий обычно намного превышает число поставок, поскольку версии создаются

в основном для внутреннего пользования и не поставляются заказчику.

В настоящее время для поддержки управления версиями разработано много разнообразных CASE-средств, с помощью которых осуществляются управление хранением каждой версии и контроль за допуском к компонентам ПП. Компоненты могут извлекаться из ПП для внесения в них изменений. После введения в ПП измененных компонентов получается новая версия, для которой с помощью системы управления версиями создается новое имя.

Идентификация версий. Любой большой ПП состоит из сотен компонентов, каждый из которых может иметь несколько версий. Процедуры управления версиями должны четко идентифицировать каждую версию компонента. Существуют три основных способа идентификации версий.

1. Нумерация версий. Каждый компонент имеет уникальный и явный номер версии. Этот способ идентификации используется наиболее широко.

2. Идентификация, основанная на значениях атрибутов. Каждый компонент идентифицируется именем, которое не является уникальным для разных версий, и набором значений атрибутов, разных для каждой версии компонента. Другими словами, версия компонента идентифицируется комбинацией имени и набора значений атрибутов.

3. Идентификация на основе изменений. Каждая версия ПП именуется так же, как в предыдущем способе, плюс даются ссылки на запросы на изменения, которые реализованы в данной версии ПП. Таким образом, версия ПП идентифицируется именем и теми изменениями, которые реализованы в компонентах.

Нумерация версий. По самой простой схеме нумерации версий к имени компонента или ПП добавляется номер версии. Например, обозначение MASM TPU 4.3 означает: версия 4.3 ассемблера TPU. Первая версия обычно получает номер 1.0, последующие ее варианты — 1.1, 1.2 и т.д. На каком-то этапе создается новая поставка — версия 2.0; нумерация вариантов этой версии начинается заново: 2.1, 2.2 и т.д. Такая линейная схема нумерации основана на предположении о последовательности создания версий. Подобный подход к идентификации версий поддерживается многими программными средствами управления версиями.

Данный способ идентификации версий достаточно прост, однако требует довольно большого количества информации для сопоставления версий с целью отслеживания различий между ними и связи между запросами на изменения и версиями. Поэтому поиск отдельной версии ПП или его компонента может быть достаточно трудным, особенно при отсутствии интеграции между базой данных конфигураций и системой хранения версий.

Идентификация, основанная на значениях атрибутов. Основная проблема способов явного именования версий заключается в том, что в данном случае не отображаются те признаки, которые можно использовать для идентификации версий. В качестве таких признаков могут выступать: имя заказчика; язык программирования; состояние разработки; аппаратная платформа; дата создания.

Если каждая версия определяется единым набором атрибутов, то нетрудно добавить новые версии, основанные на любой из существующих версий, поскольку они будут идентифицироваться единым набором значений атрибутов. При этом значения многих атрибутов новой версии будут совпадать со значениями атрибутов исходной версии; таким образом можно прослеживать взаимоотношения между версиями. Поиск версий осуществляется на основе значений атрибутов. При этом возможны такие запросы, как «самая последняя версия», «версия, созданная между определенными датами» и т.п. Например, обозначение версии MASM TPU, разработанной на языке C++ для использования под управлением Windows 95 в апреле 1996 г., будет выглядеть следующим образом: MASM TPU (язык = C++, платформа = Windows 95, дата = апрель 1996).

Идентификация, основанная на значениях атрибутов, системой управления версиями может применяться непосредственно. Однако более распространено использование только части имени версии, при этом база данных конфигураций поддерживает связь между значениями атрибутов и версиями ПП и компонентов.

Идентификация на основе изменений. Идентификация, основанная на значениях атрибутов, устраняет проблему поиска версий, свойственную простым способом нумерации, когда для поиска версии требуется знание ее атрибутов. Но в этом случае для регистрации взаимосвязей между версиями и изменениями необходимо использование отдельной системы управления изменениями.

Идентификация на основе изменений применяется, скорее, к ПП, чем к его компонентам; версии отдельных компонентов скрыты от пользователей системы управления конфигурацией. Каждое изменение в ПП описывается массивом изменений, где указаны изменения в отдельных компонентах, реализующие данное изменение ПП. Массивы изменений могут применяться последовательно таким образом, чтобы создать версию ПП, в которой реализованы все необходимые изменения. В этом случае не требуется точного обозначения версии. Ответственный за управления конфигурацией работает с системой управления версиями посредством системы управления изменениями.

Применение нескольких массивов изменений ПП должно быть согласовано, поскольку отдельные массивы изменений могут быть несовместимыми и их последовательное применение может при-

вести к появлению неработоспособного ПП. Кроме того, массивы изменений могут конфликтовать, если они предусматривают разные изменения в одном компоненте. Для устранения этих проблем применяются средства управления версиями, поддерживающие идентификацию на основе изменений, что позволяет установить точные правила согласованности последовательности версий ПП. Это, в свою очередь, ограничивает способы комбинирования массивов изменений.

9.5. Собираемые метрики, используемые методы, стандарты и шаблоны

На этапе разработки ПП необходимо выполнять оценки расхождений плановых сроков и объемов с фактическими, числа проведенных обзоров, выявленных ошибок и дефектов, а также средних трудозатрат и производительности разработки.

Все полученные данные следует хранить в ИБД проектной группы.

Используемый инструмент: система подготовки документов (например, MS Word).

Используемые методы и стандарты: процесс организации; стандарты кодирования.

Используемые шаблоны: пользовательской документации; отчета по обзору; отчета о статусе проекта.

Контрольные вопросы

1. Каково назначение этапа разработки программного продукта?
2. В чем заключается процесс кодирования?
3. Что представляет собой тестирование «черного ящика»?
4. Какое тестирование называется тестированием «стеклянного ящика»?
5. Чем отличается структурное тестирование от функционального?
6. Перечислите критерии охвата и дайте им определения.
7. Чем отличается стратегия восходящего тестирования от стратегии целостного тестирования?
8. Какие недостатки присущи стратегии целостного тестирования?
9. Объясните принцип нисходящего тестирования.
10. Каким образом выполняется статическое тестирование?
11. Как организован процесс разработки справочной системы программного продукта и руководства пользователя?
12. Какова цель создания инсталляции программного продукта?
13. Какие проблемы могут возникнуть в процессе инсталляции и как с ними бороться?
14. Для чего необходимо управление версиями программного продукта?
15. Кто занимается управлением версиями программного продукта?

16. Что называется версией, вариантом версии, выходной версией программного продукта?
17. Какие способы идентификации версий программного продукта вы знаете?
18. Каким образом осуществляется идентификация версий программного продукта:
    а) основанная на нумерации;
    б) основанная на значениях атрибутов;
    в) основанная на изменениях?
19. Какие метрики собирают на этапе разработки программного продукта?
20. Какие методы, стандарты и шаблоны используют на этапе разработки программного продукта?

ГЛАВА 10

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА

10.1. Общая характеристика тестирования и его цикл

Тестирование представляет собой деятельность по проверке программного кода и документации. Она должна заранее планироваться и систематически проводиться специально назначенным независимым тестировщиком. Работа тестировщика начинается до утверждения спецификаций требований. Он проверяет требования к ПП на полноту и возможность тестирования, определяет методы тестирования.

Одновременно с началом этапа планирования и создания спецификаций требований тестировщик разрабатывает стратегию тестирования. После утверждения спецификаций требований им разрабатывается и детализируется план тестирования. Тогда же тестировщик создает наборы тестов для проведения интеграционного и системного тестирований. Тестирование завершается созданием отчета о тестировании, в котором представляются все результаты его проведения.

Для каждого программного изделия должен существовать набор тестов, проверяющий его корректность. Существует несколько уровней тестирования, позволяющих полностью проверить

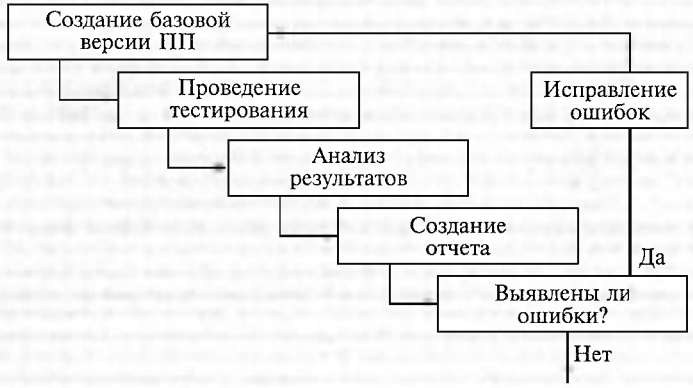


Рис. 10.1. Цикл тестирования

программное изделие. Каждый уровень имеет свои цели и компоненты. Можно выделить пять уровней тестирования: модульное; интеграционное; системное; выходное; приемочное.

Тестирование первых четырех уровней проводится внутри организации, тогда как приемочное тестирование выполняется совместно с представителями заказчика. Тестирование первого уровня осуществляет сам разработчик на этапе разработки, за остальные уровни отвечает независимый тестировщик.

Циклом тестирования называется совокупность действий, выполняемых тестировщиком с момента передачи базовой версии ПП тестировщику для интеграционного, системного или приемочного тестирования до момента успешного завершения тестирования (рис. 10.1). На каждом проходе цикла тестирования создаются:

базовая версия ПП, подлежащего тестированию;

отчет о ходе тестирования;

метрики тестирования (заносятся в базу данных проекта).

10.2. Виды тестирования

Модульное тестирование. Этот вид тестирования представляет собой процесс проверки отдельных программных процедур и подпрограмм, входящих в состав программ или программных систем. Модульное тестирование производится непосредственным разработчиком и позволяет проверять все внутренние структуры и потоки данных в каждом модуле. Этот вид тестирования является частью этапа разработки. При модульном тестировании выполняется набор тестов, определяемый разработчиком так, чтобы охват тестированием каждого модуля был не менее 70...75 %.

Элементами модульного тестирование являются:

синтаксическая проверка — проверка с использованием некоторого инструментального средства для выявления синтаксических ошибок в программном коде;

проверка соответствия стандартам кодирования — проверка кода на соответствие стандартам кодирования компании;

технический обзор программного кода.

После успешного завершения модульного тестирования все измененные модули и наборы тестов сохраняются в базе данных проекта.

Интеграционное тестирование. Этот вид тестирования проводится для проверки совместной работы отдельных модулей и предшествует тестированию всей системы как единого целого. В ходе интеграционного тестирования проверяются связи между модулями, их совместимость и функциональность. Оно осуществляется независимым тестировщиком и входит в состав этапа тестирования.

Элементами интеграционного тестирования являются:

проверка функциональности — проверка соответствия отдельных функций, выполняемых совокупностями модулей, функциям, заданным в спецификациях требований;

проверка промежуточных результатов — проверка всех промежуточных результатов и файлов на наличие и корректность;

проверка интеграции — проверка того, что модули передают друг другу информацию корректно.

Ошибки, выявленные в ходе интеграционного тестирования, заносятся в базу данных ошибок. Результаты интеграционного тестирования включаются в отчет о ходе тестирования при завершении цикла тестирования.

Системное тестирование. Этот вид тестирования предназначен для проверки программной системы в целом, ее организации и функционирования на соответствие спецификациям требований заказчика. Его проводит независимый тестировщик после успешного завершения интеграционного тестирования.

Элементами системного тестирования являются:

граничное тестирование — тестирование в граничных условиях;

прогоночное тестирование — тестирование всех функциональных характеристик реальной работы системы;

целевое тестирование — тестирование на целевой платформе (по возможности);

проверка документации — проверка пользовательской документации на корректность;

другие тесты, определяемые тестировщиком.

Ошибки, выявленные при системном тестировании, заносятся в базу данных проекта. Результаты системного тестирования включаются в отчет о ходе тестирования.

Выходное тестирование. Это завершающий этап тестирования, на котором проверяется готовность ПП к поставке заказчику. Данный вид тестирования проводит независимый тестировщик. Элементами выходного тестирования являются:

проверка инсталляции — проверка на ясность и корректность инструкций по инсталляции;

проверка документации — проверка того, что вся необходимая документация полностью подготовлена и готова к передаче заказчику.

Ошибки, выявленные при выходном тестировании, заносятся в базу данных проекта. При успешном завершении выходного тестирования ПП поставляется заказчику вместе с отчетом о результатах тестирования.

Приемочное тестирование. Этот вид тестирования проводится организацией, отвечающей за инсталляцию, сопровождение программной системы и обучение конечного пользователя.

10.3. Программные ошибки

Одними из распространенных определений программной ошибки являются следующие два:

программная ошибка — это расхождение между программой и ее спецификацией, причем тогда и только тогда, когда спецификация существует и она правильна;

программная ошибка — это ситуация, когда программа не делает того, чего пользователь от нее вполне обоснованно ожидает.

Все программные ошибки можно разделить на соответствующие категории. Рассмотрим те из них, которые встречаются наиболее часто.

Функциональные недостатки. Данные недостатки присущи программе, если она не делает того, что должна, выполняет одну из своих функций плохо или не полностью. Функции программы должны быть подробно описаны в ее спецификации, и именно на основе утвержденной спецификации тестировщик строит свою работу.

Недостатки пользовательского интерфейса. Оценить удобство и правильность работы пользовательского интерфейса можно только в процессе работы с ним. Желательно, чтобы в этой работе принимал участие сам пользователь. Этого можно добиться с помощью разработки прототипа ПП, на котором проводятся обкатка и согласование всех требований к пользовательскому интерфейсу с дальнейшей фиксацией их в спецификации требований. После утверждения спецификации требований любые отклонения от нее или невыполнение последних являются ошибкой. Это в полной мере касается и пользовательского интерфейса.

Недостаточная производительность. При разработке некоторого ПП очень важной его характеристикой может оказаться скорость работы, иногда этот критерий задается в требованиях заказчика к ПП. Плохо, если у пользователя создается впечатление, что программа работает медленно, особенно если конкурирующие программы работают ощутимо быстрее, но еще хуже, если программа не удовлетворяет заданным в спецификации требований характеристикам. Это уже ошибка, которая должна быть устранена.

Некорректная обработка ошибок. Процедуры обработки ошибок — очень важная часть программы. Правильно определив ошибку, программа должна выдать о ней сообщение. Отсутствие такого сообщения является ошибкой в работе программы.

Некорректная обработка граничных условий. Существует много различных граничных ситуаций. Любой аспект работы программы, к которому применимы понятия «больше» или «меньше», «раньше» или «позже», «первый» или «последний», «короче» или «длиннее», обязательно должен быть проверен на границах диапазона. Внутри диапазонов программа может работать прекрасно, а вот на

их границах могут происходить самые неожиданные ситуации, которые, в свою очередь, приводят к ошибкам в работе ПП.

Ошибки вычислений. К ошибкам вычислений относятся ошибки, вызванные неправильным выбором алгоритма вычислений, неправильными формулами, формулами, неприменимыми к обрабатываемым данным. Самыми распространенными среди ошибок вычислений являются ошибки округления.

Ошибки управления потоком. По логике работы программы вслед за первым действием должно быть выполнено второе. Если вместо этого выполняется третье или четвертое действие, значит, в управлении потоком допущена ошибка.

Ситуация гонок. Предположим, в системе ожидаются два события: А и Б. Если первым наступит событие А, то выполнение программы продолжится, а если событие Б, то в работе программы произойдет сбой. Разработчики предполагают, что первым всегда должно быть событие А, и не ожидают, что Б может выиграть гонки и наступить раньше. Такова классическая ситуация гонок.

Тестировать ситуации гонок довольно сложно. Наиболее типичны они для систем, где параллельно выполняются взаимодействующие процессы и потоки, а также для многопользовательских систем реального времени. Ошибки в таких системах трудно воспроизвести, и на их выявление обычно требуется очень много времени.

Перегрузки. Сбои в работе программы могут происходить из-за нехватки памяти или отсутствия других необходимых системных ресурсов. У каждой программы свои пределы, программа может не справляться с повышенными нагрузками, например со слишком большими объемами данных. Вопрос в том, соответствуют ли реальные возможности программы и ее требования к ресурсам спецификации программы и как она себя поведет при перегрузках.

Некорректная работа с аппаратурой компьютера. Программы могут посылать аппаратным устройствам неверные данные, игнорировать их сообщения об ошибках, пытаться использовать устройства, которые заняты или вообще отсутствуют. Даже если нужное устройство просто сломано, программа должна понять это, а не «зависать» при попытке к нему обратиться.

10.4. Тестирование документации

Читая и анализируя документацию, тестировщик прежде всего уделяет внимание ее точности, полноте, ясности, простоте использования и тому, насколько она соответствует ПП. В ходе тестирования документации наверняка будут найдены проблемы по каждому из указанных критериев. Поэтому заранее следует запланировать многократное тестирование печатного руководства, интерактивной справки и других документов.

Тестировщик, работающий с документацией, отвечает за техническую точность каждого ее слова. Он обязан произвести самую тщательную проверку ее соответствия требованиям спецификации и поведению программы. Особо следует обращать внимание на сложные и запутанные места текста. Они могут отражать неудачно спроектированные элементы самой программы. Технический писатель обязан описать продукт таким, каким он является на самом деле, поэтому помочь устранить запутанные места может только изменение проекта. Настаивать на таких изменениях важно еще и потому, что, в конечном счете, они обеспечат не только простоту документирования продукта, но и легкость его использования.

Необходимо проверить, не пропущены ли в документации какие-нибудь функции продукта. Технические писатели опираются на спецификацию, собственные заметки и беседы с разработчиками. Разработчики стараются держать их в курсе дела, но иногда забывают сообщить о новых функциях, только что внесенных в программу. Поскольку тестировщики сталкиваются с этими функциями гораздо раньше технических писателей, стоит позаботиться, чтобы их описания попали в документацию. Кроме того, если определенная функция описана в руководстве, то это не значит, что она будет описана и в интерактивной справке. Вполне вероятно, что информация легко может потеряться.

Следует помнить, что тестировщик одинаково не имеет права требовать изменений как в руководстве к ПП, так и в самом ПП. Обязанность тестировщика — выявить проблему, а что с ней делать, решать не ему. В частности, у тестировщика нет никакого права требовать стилистических изменений текста. Он может предложить такие изменения, но технический писатель вправе оставить все как есть и не обязан доказывать тестировщику, что поступает правильно. Для взаимодействия с техническими писателями формальная система отслеживания проблем обычно не применяется. Большинство комментариев вносится прямо в копию руководства.

По договоренности с техническим писателем выбирается способ выделения в тексте правок и комментариев. Копии комментариев следует сохранять и проверять по ним очередные версии документации.

10.5. Разработка и выполнение тестов

10.5.1. Требования к хорошему тесту

Хороший тест должен удовлетворять следующим требованиям:
должна быть достаточной вероятность выявления тестом ошибки.
Целью тестирования является поиск возможных ошибок. Поэто-

му, разрабатывая тестовые примеры, необходимо проанализиро вать все возможные варианты сбоев программы или ее некоррект ной работы;

набор тестов не должен быть избыточным. Если два теста пред назначены для выявления одной и той же ошибки, то достаточно выполнить только один из них;

тест должен быть наилучшим в своей категории. В группе похо жих тестов одни могут быть эффективнее других. Поэтому, выби рая тест, нужно взять тот, который с наибольшей вероятностью выявит ошибку;

тест не должен быть слишком простым или слишком сложным. Огромный и сложный тест трудно понять, трудно выполнить и долго создавать. Поэтому лучше всего придерживаться золотой середины, разрабатывая простые, но все же не совсем элементар ные тестовые примеры.

10.5.2. Классы эквивалентности и граничные условия

Одними из ключевых в теории тестирования являются понятия «классы эквивалентности» и «граничные условия».

Классы эквивалентности. Класс эквивалентности — это набор тестов, от выполнения которых ожидается один и тот же резуль тат. В простейшем случае тест представляет собой набор входных данных, вводимых в тестируемую программу. В случае эквивалент ных тестов эти данные обладают общими свойствами.

Группа тестов представляет собой класс эквивалентности, если выполняются следующие условия:

все тесты предназначены для выявления одной и той же ошибки;

если один из тестов выявит ошибку, то остальные тоже это сделают;

если один из тестов не выявит ошибку, то остальные тоже это не сделают.

Кроме перечисленных абстрактных условий существуют еще и практические критерии, позволяющие отнести к одному классу конкретную группу тестов. В качестве таких критериев можно рас сматривать соблюдения следующих условий:

тесты включают в себя значения одних и тех же входных данных;

для проведения тестов выполняются одни и те же операции программы;

в результате всех тестов формируются одни и те же значения выходных данных;

либо ни один из тестов не вызывает блок обработки ошибок программы, либо этот блок вызывается всеми тестами группы.

Поиск классов эквивалентности — процесс субъективный. Два человека, анализирующие одну и ту же программу, составят раз личные перечни классов. Необходимо все же стремиться выявить

как можно больше классов эквивалентности. Это сэкономит вре мя в дальнейшем и сделает тестирование более эффективным, избавив тестировщика от ненужного повторения эквивалентных тестов. Разбив все предполагаемые тесты на классы, можно затем выделить в каждом из них один или несколько тестов, которые кажутся наиболее эффективными; остальные выполнять ни к чему.

Вот несколько рекомендаций для поиска классов эквивалент ности.

Не забывайте о классах, охватывающих заведомо неверные или недопустимые входные данные. Часто такие входные данные вызывают в программе самые разнообразные ошибки. Поэтому, чем больше вы выделите типов неверного ввода, тем больше найдете ошибок.

Организуйте формируемый перечень классов в виде таблицы. Обычно классов эквивалентности оказывается очень много, поэтому нужен удобный и продуманный способ организации собранной информации. Обычно всю информацию сводят в большую табли цу, примером которой служит табл. 10.1.

Обратите внимание, что в перечень включены тесты не только допустимых, но и недопустимых или нестандартных входных дан ных. Информация в форме таблицы более понятна, ее легче и быстрее воспринимать, более очевидно разделение на допусти-

Таблица 10.1

Перечень классов эквивалентности

Входное или выходное событие	Допустимые классы эквивалентности	Недопустимые классы эквивалентности
Ввод числа	Числа от 1 до 99	Число 0. Числа больше 99. Выражение, результатом которого является недопустимое число (например, 5 – 5, результат которого равен 0). Отрицательные числа. Буквы и другие нечисловые символы
Ввод первой буквы имени	Первый символ является заглавной буквой. Первый символ является прописной буквой	Первый символ не является буквой
Рисование прямой	От одной точки прямая линия длиной до 4 см	Отсутствие рисунка. Линия длиннее 4 см. Линия не является прямой

мые и недопустимые варианты. Информацию в табличном виде легче анализировать, чтобы выяснить, все ли недопустимые варианты входных данных охвачены перечисленными классами эквивалентности.

Определите диапазоны числовых значений. С каждым новым диапазоном значений появляются и несколько новых связанных с ним классов эквивалентности. Обычно среди них имеются три недопустимых класса: все числа, которые меньше нижнего граничного значения диапазона; все числа, которые больше его верхнего граничного значения; нечисловые данные. Иногда один из этих классов отсутствует. Например, возможен ввод любого числа. В этом случае необходимо убедится, что это на самом деле так. Следует попробовать ввести очень большое число и посмотреть, что из этого получится.

Необходимо проверить также, нет ли у значений исследуемого параметра поддиапазонов. Каждый поддиапазон будет отдельным классом эквивалентности. Недопустимые классы будут располагаться ниже самого нижнего диапазона и выше верхнего из них.

Если для полей или параметров существуют фиксированные перечни значений, выясните, какие из значений входят в перечень. Если для параметра допускается только определенный перечень значений, один из классов эквивалентности может включать все значения из этого перечня, а другой — все остальные значения. В дальнейшем эти два класса можно разделить на ряд меньших классов.

Проанализируйте возможные результаты выбора из списков и меню. Любой элемент предложенного программой списка опций может представлять собой отдельный класс эквивалентности. Каждый элемент меню или списка опций обрабатывается программой особым образом, поэтому все они подлежат проверке. К классу недопустимых значений относятся ответы пользователя, которых нет в списке (если программа позволяет не только выбирать, но и вводить значения опций).

Например, если программа задает вопрос «Вы уверены? (Д/Н)», то один класс эквивалентности должен содержать ответ «Д» (также надо проверять и «д»), а второй — ответ «Н» («н»). Все остальные ответы являются недопустимыми (хотя вполне возможно, что программа интерпретирует все, что не является положительным ответом, как отрицательный, т.е. как эквивалент ответа «Н»).

Поищите классы значений, зависящих от времени. Предположим, что вы нажимаете на клавишу пробела непосредственно перед тем, как система загрузит программу, во время загрузки и сразу после нее. Как ни странно, но подобные тесты могут разрушить систему. Какие классы эквивалентности можно выделить в подобной ситуации? К первому из них относятся все события, происходящие задолго до выполнения задания, ко второму — события, происходящие в короткий отрезок времени непосредственно перед выполнением задания, к третьему — события в период его выполнения и т.д.

Выявите группы переменных, совместно участвующих в определенных вычислениях, результат которых ограничивается конкретным набором или диапазоном значений. Введите величины трех углов треугольника. К классу допустимых относятся значения, в сумме дающие 180°. Недопустимые значения можно разделить на два класса эквивалентности — с суммарным значением менее 180° и более 180°.

Посмотрите, на какие действия программа отвечает эквивалентными событиями. Выше рассматривались только входные события, поскольку анализировать их гораздо легче. Как пример выходного события рассмотрим третье событие в табл. 10.1. Программа обеспечивает вычерчивание линии длиной до 4 см, причем предполагается, что линия прямая, хотя она может оказаться и другой формы.

Трудность состоит в том, чтобы определить, какие входные данные управляют длиной и формой линии. Иногда различные классы входных данных на выходе дают один и тот же эффект. Если точный путь обработки каждого класса входных данных неизвестен, то лучше интерпретировать их как разные классы и тестировать по отдельности. Особенно это важно в тех случаях, когда в ответ на определенные входные данные при формировании выходной информации генерируется ошибка и управление передается блоку ее обработки.

Продумайте варианты операционного окружения. Бывает, что программа хорошо работает только при определенных типах мониторов, принтеров, модемов, дисковых устройств или любого другого подключенного к системе оборудования. Работа программы может зависеть даже от тактовой частоты компьютера. Поэтому, анализируя программу, особенно выполняющую низкоуровневые операции с оборудованием или ориентирующуюся на его определенные возможности, очень важно определить классы эквивалентных конфигураций системы.

Границы классов эквивалентности. Для каждого класса эквивалентности достаточно провести один-два теста. Лучшими из них будут те, которые проверяют значения, лежащие на границах класса. Эти значения могут быть наибольшими, наименьшими или какими-то другими, но в любом случае они должны быть предельными значениями параметров класса. Неправильные операторы сравнения (например > вместо ≥) вызывают ошибки только при граничных значениях аргументов. В то же время программа, которая сбоит при промежуточных значениях диапазона, почти наверняка будет сбоить и при его граничных значениях.

Необходимо протестировать каждую границу класса эквивалентности, причем с обеих сторон. Программа, которая пройдет эти

тесты, скорее всего, пройдет и все остальные, относящиеся к данному классу. Вот ряд примеров.

Если допустимы значения от 1 до 99, для тестирования допустимых данных можно выбрать 1 и 99, а для тестирования недопустимых — 0 и 100.

Если программа ожидает заглавную английскую букву, введите A и Z. Проверьте также символ @, поскольку его код предшествует коду символа A, и символ], код которого следует за кодом символа Z. Кроме того, проверьте символы a и z.

Если программа обеспечивает вычерчивание от одной точки линии длиной до 4 см, нарисуйте одну точку и линию длиной ровно 4 см. Пусть программа также попробует обеспечить вычерчивание линии нулевой длины.

Если сумма входных значений должна равняться 180, попробуйте ввести значения, дающие в сумме 179, 180 и 181.

Если программа получает определенное число входных данных, попробуйте ввести точно необходимое число, а также на единицу меньше и на единицу больше.

Если программа принимает ответы B, C и D, попробуйте ввести A и E.

Попробуйте отправить на печать файл непосредственно перед тем, как принтер напечатает еще чье-либо задание, и сразу после того.

10.5.3. Тестирование переходов между состояниями

В каждой интерактивной программе осуществляются переходы из одного очевидного состояния в другое. Простейшим примером может служить меню. После запуска программы в нем имеется один перечень команд. После выбора одной из них состояние программы меняется и в меню появляются команды, доступные в этом новом состоянии.

Необходимо протестировать каждую предлагаемую программой опцию, каждую команду меню. Команда 10 может быть доступна в режиме, открываемом по команде 9 или по команде 22. В этом случае команду 10 придется протестировать дважды — в обоих режимах. Однако команд меню, всевозможных режимов программы и путей перехода в эти режимы может быть так много, что протестировать их все просто нереально. Поэтому, отбирая тесты для проверки путей выполнения программы, лучше всего руководствоваться следующими принципами:

тестировать все наиболее вероятные последовательности действий пользователей;

если можно предположить, что действия пользователя в одном режиме могут влиять на представление данных или набор предоставляемых программой возможностей в другом режиме, тестировать эти действия;

кроме проведения самых необходимых тестов из тех, что описаны выше, поработать с программой в произвольном режиме, случайным образом выбирая путь ее выполнения.

Переходы между состояниями могут быть гораздо более сложными, чем просто выбор команд меню. Содержимое и структура очередной формы ввода данных могут зависеть от информации, введенной в предыдущей форме, значения одних полей могут определять допустимые значения других, ввод определенной информации может инициировать серию дополнительных запросов. Например, при вводе чисел от 1 до 99 программа выводит одну форму запроса, обращенного к пользователю, а при вводе любых других чисел — другую. В этом случае вместе с классами эквивалентности и их граничными значениями следует проанализировать и возможные пути выполнения программы, чтобы составить действительно полноценный набор тестов.

Очень полезно составление схем меню. В подобной схеме отражаются все состояния программы и команды, вызывающие переходы между этими состояниями. В нее включаются команды, активизируемые через меню, графические средства (например, различные кнопки), и команды, выполняемые после нажатия определенных клавиш. Например, в схеме может быть показан путь от меню «Файл» к команде «Открыть», затем к диалоговому окну «Открытие файла» и назад, к основному состоянию программы. Особенно удобны подобные схемы в случае, если определенное диалоговое окно можно открыть несколькими способами и выйти из него в несколько различных режимов. В этом случае можно нарисовать на схеме все направления переходов и по ним протестировать программу. Это более надежный способ, чем работа с программой без всякого плана с риском пропустить важные взаимосвязи ее состояний.

10.5.4. Условия гонок и другие временные зависимости

Попробуйте вмешаться в работу программы, когда она выполняет переход между двумя состояниями. Понажимайте на клавиши, особенно командные. Попробуйте понажимать на клавиши или повыбирать какие-либо пункты меню, когда программа выполняет операции обработки данных или ввода—вывода, предложите программе ввести или вывести параллельно еще какую-нибудь информацию. Например, во время печати файла попросите ее распечатать еще один.

Если в программе определены ситуации тайм-аута, когда она ждет определенного события в течение заданного времени, а затем переходит в другое состояние, проверьте ее реакцию на действия пользователя, запросы системы или наступление ожидаемого события на границах интервала тайм-аута. Посмотрите, что

будет, если событие произойдет за секунду до того, как програм-
ма должна прекратить ожидать его, или через секунду после этого.

Протестируйте систему при повышенной нагрузке. В мультиза-
дачной среде запустите несколько других программ и посмотри-
те, как поведет себя ваша, успешно ли она справится со своей
работой. Отправьте большой файл на принтер, чтобы процессор
все время переключался на обслуживание печати. Подключите
различные внешние устройства и заставьте их генерировать пре-
рывания так часто, как только удастся. Короче говоря, замедлите
и нагрузите компьютер, насколько это возможно. В результате ваша
программа будет выполняться медленнее, и, быстро вводя дан-
ные, можно попробовать превысить ее возможности приема. Если
в нормальном режиме работы сбоя программы добиться не удает-
ся, это может получиться при повышенной нагрузке.

Выполняя «стандартное» тестирование программы при сильно
повышенной нагрузке, можно столкнуться с совершенно неожи-
данными ситуациями гонок. Если окажется, что программа в этом
отношении уязвима, то необходимо провести в таких условиях
полный цикл тестирования. Главная задача — обеспечить такую
надежность разрабатываемого программного обеспечения, чтобы
оно работало, пусть медленно, но без сбоев в любой системе и
при любых дополнительных нагрузках. По крайней мере, необхо-
димо совершенно точно выяснить, какие конфигурации системы
являются предельными для эксплуатации программы.

10.5.5. Нагрузочные испытания

Важно не забыть протестировать те ограничения возможностей
ПП, которые определены в его документации. Откройте максим-
мальное число файлов или других структур данных, с которыми
программа может работать, попробуйте подольше поэксплуати-
ровать ее в таком состоянии. Если в документации ограничения не
описаны, но существуют логически допустимые значения каких-
либо параметров, то следует проверить и их. Если программа не
справится с достаточно большим числовым значением, которое
пользователь вполне может ввести, то составляется отчет об ошибке.
Если же программа спокойно принимает и обрабатывает как очень
маленькие, так и очень большие значения параметров, возмож-
но, ограничений на них нет.

Необходимо проверить, как ведет себя программа, когда ис-
черпываются различные аппаратные ресурсы, например перепол-
няется диск или в принтере заканчивается бумага. Выясните, что
будет, когда в системе останется очень мало свободной памяти,
нагрузите компьютер как следует и посмотрите, что получится.

Нагрузочное тестирование — это, по сути дела, один из видов
тестирования граничных условий. Схема его проведения абсолют-

но аналогична. Сначала программу запускают в условиях, в кото-
рых она должна работать, а затем в условиях, для которых она не
предназначена. Имеет смысл проверить и различные комбинации
условий. Вполне возможно, что, справившись с различными по-
вышенными нагрузками по отдельности, программа не выдержит
их все вместе. Нагрузив систему, проведите не просто один-два
теста, а достаточно длительное и обстоятельное тестирование.
Поэксплуатируйте программу в таких условиях некоторое время,
возможно, сбой не сразу, но все же произойдет.

10.5.6. Прогнозирование ошибок

Иногда тестировщик предполагает, что определенный класс
тестов вызовет сбой программы, хотя и не может это логически
обосновать. Доверяйте своей интуиции и обязательно включайте
подобные тесты в общий план. Существует целый ряд ситуаций и
значений, которые, хотя и не являются граничными, но часто
вызывают программные сбои. Типичным примером таких значе-
ний является 0. Не стоит тратить время на поиски обоснований
того, почему определенное входное значение или место програм-
мы кажется вам подозрительным. Просто протестируйте его.

Случается, что в сложных ситуациях интуиция подсказывает
гораздо лучшую тактику тестирования, чем тривиальная логика.
Бывает, что срабатывает и ассоциативная связь: вы уже находили
ошибку в подобных обстоятельствах, хотя можете этого даже не
помнить. Как бы там ни было, доверяйте своему внутреннему чув-
ству и учитесь к нему прислушиваться: с опытом оно будет стано-
виться все более развитым и надежным.

10.5.7. Тестирование функциональной эквивалентности

При тестировании функциональной эквивалентности сравни-
ваются результаты вычислений разными программами одной и
той же математической функции. Термин «функциональная эквив-
валентность» не имеет ничего общего с термином «классы эквив-
валентности». Если обе программы при вычислении одной и той
же функции дают одинаковые результаты, значит, в них приме-
нены эквивалентные методы вычислений.

Предположим, что тестируется программа, которая вычисляет
математическую функцию и печатает результат. Это может быть
простая тригонометрическая функция или гораздо более сложная
функция, инвертирующая матрицу или возвращающая коэффи-
циенты для построения кривой, отражающей некоторый набор
данных. Обычно в таких случаях можно найти другую программу,
выполняющую те же действия, и при этом достаточно надежную
и проверенную временем. Обеим программам предлагается обра-

ботать одинаковые наборы входных данных. Если результаты совпадут, значит, тестируемая программа работает правильно.

Автоматизация тестирования функциональной эквивалентности. Везде, где возможно применить метод тестирования функциональной эквивалентности, он будет наилучшим выбором. Прежде всего, не придется вычислять значения вручную. Если функция сложна, это поможет сэкономить огромное количество времени и избежать ошибок, так часто возникающих при неавтоматизированных расчетах.

Процесс сравнения также, скорее всего, удастся автоматизировать. Простейшим способом может быть вывод результатов расчетов в файлы и их последующее сравнение с помощью соответствующей программы. Компьютер выполнит сравнение файлов и быстрее, и аккуратнее. Можно определить и допустимые расхождения результатов, например погрешности округлений.

Кроме того, вполне возможно автоматизировать и весь процесс тестирования, от ввода выходных данных до сравнения выходных. Если это получится, процедура тестирования будет выполняться практически мгновенно и исключительно надежно.

Хотя автоматизация подобных тестов — процесс не особенно сложный, он требует некоторого времени. Если программа может считать входные данные из файла, его необходимо подготовить. Кроме того, придется написать небольшие программки, выполняющие сравнение результатов.

Тестирование функциональной эквивалентности может потребовать некоторых затрат. Надежная эталонная программа, которая будет использоваться для сверки результатов, может оказаться не такой уж дешевой. К тому же, скорее всего, придется написать кое-какие программки. Может потребоваться и дополнительная техника, например второй компьютер. Разумеется, нельзя определить универсальные критерии того, сколько денежных средств имеет смысл потратить на проведение подобного тестирования. Однако есть определенные мерки.

Прежде всего оцените, сколько дней потребуется на тестирование программы вручную. Включите в расчет время планирования, выполнения вычислений и проведения тестов. Не забудьте и о том, что каждый тест придется провести не один раз, поскольку вы будете выявлять ошибки и повторять всю процедуру тестирования с самого начала. Прикиньте, сколько циклов тестирования потребуется провести.

Оцените также, сколько времени сэкономят автоматизированные средства тестирования. Снова учтите весь процесс: планирование, программирование и отладку тестов. Постарайтесь оценить необходимое время как можно более реалистично.

Число дней, которые предполагается сэкономить, умножьте на свой двойной оклад. Сумма оклада умножается на два потому, что в

расчет берется еще и выгода, которую компания получает от ускорения процесса тестирования и повышения его надежности. Если ваша оценка верна, полученная сумма — это минимум того, что можно сэкономить при функциональном тестировании с помощью эталонной программы. Если сама программа стоит меньше этой суммы, то любой разумный руководитель одобрит ее покупку.

Подготовьте предложение и презентацию, поясняющие назначение покупаемой эталонной программы и основы расчетов. Если компания не настолько заинтересована в сокращении времени разработки, чтобы вкладывать в нее дополнительные средства, поясните, что вложенные средства гарантированно окупятся благодаря надежности и качеству продукта.

Анализ чувствительности. Автоматизация функционального тестирования позволяет выполнить гораздо больше тестов, чем вручную. Однако отбираться они должны не менее тщательно, поскольку число возможных значений входных данных, скорее всего, по-прежнему будет слишком велико, чтобы можно было провести абсолютно полное тестирование. У большинства функций число возможных значений аргументов бесконечно, так что справиться с ними не под силу даже компьютеру. Обязательно нужно будет проверить граничные значения. Как же отобрать наилучшие тесты? Чаще всего для этого применяется анализ чувствительности. Эта процедура заключается в следующем.

Прежде всего, получают общее представление о поведении функции, вычислив ее значения для ряда параметров, располагающихся по всей области определения. Затем ищут участки области определения, на которых небольшие изменения аргументов вызывают значительные скачки результирующих значений. Именно такие участки наиболее чреваты ошибками.

Значения, полученные в результате тестирования программируемой функции и ее эталона, могут не вполне совпадать. Если в процессе расчетов выполняются операции с плавающей запятой, то неизбежны округления или усечения результатов, а значит, и небольшие расхождения. Обычно это не страшно. Необходимо только правильно оценить допустимые погрешности округлений, чтобы можно было зафиксировать превышающие их расхождения, вызванные иными причинами.

Рекомендуется равномерно разделить каждый диапазон тестируемых входных значений на ряд поддиапазонов (их может быть около 100) и протестировать по одному значению внутри каждого из них. После ввода каждого значения проверяйте, правильный ли получился результат, чтобы не тратить зря времени, если что-то не так.

Получив общую картину поведения функции, проанализируйте его на предмет резких перемен. Если значения функции на отдельных участках области ее определения резко возрастают или

уменьшаются либо наблюдаются разрывы и скачки, на них необходимо обратить более пристальное внимание.

Предположим, что при каком-то входном диапазоне значения функции (или их расхождение со значениями эталонной функции) резко возрастают. Разделите этот диапазон на 100 равных частей и проверьте по одному значению внутри каждой из них. Если все в порядке, то вы убедитесь, что значения тестируемой и эталонной функции для всех тестируемых аргументов совпадают, а если нет, то вы нашли ошибку.

При профессиональном тестировании математических функций не обойтись без некоторых знаний из теории вероятности. Если речь идет не об одной, а о целом ряде функций, необходимы боле эффективные и научно обоснованные технологии поиска критических участков области определения функции, т.е. тех, где ее значения резко меняются или отличаются от эталонных.

Случайный ввод. Вместо разделения всей тестируемой области определения функции на определенное число равных участков можно воспользоваться другим способом подбора входных значений — случайным. Случайный выбор значений более эффективен, поскольку гарантирует их полную равноправность. Например, тестируя такую последовательность входных значений, как 0,02; 0,04; 0,06 и т.д., вы никогда не узнаете, как программа обрабатывает нечетные числа (допустим, 0,03) или числа с большим количеством значащих цифр (например, 0,1415). В то же время при выборе входных значений случайным образом область определения функции покрывается гораздо более полно, все типы и диапазоны значений входных данных охватываются равномерно.

Если вы затрудняетесь в выборе методики подбора входных данных или не вполне уверены в поведении тестируемой функции, остановитесь на случайном способе. Он прекрасно подходит и для автоматизированного тестирования.

Отсутствие четкого обоснования для выбора конкретных входных значений можно компенсировать числом проводимых тестов. Здесь нет никаких ограничений — чем больше тестов проводится для каждого из классов эквивалентности, тем лучше. Обычно при автоматизированном тестировании со случайными входными значениями проводится, как минимум, 1000 вычислений.

Использование генератора случайных чисел. Случайный ввод вовсе не означает «все, что приходит в голову», иначе он будет слишком предвзятым, чтобы претендовать на равномерный охват области определения функции. Больше подойдет компьютерная программа, которая может генерировать случайные числа в неограниченном количестве. Однако следует иметь в виду, что алгоритмы, используемые многими подобными программами, вовсе не случайны. Кроме того, нередко в программах даже базовый алгоритм реализован не точно. Поэтому, прежде чем выбрать конкретный генератор случайных чисел, даже созданный вполне авторитетной компанией или встроенный в один из стандартных языков программирования, необходимо выяснить, какой алгоритм положен в основу его работы и подходит ли он для ваших нужд. То, что годится для простой программы, позволяющей получить на экране разноцветный салют, может совершенно не подойти для профессионального тестирования сложных инженерных программ. Нередко генераторы случайных чисел повторяют их последовательность через определенный интервал, пусть даже очень большой. В связи со всем этим прислушайтесь к следующим советам.

Не доверяйте какой-либо программе просто потому, что она у вас уже есть, а другую еще придется поискать. Попытайтесь разобраться в программе, и только когда она вам будет полностью ясна, используйте ее.

Если вы собираетесь воспользоваться генератором случайных чисел, встроенным в язык программирования, стоит его немного доработать. Сгенерировав с его помощью достаточно большое количество чисел (100...1000), перемешайте их: измените их порядок с помощью последующих чисел, выдаваемых этим же генератором. Хотя это и замедлит работу, зато результат в случае плохого исходного генератора может быть уже вполне приемлемым.

Применение технологии эквивалентности. Тестирование математических функций — не единственная область применения эталонных программ. Путем сравнения с готовым и проверенным ПП можно тестировать самые разные аспекты поведения программы. Вот несколько примеров.

Если тестируется программа проверки правописания, в основе которой лежит тот же алгоритм, что и в одной из уже существующих программ, можно предложить обеим программам проверить один и тот же набор слов.

Если тестируется программа автоматического переноса слов, особенно если отрабатывается модификация ее алгоритма для другого языка, можно взять для сверки проверенную программу, продаваемую на рынке программного обеспечения, подготовить узкий столбик текста и предложить его обеим программам.

При тестировании программы, выполняющей выравнивание текста по ширине строки, необходимо проверить, насколько равномерно она разделяет слова пробелами. Для образца можно взять обычный текстовый процессор и обеим программам предложить один и тот же текст, набранный одинаковыми шрифтами.

Для отладки посылаемых на принтер двойных управляющих последовательностей можно перенаправить вывод в файл и то же самое сделать в эталонной программе, создав в ней точно такой же файл. Затем оба файла надо сравнить — они должны быть идентичны.

Во всех случаях, когда необходимо протестировать выходные данные, которые легко направить в файл, также можно использовать эталонную программу, умеющую генерировать те же данные. Их легко можно сравнить.

В каждом конкретном случае существуют свои аргументы «за» и «против» этой технологии. Например, на ее реализацию может потребоваться слишком много времени, средств или усилий. Кроме того, эталонная программа тоже вполне может содержать ошибки. Но в любом случае методику тестирования эквивалентности следует иметь в виду — нередко ее применение значительно ускоряет работу и во много раз повышает ее эффективность.

Не забывайте включать в отчеты об ошибках выходные данные, полученные от обеих программ. Они очень важны для поиска причины ошибки.

10.5.8. Регрессионное тестирование

Общие сведения. Основной работой тестировщиков является регрессионное тестирование. У этого термина два значения, которые объединяет идея повторного использования разработанных тестов.

Представьте, что после проведения теста была обнаружена ошибка и программист ее исправил. Тестировщик снова проводит тот же тест, чтобы убедиться, что ошибки больше нет. Это и есть регрессионное тестирование. Можно провести несколько вариаций исходного теста, чтобы как следует проверить исправленный фрагмент программы. В данном случае задача регрессионного тестирования состоит в том, чтобы убедиться, что выявленная ошибка полностью исправлена программистом и больше не проявляется.

Второй пример применения регрессионного тестирования. После выявления и исправления ошибки проводится стандартная серия тестов, но уже с другой целью: убедиться, что, исправив одну часть программы, программист не испортил другую. В этом случае тестируется целиком вся программа, а не какой-то исправленный ее фрагмент.

Рекомендации по исправлению ошибок. Получив отчет об ошибке, программист должен тщательно проанализировать исходный код программы, найти причину ошибки, исправить ее и протестировать результат. Это идеальный вариант. На практике некоторые программисты устраняют только описанные в отчете симптомы наличия ошибки, а не саму ее. В результате какие-то проявления ошибки исчезают, но другие остаются. Бывает и так, что программист неправильно понимает отчет и исправляет не то, что надо. Некоторые недобросовестные сотрудники вообще не проверяют свою работу, не глядя, вносят исправления и немедленно возвращают программу тестировщикам. В результате старая ошибка остается и новые появляются. Опытный тестировщик должен быть готов к подобным ситуациям. Считается, что около трети вносимых в программу исправлений не срабатывают или даже «ломают» то, что уже работало.

Вот три задачи, которые должен поставить перед собой тестировщик, проверяющий внесенные программистом исправления:

убедиться, что ошибка исправлена, для чего выполнить тот же тест, в котором она проявилась и который был описан в отчете. Если программа его не пройдет, дальнейшее тестирование не имеет смысла;

постараться найти связанные ошибки. Предположим, что программист устранил описанные в отчете симптомы ошибки, но саму ее не исправил. Можно попробовать спровоцировать симптомы ошибки каким-нибудь иным способом. В программе могут оказаться другие подобные ситуации? Следует провести как можно больше тестов;

протестировать оставшуюся часть программы. Неожиданные последствия исправления могут проявиться где-нибудь еще. Их поиск проводится неформально — без заранее подготовленного плана. Просто необходимо подумать, какие части программы могут быть затронуты внесенными исправлениями, и проверить их.

Стандартная серия тестов. Некоторое время спустя после начала тестирования ПП формируется *библиотека регрессионных тестов*. Это полный набор тестов, охватывающий всю программу и выполняющийся каждый раз, когда программисты сдают ее очередную рабочую версию. Лучше всего, если тесты полностью автоматизированы. В этом случае все равно приходится тратить на тестирование некоторое время, но вся процедура получается гораздо менее трудоемкой.

Когда для тестирования предоставляется очередная версия программы и наступает время повторить все тесты библиотеки, возникает ряд вопросов. Насколько велика библиотека? Действительно ли необходимо выполнять все тесты библиотеки снова? Не всегда легко заранее определить, какие тесты следует включать в регрессионную библиотеку. Поэтому первоначально в ней может оказаться больше тестов, чем это действительно необходимо. Как минимум, в нее должны входить примеры для проверки граничных условий и временных характеристик. Но стоит ли выполнять их все каждый раз?

Выполнять регрессионные тесты обычно не хочется, поскольку вероятность выявления ими ошибок не особенно велика. При первом проведении тестирования некоторые из этих тестов могут выявить ошибки, но после их окончательного исправления выполнение тех же тестов снова и снова кажется пустой потерей времени. Если ошибки больше нет, то какова вероятность, что она появится снова? А как быть с тестами, которые выполнялись уже несколько раз и ни разу не выявили ошибок? В некоторых

компаниях такие тесты исключают из библиотеки, оставляя толь-
ко те, в которых проявлялись ошибки.

Вместо того чтобы мучительно обдумывать каждый тест, луч-
ше пойти более простым путем. Включите в регрессионную биб-
лиотеку все тесты, которые покажутся вам полезными. Периоди-
чески, примерно через каждые три цикла, пересматривайте эту
библиотеку и удаляйте те тесты, отсутствие которых не снизит
качества работы. Вот несколько полезных советов.

*Удалите тесты, которые эквивалентны другим тестам библи-
отеки.* В идеале такие тесты вообще не должны попадать в библи-
отеку, но когда она создается несколькими сотрудниками, по-
добные накладки вполне возможны.

*Уменьшите числа тестов, объектом которых является уже ис-
правленная ошибка.* Если ошибка или некоторые ее разновидности
проявляются в течение целого ряда циклов тестирования, то в
библиотеку стоит добавить достаточное число тестов для их выяв-
ления. Это вполне нормально и уместно. Соответствующую часть
программы необходимо тщательнейшим образом тестировать до
тех пор, пока в ней не останется и следа ошибки. Однако после
этого большую часть тестов, направленных на поиск исправлен-
ной ошибки, можно удалить из библиотеки.

Комбинируйте тесты. Если 15 тестов, которые программа скорее
всего пройдет, можно объединить в один — сделайте это. В начале
тестирования так поступать не стоит, но в дальнейшем объедине-
ние тестов позволит значительно ускорить работу.

По возможности автоматизируйте тестирование. Если вы уве-
рены, что определенная группа тестов будет выполняться в тече-
ние пяти или десяти последующих циклов тестирования, то стоит
потратить время на автоматизацию.

Выделите часть тестов для периодического выполнения. Все тес-
ты регрессионной библиотеки не обязательно выполнять после
каждого изменения программы. Это можно делать реже — на каж-
дом втором или на каждом третьем цикле. На последней стадии
тестирования лучше выполнить максимально возможное число
тестов, чтобы убедиться, что программа готова к выпуску, на дру-
гих циклах достаточно половины или даже трети всех тестов.

Регрессионная библиотека должна включать в себя все лучшие
тесты из тех, что уже разработаны, но, если она будет слишком
велика, не останется времени на разработку новых тестов. А ведь
именно новейшие тесты с наибольшей вероятностью выявляют
еще не найденные ошибки. Поэтому планировать работу надо так,
чтобы регрессионная библиотека служила средством повышения
эффективности тестирования, а не его тормозом.

Выполнение тестов. Придумать хороший тест — это только по-
ловина дела. Его еще нужно правильно выполнить. Вот несколько
рекомендаций.

Если вы хотите, чтобы при установке программы на компью-
тер у пользователя была возможность выбора конфигурации, не-
достаточно просто запустить программу установки и посмотреть,
предоставляет ли она необходимые опции. Выполните каждый
вариант установки, всякий раз запуская саму программу и прове-
ряя, действительно ли установлена выбранная конфигурация. Убе-
дитесь, что программа при этом полностью работоспособна.

Если программа позволяет указать размер печатаемой страни-
цы, отступы и другую подобную информацию, не считайте дело
сделанным, увидев, что документ правильно выглядит на экране.
Его необходимо распечатать.

Если в программе используются символы из расширенного на-
бора ASCII, недостаточно увидеть их на экране. Они должны пра-
вильно печататься, пересылаться через модем и т.д. Необходимо
учесть все программное обеспечение, через которое будут прохо-
дить выходные данные: драйверы устройств, алгоритмы импорта.

Основное правило, вытекающее из приведенных рекоменда-
ций, можно сформулировать так: тестовая процедура должна за-
ставить программу использовать введенные данные и подтвердить,
что они используются правильно.

10.6. Собираемые метрики, используемые методы, стандарты и шаблоны

На фазе тестирования необходимо выполнять оценки расхож-
дений плановых сроков и объемов с фактическими, числа прове-
денных обзоров, выявленных ошибок и дефектов, а также сред-
них трудозатрат и производительности тестирования. Кроме того,
должен производиться сбор метрик покрытия ПП тестированием
и соотношения открытых и закрытых дефектов. Все полученные
данные следует хранить в ИБД проектной группы.

Используемый инструмент: система подготовки документов
(например, MS Word).

Используемые методы и стандарты: процесс организации;
метрическая программа.

Используемые шаблоны: отчета о результатах тестирования;
отчета о ходе тестирования; отчета о поставке ПП; руководства
по поставляемому ПП; плана тестирования; описания процедур
тестирования; отчета по обзору; отчета о статусе проекта.

Контрольные вопросы

1. Каково назначение этапа тестирования в жизненном цикле разра-
ботки программного продукта?

2. Что подвергается тестированию в течение жизненного цикла разработки программного продукта?

3. Какие существуют уровни тестирования и кто ответственен за проведение тестирования на каждом из них?

4. Что такое цикл тестирования, какие основные действия он в себя включает?

5. Каковы назначение и основные элементы тестирования: а) модульного; б) интеграционного; в) системного; г) выходного?

6. Что является программной ошибкой?

7. Какие категории программных ошибок вы знаете?

8. Какова цель тестирования документации?

9. Какими характеристиками должен обладать хороший тест?

10. Дайте определение понятия «классы эквивалентности».

11. Какие условия должны выполняться, чтобы тесты можно было отнести к одному классу эквивалентности?

12. Какие критерии используют для определения класса эквивалентности?

13. Дайте определение понятия «границы классов эквивалентности».

14. Что вы понимаете под тестированием переходов между состояниями программного продукта?

15. Какие критерии существуют для отбора тестов по проверке путей выполнения программы?

16. Какие условия можно создать программному продукту для выявления ситуации гонок и других временных зависимостей?

17. Для чего проводится нагрузочное тестирование программного продукта?

18. Что вы думаете о прогнозировании ошибок?

19. Дайте определение понятия «функциональная эквивалентность».

20. Какие преимущества дает автоматизация тестирования?

21. Как выполняется анализ чувствительности?

22. Каковы преимущества случайного ввода входных значений?

23. Каким образом можно организовать случайный ввод?

24. Каковы цели и назначение регрессионного тестирования?

25. Как можно определить, что ошибка успешно исправлена?

26. Куда заносятся результаты тестирования и как отслеживается исправление ошибок?

27. Какие метрики собирают на этапе тестирования?

28. Какие методы, стандарты и шаблоны используют на этапе тестирования?

ГЛАВА 11

СОПРОВОЖДЕНИЕ ПРОГРАММНОГО ПРОДУКТА

11.1. Роль этапа сопровождения в жизненном цикле программного продукта

Сопровождение ПП — это процесс адаптации поставляемого ПП к новым условиям, внесения изменений в ПП и соответствующую документацию, вызванных возникшими проблемами или потребностями в модификации, при сохранении неизменными его основных функций. Сопровождение ПП выполняется сопровождающей организацией или службой сопровождения организации, разработавшей ПП.

Согласно стандарту IEEE-90 под сопровождением понимается внесение изменений в ПП в целях исправления обнаруженных ошибок, повышения производительности или адаптации к изменившимся условиям работы или требованиям.

Элементами сопровождения являются:

перепроектирование и переработка небольших частей поставляемого ПП;

перепроектирование и переработка интерфейсных программных модулей;

модификация кода, документации или структуры базы данных ПП.

В задачи сопровождения ПП входят обновление, приводящее к изменению функционального назначения ПП, и исправление, не затрагивающее функционального назначения ПП. Обновление ПП осуществляют путем подачи заявки на изменение ПП. Исправление ПП включает в себя корректирующее сопровождение (обработка, локализация или исправление ошибок в программах), адаптивное сопровождение (приспособление к новому окружению) и совершенствующее сопровождение (улучшение характеристик или эксплуатационной надежности).

Процесс сопровождения включает в себя следующие основные действия.

1. Подготовительная работа, предусматривающая:

планирование действий и работ, выполняемых в процессе сопровождения;

определение процедур локализации и разрешения проблем, возникающих в процессе сопровождения.

2. Анализ проблем и запросов на модификацию ПП, предполагающий:

анализ сообщения о возникшей проблеме или запроса на модификацию ПП, в ходе которого изучаются возможность выполнения модификации, ее тип (корректирующая, улучшающая, профилактическая или адаптирующая к новой среде), масштаб (размеры модификации, стоимость и время ее реализации); критичность (воздействие на надежность, производительность и безопасность);

оценку целесообразности проведения работ и вариантов ее проведения;

утверждение выбранного варианта модификации.

3. Модификация ПП, предусматривающая определение компонентов ПП, их версий и документации, подлежащих модификации, и внесение необходимых изменений.

4. Проверка и приемка, в ходе которых проверяется целостность модифицированного ПП и утверждаются внесенные изменения.

5. Перенос (*конвертирование*) *ПП в новую среду работы.*

6. Снятие ПП с эксплуатации.

На этапе сопровождения составляют следующие отчеты по обзорам новых версий ПП; по метрикам сопровождения; о завершении жизненного цикла ПП (смерти ПП).

11.2. Собираемые метрики, используемые инструменты и шаблон

На этапе сопровождения необходимо определять число дефектов, выявленных после поставки ПП, а также оценивать трудоемкость внесения изменений. Все полученные данные следует хранить в ИБД проектной группы.

На данном этапе используются все инструментальные средства, перечисленные в описаниях предыдущих этапов жизненного цикла ПП (см. подразд. 6.6, 7.6, 8.5, 9.5 и 10.6), и шаблон запроса на изменение.

Контрольные вопросы

1. Каково назначение этапа сопровождения в жизненном цикле программного продукта?
2. Какие элементы включает в себя этап сопровождения?
3. Какие основные задачи выполняются на этапе сопровождения?
4. Какие типы исправлений программного продукта вы знаете?
5. Какие основные действия выполняются на этапе сопровождения?
6. Какие документы составляют на этапе сопровождения?
7. Какие метрики собирают на этапе сопровождения?

ГЛАВА 12

УПРАВЛЕНИЕ ПОСТАВКАМИ ПРОГРАММНЫХ ПРОДУКТОВ

12.1. Общие сведения об управлении поставками

Для управления поставками ПП в каждой организации, занимающейся разработкой ПП, должны быть определены:

процедура поставки ПП;

схемы классификации поставляемых ПП;

методы проверки и отслеживания соответствия ПП руководству по поставке ПП.

Процедура поставки ПП разрабатывается и утверждается внутри каждой организации, занимающейся разработкой ПП, но в зависимости от требований заказчика она может быть модифицирована для какого-либо конкретного проекта. Процедура поставки программного продукта включает в себя перечень основных действий и задач, которые должны быть выполнены в процессе поставки, с указанием их очередности.

Схемы классификации поставляемых ПП содержат перечень необходимых поставок и описание их содержимого, а также описание используемых принципов нумерации версий ПП.

Методы проверки и отслеживания соответствия ПП руководству по поставке ПП направлен на обеспечение выполнения процедуры поставки, соблюдения схемы классификации поставляемого ПП и соответствия поставки руководству по поставке ПП.

12.2. Классификация поставляемых программных продуктов

Готовый к поставке ПП должен быть классифицирован как один из следующих типов:

ES-поставка — поставка прототипа;

PA-поставка — поставка Альфа-версии;

PB-поставка — поставка Бета-версии;

RP-поставка — окончательная поставка.

Прототип является начальной версией ПП, которая используется для демонстрации концепций, заложенных в системе, про-

верки вариантов требований, а также поиска проблем, которые могут возникнуть как в ходе разработки, так и при эксплуатации ПП, и возможных вариантов их решения.

Очень важна быстрая разработка прототипа ПП, чтобы пользователи могли начать экспериментировать с ним как можно раньше.

В отличие от прототипа Альфа-версия ПП реализует все или практически все основные функции ПП, однако некоторые из них могут еще отсутствовать или выполняются с ошибками, возможно крайне нестабильно. Индивидуальность ПП уже полностью сформирована, видны его основные особенности и возможности.

Спецификация, конструкторская документация, справочная система и документация пользователя практически готовы. Альфа-версию обычно отдают только заказчикам.

В Бета-версии реализован полный набор запланированных функций ПП. Фатальных ошибок в ПП нет, серьезных ошибок очень мало. Готовы все проектные документы и ПП соответствует требованиям заказчика. Справочная система и документация пользователя также полностью готовы. Продукт можно отдать на тестирование сторонним пользователям или распространять в рекламных целях.

Окончательная поставка подразумевает завершение работы над проектом по разработке ПП и переход к этапу сопровождения. Продукт со всей сопутствующей документацией передается заказчику.

12.3. Действия, выполняемые при поставке программного продукта

Продукция, готовая к поставке, должна быть представлена на обзор, проводимый при участии высшего руководства и группы процесса.

Для проведения такого обзора необходимо:

присвоить поставляемой продукции идентификатор;

создать базовую версию;

подготовить руководство по поставляемой продукции.

Для обзора поставляемой продукции назначается руководитель обзора, который отвечает за подготовку и проведение обзора в соответствии с процедурой, определенной в процессе организации.

В ходе обзора проверяются полнота поставляемого продукта и правильное выполнение процедуры поставки. Вся выпускаемая продукция должна быть классифицирована в соответствии с руководством по поставке ПП.

Контрольные вопросы

1. Что включает в себя процедура поставки?
2. Для каких целей используется схема классификации поставляемого программного продукта?
3. Какие типы поставок вы знаете? Охарактеризуйте каждый из них.
4. Какие действия необходимо выполнять при подготовке поставляемого программного продукта к обзору?
5. Кто отвечает за проведение обзора и подготовляемой продукции к нему?

ГЛАВА 13

ОБЕСПЕЧЕНИЕ НАДЕЖНОСТИ ПРОГРАММНЫХ ПРОДУКТОВ

13.1. Используемые термины

При определении надежности ПП пользуются следующими принятыми терминами.

Надежность — состояние, позволяющее избежать повреждений в момент совершения ошибки. Ошибки ПП происходят в силу дефектов или ошибок проекта, кодирования, организационных ошибок, неадекватной отладки и ошибок тестирования (определения понятиям «ошибка» и «дефект» даны в подразд. 5.4.3).

Отказоустойчивость ПП — свойство ПП, заключающееся в возможности коррекции отдельных ошибок при сохранении возможности продолжения выполнения программы.

Проблема — отклонение от заданных технических характеристик или ожидаемых результатов.

Ошибка при обработке — вывод некорректных результатов при выполнении процесса обработки.

Процесс — ограниченный ряд взаимосвязанных действий, в ходе осуществления которых используются один или больше типов исходных продуктов, а затем с помощью одного или нескольких преобразований создается конечный продукт, который представляет ценность для заказчика.

Отказ при выполнении процесса — событие, посредством которого ошибка в исходном продукте, используемом в процессе, порождает ошибку на выходе, которая в конечном итоге становится явной.

Сбой при выполнении процесса — сбой, имеющий отношение к используемым в процессе некорректным входным данным и вызывающий неправильное состояние процессе или системы, к которой относится процесс.

Устойчивость — см. определение в подразд. 8.4.

13.2. Основные понятия о надежности программных продуктов и методах ее обеспечения

Надежность считается ключевым показателем качества ПП. Особое внимание обеспечению надежности ПП уделяется в силу

того, что этот показатель наиболее важен для конечного пользователя, а факторы качества, связанные с изменением ПП и разработкой его новых версий, имеют определяющее значение для разработчиков ПП и групп технической поддержки. Вряд ли кому понравится программа либо полностью неработоспособная, либо некорректно работающая.

На рис. 13.1 представлена топология факторов качества, предложенная Р. Макколом, Б. Ричардсом и С. Уолтерсом в 1977 г.

Издание «IEEE. Standart Glossary of Software Engineering Terms» («IEEE. Перечень стандартных терминов, используемых в программной инженерии») определяет надежность ПП как способность системы или ее компонента выполнять требуемые функции в заданных условиях на протяжении указанного периода времени. Степень надежности ПП непосредственно зависит от совершенства процесса разработки. Основной показатель, влияющий на надежность ПП — сложность разрабатываемых программ.

Процесс создания надежного ПП в отличие от аппаратного обеспечения не зависит от времени, что демонстрируют модели надежности аппаратного обеспечения и ПП, имеющие вид традиционных U-образных кривых, представленных соответственно на рис. 13.2 и 13.3, где λ — планируемое число сбоев.

Рис. 13.1. Топология факторов качества программного продукта

Рис. 13.2. U-образная кривая надежности аппаратного обеспечения

Методы оценки надежности ПП окончательно еще не разработаны, однако если не выполнять адекватную оценку данных о случившихся сбоях, невозможным становится и выполнение расширенных статических моделей, требующихся для анализа реальной степени надежности. До сих пор еще не был разработан ни один надежный количественный метод оценки надежности, не содержащий чрезмерное число ограничений.

Степень надежности ПП можно улучшать с помощью различных методов, тем не менее, трудно достичь нужного соотношения между временем разработки, ее бюджетной стоимостью и кажущейся высокой ценой, уплаченной за достигнутую надежность ПП.

В отличие от аппаратного обеспечения ПП с течением времени не «изнашивается», просто выявляются все новые и новые его дефекты. Изображенная на рис. 13.3 U-образная кривая демонстрирует распределение сбоев на протяжении эксплуатации про-

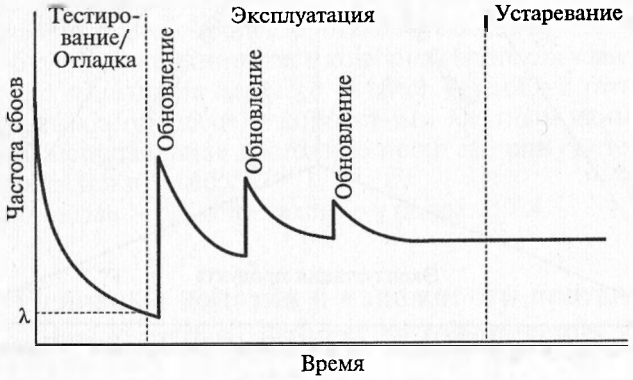


Рис. 13.3. U-образная кривая надежности программных продуктов

граммы. Проявление сбоев в ПП отличается от характера сбоев аппаратуры, причем их вероятность всегда выше нуля.

При определении размера требуемых инвестиций для обеспечения желаемой надежности ПП необходимо учитывать вопросы, связанные с риском появления ненадежных программ. Баланс между затратами и достигаемой надежностью при проектировании ПП определяется критерием уменьшения указанного риска. Если достигаемая надежность не позволяет значительно уменьшить риск, то на ее обеспечение не стоит тратить дополнительных средств.

Большинство проблем надежности ПП не являются жизненно важными. Основная масса проблем, связанных с достижением надежности, относится к тестированию ПП.

Остановимся на рассмотрении четырех методов, обеспечивающих создание высоконадежного ПП:

прогнозирование ошибок — создание моделей надежности, анализ исторических данных, сбор информации об ошибках, профилирование операционной среды;

предотвращение ошибок — использование формальных методов, повторное использование программ, применение инструментов конструирования программ;

устранение ошибок — формальное инспектирование, верификация и аттестация;

обеспечение отказоустойчивости — использование методов мониторинга, верификация решений, анализ избыточности, исключительных ситуаций.

13.3. Методы обеспечения надежности на различных этапах жизненного цикла разработки программного продукта

Надежность ПП необходимо планировать на начальных стадиях выполнения проекта. Процесс определения надежности разрабатываемого ПП требует сбора большого количества информации. Действия по сбору и анализу метрических данных описаны в гл. 5. Методы измерения вырабатываются разработчиками ПП в течение всего жизненного цикла. Методы обеспечения надежности, реализуемые на различных этапах жизненного цикла разработки ПП, приведены на рис. 13.4.

Прогнозирование ошибок выполняется на этапах планирования и составления требований, предотвращение ошибок — на этапах составления требований, проектирования и разработки, устранение ошибок — на этапах проектирования, разработки и тестирования. Период отказоустойчивости начинается на этапе разработки и длится до окончания жизненного цикла ПП.

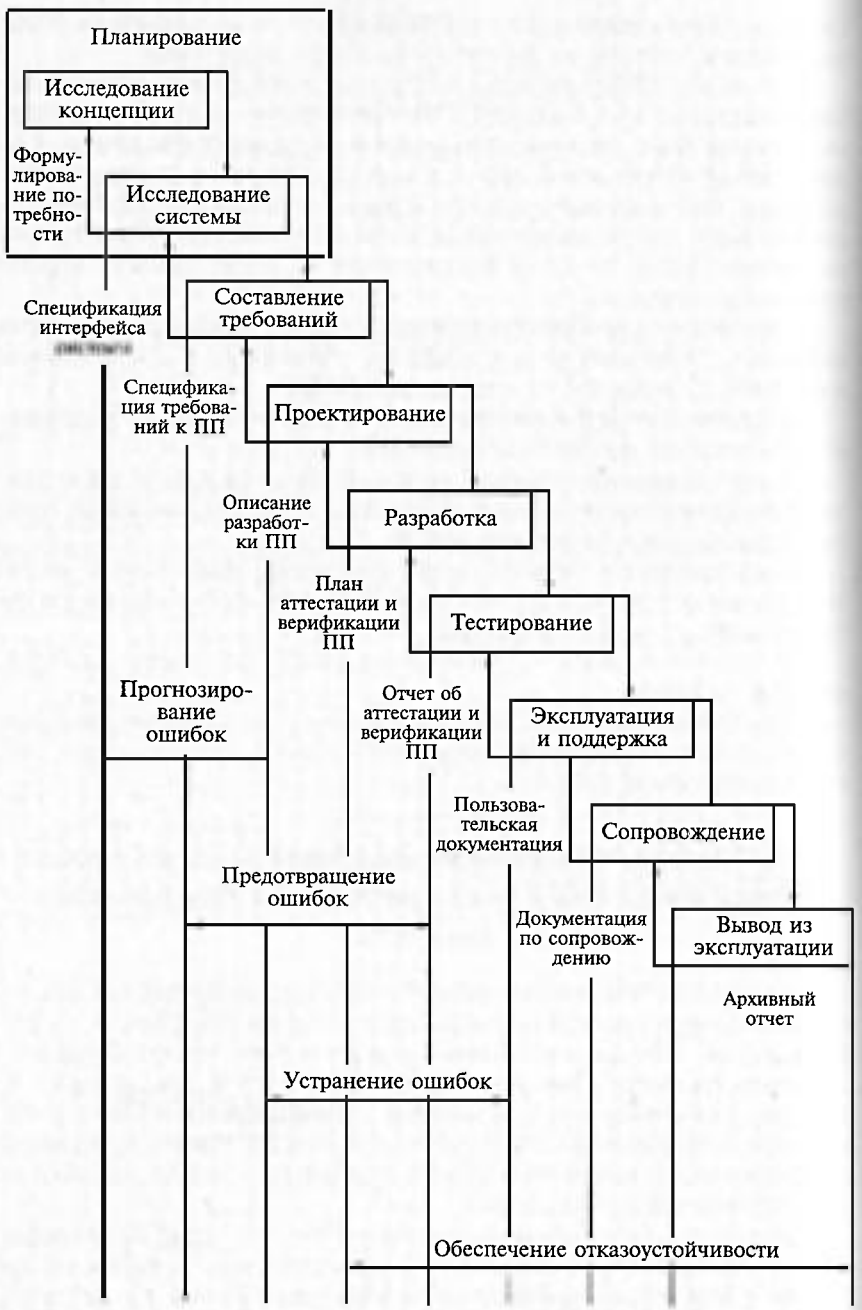


Рис. 13.4. Методы обеспечения надежности на различных этапах жизненного цикла разработки программного продукта

Таблица 13.1

Связь методов обеспечения надежности с этапами жизненного цикла разработки программного продукта

Этапы жизненного цикла разработки ПП	Основные действия	Прогнозирование ошибок	Предотвращение ошибок	Устранение ошибок	Обеспечение отказоустойчивости
Планирование и составление требований	Определение функционального профиля	+	+		
	Определение и классификация ошибок	+	+		
	Определение потребностей заказчиков в обеспечении требуемого уровня надежности ПП	+	+		
	Проведение альтернативных учебных курсов	+	+		
	Определение целей, связанных с обеспечением надежности	+	+		
Проектирование и разработка	Распределение функций по обеспечению надежности между всеми компонентами ПП		+	+	+
	Встречи с инженерами для установления целей по достижению надежности		+	+	+
	Сосредоточение ресурсов на основе функционального профиля		+	+	+
	Управление вводом и распространением ошибок		+	+	+
	Измерение надежности приобретенного ПП		+	+	+

Окончание табл. 13.1

Этапы жизненного цикла разработки ПП	Основные действия	Прогнозирование ошибок	Предотвращение ошибок	Устранение ошибок	Обеспечение отказоустойчивости
Тестирование	Определение эксплуатационного профиля			+	+
	Тестирование степени увеличения надежности			+	+
	Отслеживание хода выполнения тестирования			+	+
	Определение потребности в дополнительном тестировании проекта			+	+
	Определение, достигнута ли требуемая степень надежности			+	+
Эксплуатация и сопровождение	Определение потребности в персонале на завершающих стадиях проекта				+
	Сравнительный мониторинг целей создания ПП и его надежности				+
	Отслеживание степени удовлетворенности заказчиков достигнутым уровнем надежности				+
	Мониторинг надежности при добавлении новых свойств				+
	Управление улучшениями продукта и процесса с одновременным измерением достигнутой надежности				+

Связь методов обеспечения надежности с этапами жизненного цикла разработки ПП показана в табл. 13.1.

13.4. Прогнозирование ошибок

Прогнозирование ошибок означает предсказуемый подход к разработке надежного ПП. Зрелые организации, специализирующиеся на разработке ПП, выполняют прогнозирование ошибок как составную часть оценивания проекта/процесса ПП. Единственный способ достижения даже небольшой степени точности для прогнозирующих моделей заключается в предоставлении доступа к соответствующим историческим моделям обеспечения надежности данных. Анализ исторических данных и сбор данных об ошибках являются ключевыми действиями для данного метода. В табл. 13.2 (шаблон) должны быть отражены результаты прогнозирования ошибок.

Определение функционального профиля является первым действием при прогнозировании ошибок. Прослеживая состояния переходов от модуля к модулю и от функции к функции, можно точно выявить наиболее уязвимое место системы. Если объединить полученную информацию с функциональным профилем, можно определить, насколько надежной будет система при заданных условиях ее использования.

При выполнении программ осуществляются отслеживаемые переходы между модулями. При переходе к программным модулям, которые перегружены ошибками, возрастает риск неуда-

Таблица 13.2

Шаблон для учета результатов прогнозирования ошибок на этапах планирования и составления требований

Основные действия при прогнозировании ошибок на этапах планирования и составления требований	Результаты прогнозирования ошибок
Определение функционального профиля	(Перечисление результатов)
Определение и классификация ошибок	То же
Определение потребностей заказчиков в обеспечении требуемого уровня надежности ПП	»
Проведение альтернативных учебных курсов	»
Определение целей, связанных с обеспечением надежности	»

чи. Моделирование подобных переходов осуществляется с помощью некоего стохастического процесса. В конечном итоге при разработке математического описания поведения ПП, обусловленного выполняемыми функциями различных модулей ПП, возможно создание описания надежности выполняемых функций. Программная система состоит из собственных выполняемых функций. Имея представление о надежности выполняемых функций и порядке распределения системой машинного времени среди этих функций, можно сделать заключение о надежности самой системы.

Определение и классификация ошибок — второе действие при их прогнозировании. Как указывалось в подразд. 13.1, ошибки ПП являются результатом дефектов или ошибок проекта, дефектов или ошибок кодирования, организационных ошибок, неадекватной отладки и ошибок тестирования. Определение ошибок подразумевает установление их источника. Классификация ошибок производится по степени их серьезности. Классификация, разработанная Борисом Бейцером, имеет соответствующие уровни детализации.

1. Слабый — симптомы имеют чисто эстетическое проявление.
2. Умеренный — выходные данные некорректны или избыточны.
3. Раздражающий — некорректное поведение системы.
4. Очень серьезный — проблема в работе системы, затрагивающая нескольких пользователей (например, вместо того, чтобы учесть платежный чек клиента, система переводит деньги на чужой счет).
5. Экстремальный — проблема в работе системы, затрагивающая широкий круг пользователей.
6. Невыносимый — база данных надолго и непоправимо выходит из строя.
7. Катастрофический — решение о прекращении выполнения программы исходит от пользователя, система разрушается.
8. Инфекционный — система разрушает другие системы, даже если ее собственная работоспособность не нарушена.

Завершив составление матрицы источников ошибок и их классификацию, необходимо отследить метрические показатели, характеризующие ошибочные данные, как показано в табл. 13.3 (шаблон). Каждая ячейка таблицы содержит итоговые сведения об ошибках (итог) согласно исторической информации, доступной для аналогичных проектов и продуктов.

Определение потребностей заказчиков в обеспечении требуемого уровня надежности ПП является третьим действием при прогнозировании ошибок. Эти потребности заранее определены и задокументированы в первичных требованиях заказчика. Потребности заказчика в требуемом уровне надежности должны быть установлены с той точностью, которую обеспечивают методы из-

Таблица 13.3

Шаблон для учета итоговых сведений об ошибках

Уровень ошибки	Ошибки или дефекты проекта	Ошибки или дефекты кодирования	Организационные ошибки	Неадекватная отладка	Ошибки тестирования
Слабый	Итог	Итог	Итог	Итог	Итог
Умеренный		»	»	»	»
Раздражающий		»	»	»	»
Очень серьезный		»	»	»	»
Экстремальный		»	»	»	»
Невыносимый		»	»	»	»
Катастрофический		»	»	»	»
Инфекционный		»	»	»	»

мерения. Несмотря на то что все требования должны быть измеримыми и обеспечивать возможность контроля, требованиям, связанным с обеспечением надежности, присваиваются определенные номера. Ниже приведены соответствующие номерам примеры утверждений, касающихся надежности.

1. Надежность системы управления спутниками оценивается как 0,999 от 95 % доверия на период, равный 15 годам с момента разделения полезной нагрузки.
2. Для авиационной системы характерно в среднем 500 летных часов между критическими ошибками.
3. Встроенная система самоконтроля определит, что ракета неисправна, с вероятностью 60 %.
4. Встроенная система самоконтроля посчитает исправную ракету за неисправную с вероятностью менее 1 %.
5. Среднее время исправления функционирующего ПП составляет 30 мин или меньше.
6. Максимальное время завершения работы встроенной системы самоконтроля составляет 20 с.
7. Среднее время загрузки ПП в полном объеме и выполнения полного внутреннего самоконтроля составляет 10 мин.
8. Среднее время обновления одной интерактивной страницы документации составляет 30 мин.

Проведение альтернативных учебных курсов — четвертое действие при прогнозировании ошибок. С помощью клиентского функционального профиля и информации о классификации ошибок, почерпнутых из предыдущих систем, анализируются

особые требования для определения того, поддерживают ли цели исторические данные. Производится анализ курсов для определения вероятности достижения целей надежности, сформулированных в требованиях. При отсутствии исторических данных, связанных с аналогичными продуктами и системами, вероятность достижения искусственно установленного уровня надежности чрезвычайно низкая. На этой стадии руководителю проекта необходимо разработать экстенсивные системные модели, применяемые для определения возможных уровней надежности нового ПП. Такие инструменты, как формальные методы, применимы к требованиям надежности для математических доказательств, связанных с наиболее критическими подсистемами. Этот дорогостоящий процесс используется исключительно в случаях отсутствия источника исторических данных о надежности.

Определение целей, связанных с обеспечением надежности, — пятое действие при прогнозировании ошибок, базирующееся на результатах проведения альтернативных учебных курсов. Окончательный набор целей обеспечения надежности передается процессу определения требований, позволяя изменить уже существующие. Благодаря данному действию собранная информация и результаты проведенного анализа передаются для спецификации требований. Цели и требования, касающиеся надежности, будут использованы для подтверждения надежности всей системы и получения одобрения пользователя.

13.5. Предотвращение ошибок

Предотвращение ошибок включает в себя интерактивное уточнение системных требований и разработку технических характеристик ПП наравне с моделированием, проверяемыми методиками проекта и оптимальными способами кодирования. Этот метод обеспечения надежности применяется на этапах составления требований, проектирования и разработки ПП. Предотвращение ошибок является активной частью процесса проектирования. Первая стадия предотвращения ошибок заключается в исследовании системы в целом и требований к ПП. Первоначальные шаги необходимы для анализа требований и целей, касающихся надежности ПП, независимо от применяемых подходов, призванных помочь в достижении его надежности.

Второй основной стадией предотвращения ошибок являются проектирование и разработка проекта. При этом функции по обеспечению надежности распределяются между всеми компонентами ПП. Определяется процесс проектирования, в результате которого создаются компоненты. Методики улучшения степени па-

раллельности и увеличения связности модулей обеспечивают повышение надежности компонентов. Целесообразно применять наилучшие наработки, относящиеся к проектированию, причем независимо от того, являются они структурированными или объектно-ориентированными. Благодаря этому обеспечивается возможность разработки ПП, обладающих высокой степенью надежности.

На ранних этапах жизненного цикла разработки ПП целью обеспечения надежности является предотвращение возможных ошибок. В табл. 13.4 (шаблон) должны быть перечислены все действия, предпринимаемые в организации для поддержания работы по предотвращению ошибок.

Цели обеспечения надежности, предварительно определенные и задокументированные, должны быть установлены для всех модулей во время этапа проектирования. Руководителю проекта следует сосредоточить ресурсы на основе функционального профи-

Таблица 13.4

Шаблон для учета действий по предотвращению ошибок на этапах составления требований, проектирования и разработки

Основные действия по предотвращению ошибок на этапах составления требований, проектирования и разработки	Состав действий по предотвращению ошибок
Определение функционального профиля	(Перечисление действий)
Определение и классификация ошибок	То же
Определение потребностей заказчиков в обеспечении требуемого уровня надежности ПП	»
Проведение альтернативных учебных курсов	»
Определение целей, связанных с обеспечением надежности	»
Распределение функций по обеспечению надежности между всеми компонентами ПП	»
Встречи с инженерами для установления целей по достижению надежности	»
Сосредоточение ресурсов на основе функционального профиля	»
Управление вводом и распространением ошибок	»
Измерение надежности приобретенного ПП	»

ля, который должен использоваться и проходить аттестацию в ходе этапа проектирования.

Единственный способ активного предотвращения ошибок заключается в управлении вводом и распространением ошибок.

Экспертные оценки и инспекционные проверки — традиционный способ активного сокращения ошибок, появляющихся на одном этапе, и предотвращения их перехода на другой этап. Еще одно важное действие — измерение надежности приобретенного ПП. При расширенном использовании инструментальных средств, связанных с сетью Internet, и при более доступных совместно используемых библиотеках программное обеспечение неизвестного происхождения (SOUP — Software Of Uncertain Pedigree) становится частью продукта. Команда разработчиков проекта нуждается в процессе верификации, аттестации, принятия и оценки надежности SOUP-компонентов. Это должен быть формальный процесс с тем же уровнем отслеживания и конфигурации, что и в случае с компонентами ПП, созданными «с нуля». Повторное использование программ обеспечивает огромное повышение продуктивности разработчика. Однако при этом могут проявляться скрытые проблемы и дефекты.

13.6. Устранение ошибок

Старая поговорка о том, что день, потраченный на предотвращение, стоит года, потраченного на исправление последствий, остается весьма актуальной. Устранение возникающей в системе ошибки обходится в 10—100 раз дороже, чем ее предотвращение на начальном этапе.

Процесс устранения ошибок начинается на этапе проектирования ПП и распространяется на этапы разработки и тестирования. После завершения работы по функциональному профилированию осуществляется следующий шаг — тестирование степени увеличения надежности ПП, называемое также испытанием под нагрузкой. Цель такого тестирования — определить совокупность нагрузок, при которых система выходит из строя. Это формальный процесс, при котором отслеживание хода выполнения тестирования имеет определенное значение. Результаты тестов анализируются с целью их применения для повторной калибровки моделей, применяемых для прогнозирования надежности на начальных стадиях проекта. Одной из задач руководителя проекта является поддержание постоянного совершенствования процесса разработки. Благодаря использованию данных, полученных при выполнении предыдущих проектов, в текущем проекте поддерживается способность организации к обучению и совершенствованию.

Таблица 13.5

Шаблон для учета действий по устранению ошибок на этапах проектирования, разработки и тестирования

Основные действия по устранению ошибок на этапах проектирования, разработки и тестирования	Состав действий по устранению ошибок
Распределение функций по обеспечению надежности между всеми компонентами ПП	(Перечисление действий)
Встречи с инженерами для установления целей по достижению надежности	То же
Сосредоточение ресурсов на основе функционального профиля	»
Управление вводом и распространением ошибок	»
Измерение надежности приобретенного ПП	»
Определение эксплуатационного профиля	»
Тестирование степени увеличения надежности	»
Отслеживание хода выполнения тестирования	»
Определение потребности в дополнительном тестировании проекта	»
Определение, достигнута ли требуемая степень надежности	»

На средних этапах жизненного цикла разработки ПП усилия по обеспечению надежности сосредоточиваются на устранении дефектов. В табл. 13.5 (шаблон) должны быть перечислены все предпринимаемые в организации действия по устранению ошибок, выполняемые на основных этапах жизненного цикла разработки.

Проектирование модуля, выполняющего необходимое тестирование, является результатом анализа тестируемых данных. Результаты тестирования дополнительной нагрузкой могут быть неадекватны, вследствие чего сложно подтверждать реализацию всех целей обеспечения надежности. Некоторые модули, возможно, должны будут пройти повторное тестирование методом «черного» или «белого» ящика. При этом должно быть расширено и увеличено в объеме регрессионное тестирование. К моменту завершения устранения ошибок руководитель проекта должен быть удовлетворен результатами, свидетельствующими о достижении целей обеспечения надежности.

13.7. Обеспечение отказоустойчивости

Принципы достижения отказоустойчивости ПП (см. определение в подразд. 13.1) в значительной степени отличаются от принципов достижения отказоустойчивости аппаратного обеспечения. В системе, построенной на основе отказоустойчивого аппаратного обеспечения, параллельно функционируют несколько наборов аппаратных средств. При этом происходит дублирование всех устройств с тем, чтобы при выходе из строя одного устройства другое сразу же могло бы продолжить работу.

Попытка реализовать отказоустойчивость ПП таким же образом, т. е. параллельным выполнением одной и той же программы различными процессорами, приводит только к тому, что вторая копия точно такой же программы отстает на доли секунды от первой копии. Простое выполнение отдельной копии программы никак не отражается на отказоустойчивости ПП.

Начиная со средних этапов жизненного цикла разработки ПП, усилия по обеспечению надежности фокусируются на достижении отказоустойчивости. В табл. 13.6 (шаблон) должны быть перечислены все предпринимаемые в организации действия по обеспечению отказоустойчивости, выполняемые на основных этапах жизненного цикла разработки.

Процесс обеспечения отказоустойчивости начинается на этапе разработки ПП и распространяется на этапы тестирования, эксплуатации и сопровождения. Завершение этого процесса определяется моментом устаревания ПП. До тех пор пока ПП работает в обычном режиме, отказоустойчивый подход к обеспечению надежности используется достаточно широко.

Обеспечение отказоустойчивости представляет собой логическое продолжение этапа устранения ошибок. В данном случае применяются все действия, используемые при устранении ошибок. Различие заключается в целях, преследуемых после завершения процесса разработки ПП. Потребности персонала, занятого сопровождением ПП, определяются только с учетом информации, касающейся разработки предыдущих версий ПП. Организация должна иметь доступ к базе данных об ошибках, обнаруженных после установки других ПП.

Подобная информационная структура, включающая в себя описание ошибок и действий, предпринятых для их устранения, используется для оценки необходимых трудозатрат на этапе сопровождения ПП. Используя хронологические сведения о результатах оценок ошибок, обнаруженных на этапе разработки, и зная соотношение между объемом нового продукта (в LOC) и объемами других ПП, можно выполнить быструю оценку остальных ошибок и трудозатрат, необходимых для их устранения в новой версии продукта.

Чтобы обеспечить набор данных для разрабатываемых ПП, руководитель проекта должен контролировать надежность ПП в ходе эксплуатационных испытаний, учитывая цели создания ПП. Это позволяет отслеживать степень удовлетворенности заказчиков достигнутым уровнем надежности. Конечный пользователь — наилучший источник информации о надежности ПП.

Таблица 13.6

Шаблон для учета действий по обеспечению отказоустойчивости на этапах разработки, тестирования, эксплуатации и сопровождения

Основные действия по обеспечению отказоустойчивости на этапах разработки, тестирования, эксплуатации и сопровождения	Состав действий по обеспечению отказоустойчивости
Распределение функций по обеспечению надежности между всеми компонентами ПП	(Перечисление действий)
Встречи с инженерами для установления целей по достижению надежности	То же
Сосредоточение ресурсов на основе функционального профиля	»
Управление вводом и распространением ошибок	»
Измерение надежности приобретенного ПП	»
Определение эксплуатационного профиля	»
Тестирование степени увеличения надежности	»
Отслеживание хода выполнения тестирования	»
Определение потребности в дополнительном тестировании проекта	»
Определение, достигнута ли требуемая степень надежности	»
Определение потребности в персонале на завершающих стадиях проекта	»
Сравнительный мониторинг целей создания ПП и его надежности	»
Отслеживание степени удовлетворенности заказчиков достигнутым уровнем надежности	»
Мониторинг надежности при добавлении новых свойств	»
Управление улучшениями продукта и процесса с одновременным измерением достигнутой надежности	»

Руководитель проекта должен определить время предоставления заказчикам нового свойства ПП. Не стоит предоставлять заказчикам новые свойства продукта, прежде чем будут устранены все связанные с этими свойствами ошибки. Поставка ПП с набором новых свойств и исправленными ошибками является наиболее подходящей практикой для организаций — разработчиков ПП.

Управление улучшениями продукта и процесса с одновременным измерением достигнутой надежности дополняет процесс повышения надежности на базе информации, собранной на основе практики заказчика. При этом устраняются ошибки ПП и совершенствуется непрерывный процесс разработки. Оценивание надежности является весьма дорогостоящей процедурой. Ее результаты передаются обучающей организации.

13.8. Инструменты, обеспечивающие надежность программных продуктов. План обеспечения надежности

Основой для всех инструментов обеспечения надежности является статистический анализ, поэтому любые программные инструменты, способные анализировать наборы данных и выполнять элементарные статистические вычисления (например, MS Excel), могут быть использованы для этих задач.

Деятельность по обеспечению надежности ПП является очень дорогостоящей. Как и любая другая деятельность по разработке ПП, она должна быть запланирована и задокументирована. Ниже приведена схема плана обеспечения надежности ПП. Предлагаемый план является производным множества планов, опубликованных IEEE, SEI и ISO.

Схема плана обеспечения надежности

1. Заключение относительно потребностей в надежности.
2. Определения, акронимы и аббревиатуры, ссылки на вопросы, связанные с обеспечением надежности.
3. Взаимосвязь с действиями по управлению рисками:
   a. Уменьшение специфических рисков;
   b. Сбережения бюджета проекта;
   c. Влияние надежности ПП;
   d. Описание методов, предназначенных для обеспечения надежности (прогнозирование ошибок; предотвращение ошибок; устранение ошибок; обеспечение отказоустойчивости).
4. Метод прогнозирования ошибок;
   a. Определение функционального профиля;

   b. Определение ошибок;
   c. Схема классификации ошибок и отказов;
   d. Определение потребностей заказчиков в обеспечении требуемого уровня надежности;
   e. План проведения альтернативных учебных курсов;
   f. Определение целей, связанных с обеспечением надежности ПП.
5. Метод предотвращения ошибок:
   a — f такие же, как в п. 4 плана;
   g. Распределение функций по обеспечению надежности между всеми компонентами ПП;
   h. Разработка процесса, удовлетворяющего цели обеспечения надежности;
   i. План сосредоточения ресурсов на основе функционального профиля;
   j. План управления вводом и распространением ошибок.
6. Метод устранения ошибок:
   a — d соответствуют подпунктам g — j в п. 5 плана;
   e. План оценки надежности;
   f. Определение эксплуатационного профиля;
   g. План тестирования степени увеличения надежности;
   h. План отслеживания хода выполнения тестирования;
   i. План дополнительного тестирования;
   j. Процесс сертификации целей надежности.
7. Метод обеспечения отказоустойчивости:
   a — j такие же, как в п. 6 плана;
   k. Определение потребности в персонале на завершающих стадиях проекта;
   l. Сравнительный мониторинг целей создания ПП и его надежности при испытаниях у заказчика;
   m. Отслеживание степени удовлетворенности заказчиков достигнутым уровнем надежности;
   n. Расчет времени предоставления нового свойства ПП путем отслеживания надежности ПП;
   o. Управление улучшениями продукта и процесса с одновременным измерением достигнутой надежности.
8. Одобрение плана обеспечения надежности.

Контрольные вопросы

1. Дайте определение понятия «надежность программного продукта».
2. Нарисуйте и объясните кривую распределения ошибок на протяжении времени эксплуатации программного продукта.
3. Перечислите и охарактеризуйте основные методы создания высоконадежного программного продукта.
4. Какие методы обеспечения надежности используются на этапах:
   а) планирования и составления требований;
   б) проектирования и разработки;
   в) тестирования;

г) эксплуатации и сопровождения.

5. Дайте определения понятий «отказоустойчивость», «проблема», «ошибка при обработке», «процесс», «отказ при выполнении процесса», «сбой при выполнении процесса», «устойчивость», «ошибка программного продукта».

6. Перечислите и охарактеризуйте этапы прогнозирования ошибок.

7. Назовите основные классы ошибок.

8. Объясните назначение матрицы источников ошибок.

9. Перечислите основные действия, направленные:

а) на предотвращение ошибок;

б) устранение ошибок;

в) обеспечение отказоустойчивости.

10. Назовите и поясните основные пункты плана обеспечения надежности.

ГЛАВА 14

ОСНОВНЫЕ ПОНЯТИЯ И НАЗНАЧЕНИЕ ЯЗЫКА UML

14.1. Назначение языка UML

Язык UML представляет собой общецелевой язык визуального моделирования, который разработан для спецификации, визуализации, проектирования и документирования компонентов программного обеспечения, бизнес-процессов и других различных систем. Язык UML является одновременно простым и мощным средством моделирования. Он может быть эффективно использован для построения концептуальных, логических и графических моделей сложных систем самого различного целевого назначения. Этот язык вобрал в себя наилучшие качества методов программной инженерии, которые с успехом использовались на протяжении последних лет при моделировании больших и сложных систем.

Язык UML основан на некотором числе базовых понятий, которые могут быть изучены и применены большинством программистов и разработчиков, знакомых с методами объектно-ориентированного анализа и проектирования (ООАП). При этом базовые понятия могут комбинироваться и расширяться таким образом, что специалисты объектного моделирования получают возможность самостоятельно разрабатывать модели больших и сложных систем в самых различных областях приложений.

Конструктивное использование языка UML основывается на понимании общих принципов моделирования сложных систем и особенностей процесса объектно-ориентированного анализа и проектирования в частности. Выбор выразительных средств для построения моделей сложных систем предопределяет те задачи, которые могут быть решены с использованием данных моделей. При этом одним из основных принципов построения моделей сложных систем является принцип *абстрагирования*, который предписывает включать в модель только те аспекты проектируемой системы, которые имеют непосредственное отношение к выполнению системой своих функций или к ее целевому предназначению. При этом все второстепенные детали опускаются, чтобы чрезмерно не усложнять процесс анализа и исследования полученной модели.

Другим принципом построения моделей сложных систем является принцип *многомодельности*. Этот принцип представляет собой утверждение о том, что никакая единственная модель не может с достаточной степенью адекватности описывать различные аспекты сложной системы. Применительно к методологии ООАП это означает, что достаточно полная модель сложной системы допускает некоторое число взаимосвязанных представлений (views), каждое из которых адекватно отражает некоторый аспект поведения или структуры системы. При этом наиболее общими представлениями сложной системы принято считать статическое и динамическое представления, которые, в свою очередь, могут подразделяться на другие более частные представления. Феномен сложной системы как раз и состоит в том, что никакое ее единственное представление не является достаточным для адекватного выражения всех ее особенностей.

Еще одним принципом прикладного системного анализа является принцип *иерархического построения моделей* сложных систем. Этот принцип предписывает рассматривать процесс построения модели на разных уровнях абстрагирования или детализации в рамках фиксированных представлений. При этом исходная, или первоначальная, модель сложной системы имеет наиболее общее представление (метапредставление). Такая модель строится на начальном этапе проектирования и может не содержать многих деталей и аспектов моделируемой системы.

Таким образом, процесс ООАП можно представить как поуровневый спуск от наиболее общих моделей и представлений концептуального уровня к более частным и детальным представлениям логического и физического уровней. При этом на каждом из этапов ООАП данные модели последовательно дополняются все большим числом деталей, что позволяет им более адекватно отражать различные аспекты конкретной реализации сложной системы. Общая схема взаимосвязей моделей ООАП представлена на рис. 14.1.

Термин «физическая модель» в ООАП и языке UML имеет трактовку, отличающуюся от общепринятой в общей классификации моделей систем.

В последнем случае под физической моделью системы понимают некоторую материальную конструкцию, обладающую свойствами подобия с формой оригинала. Примерами таких моделей могут служить модели технических систем (самолетов, кораблей), архитектурных сооружений (зданий, микрорайонов). Что касается этого термина в ООАП и языке UML, то здесь физическая модель отражает компонентный состав проектируемой системы с точки зрения ее реализации на некоторой технической базе и вычислительных платформах конкретных производителей.

Рис. 14.1. Общая схема взаимосвязей моделей и представлений сложной системы в процессе объектно-ориентированного анализа и проектирования

Создание языка UML предусматривало решение следующих задач.

1. Предоставить в распоряжение пользователей легко воспринимаемый и выразительный язык визуального моделирования, специально предназначенный для разработки и документирования моделей сложных систем самого различного целевого назначения. Важным фактором дальнейшего развития и повсеместного использования методологии ООАП является интуитивная ясность и понятность основных конструкций соответствующего языка моделирования. Язык UML включает в себя не только абстрактные конструкции для представления метамоделей систем, но и целый ряд конкретных понятий, имеющих вполне определенную семантику. Это позволяет языку UML одновременно достичь не только универсальности представления моделей для самых различных приложений, но и возможности описания достаточно тонких деталей реализации этих моделей применительно к конкретным системам.

Практика системного моделирования показала, что абстрактного описания языка на некотором метауровне недостаточно для разработчиков, которые ставят своей целью реализацию проекта системы в конкретные сроки. В настоящее время имеет место некоторый концептуальный разрыв между общей методологией моделирования сложных систем и конкретными инструментальными средствами быстрой разработки приложений. Именно этот разрыв и призван заполнить язык UML.

Отсюда следует, что для адекватного понимания базовых конструкций языка UML важно не только владеть некоторыми навыками объектно-ориентированного программирования, но и хорошо представлять себе общую проблематику процесса разработки моделей систем. Именно интеграция этих представлений образует новую парадигму ООАП, практическим следствием и центральным стержнем которой является язык UML.

2. Снабдить исходные понятия языка UML возможностью расширения и специализации для более точного представления моделей систем в конкретной предметной области. Хотя язык UML является формальным языком — языком спецификаций, формальность его описания отличается как от традиционных формально-логических языков, так и от известных языков программирования. Разработчики из OMG (Object Management Group — консорциум, созданный в 1989 г. для разработки индустриального стандарта языка UML) предполагают, что язык UML как никакой другой может быть приспособлен для конкретных предметных областей. Это становится возможным по той причине, что в самом описании языка UML заложен механизм расширения базовых понятий, который является самостоятельным элементом языка и имеет собственное описание в форме правил расширения.

В то же время разработчики из OMG считают крайне нежелательным переопределение базовых понятий языка по какой бы то ни было причине. Это может привести к неоднозначной интерпретации их семантики и возможной путанице. Базовые понятия языка UML не следует изменять больше, чем это необходимо для их расширения. Все пользователи должны быть способны строить модели систем для большинства обычных приложений с применением только базовых конструкций языка UML, без использования механизма расширения. При этом новые понятия и нотации целесообразно применять только в тех ситуациях, когда имеющихся базовых понятий явно недостаточно для построения моделей системы.

Язык UML допускает также специализацию базовых понятий. Речь идет о том, что в конкретных приложениях пользователи должны уметь дополнять имеющиеся базовые понятия новыми характеристиками или свойствами, которые не противоречат семантике этих понятий в языке UML.

3. Обеспечить, чтобы описание языка UML поддерживало такую спецификацию моделей, которая не зависит от конкретных языков программирования и инструментальных средств проектирования программных систем. Ни одна из конструкций языка UML не должна зависеть от особенностей ее реализации в известных языках программирования. Хотя отдельные понятия языка UML и связаны с последними очень тесно, их жесткая интерпретация в форме каких бы то ни было конструкций программирования не

может быть признана корректной. Другими словами, разработчики из OMG считают необходимым свойством языка UML его контекстно-программную независимость.

С другой стороны, язык UML должен обладать потенциальной возможностью реализации своих конструкций на том или ином языке программирования. Конечно, в первую очередь, имеются в виду языки, поддерживающие концепцию объектно-ориентированного проектирования, такие как C++, Java, Object Pascal. Именно это свойство языка UML делает его современным средством решения задач моделирования сложных систем. В то же время предполагается, что для программной поддержки конструкций языка UML могут быть разработаны специальные инструментальные CASE-средства. Наличие последних имеет принципиальное значение для широкого распространения и использования языка UML.

4. Обеспечить, чтобы описание языка UML включало в себя семантический базис для понимания общих особенностей ООАП. Говоря об этой особенности, имеют в виду самодостаточность языка UML для понимания не только его базовых конструкций, но, что не менее важно, и общих принципов ООАП. Поскольку язык UML не является языком программирования, а служит средством для решения задач объектно-ориентированного моделирования систем, его описание должно по возможности включать в себя все необходимые понятия для ООАП. Без этого свойства язык UML может оказаться бесполезным и невостребованным большинством пользователей, которые не знакомы с проблематикой ООАП сложных систем.

С другой стороны, какие бы то ни было ссылки на дополнительные источники, содержащие важную для понимания языка UML информацию, по мнению разработчиков из OMG, должны быть исключены. Это позволит избежать неоднозначного толкования принципиальных особенностей процесса ООАП и реализации этих особенностей в форме базовых конструкций языка UML.

5. Поощрять развитие рынка объектных инструментальных средств. Более 800 ведущих производителей программных и аппаратных средств, усилия которых сосредоточены в рамках OMG, видят перспективы развития современных информационных технологий и основу своего коммерческого успеха в широком продвижении на рынок инструментальных средств, поддерживающих объектные технологии. Говоря об объектных технологиях, разработчики из OMG имеют в виду, прежде всего, совокупность технологических решений системы проектирования CORBA и языка UML. С этой точки зрения языку UML отводится роль базового средства для описания и документирования различных объектных компонентов CORBA.

6. Способствовать распространению объектных технологий и соответствующих понятий ООАП. Эта задача тесно связана с пре-

дыдущей и имеет с ней много общего. Если исключить из рассмотрения рекламные заявления разработчиков об исключительной гибкости и мощности языка UML и попытаться составить объективную картину возможностей этого языка, то можно прийти к следующему заключению. Усилия достаточно большой группы разработчиков были направлены на интеграцию в рамках языка UML многих известных технологий визуального моделирования, которые успешно зарекомендовали себя на практике. Хотя это привело к усложнению языка UML по сравнению с известными нотациями структурного системного анализа, платой за сложность являются действительно высокая гибкость и изобразительные возможности уже первых версий языка UML. В свою очередь, использование языка UML для решения всевозможных практических задач будет только способствовать его дальнейшему совершенствованию, а значит, и дальнейшему развитию объектных технологий и практики ООАП.

7. Интегрировать новейшие и наилучшие практические достижения ООАП. Язык UML непрерывно совершенствуется разработчиками, и основой этой работы является его дальнейшая интеграция с современными модельными технологиями. При этом различные методы системного моделирования получают свое прикладное осмысление в рамках ООАП. В последующем эти методы могут быть включены в состав языка UML в форме дополнительных базовых понятий, наиболее адекватно и полно отражающих наилучшие практические достижения ООАП.

Чтобы решить перечисленные задачи, язык UML претерпел за свою недолгую историю определенную эволюцию. В результате описание самого языка UML стало нетривиальным, поскольку семантика базовых понятий включает в себя целый ряд перекрестных связей с другими понятиями и конструкциями языка. В связи с этим так называемое линейное, или последовательное, рассмотрение основных конструкций языка UML стало практически невозможным, поскольку одни и те же понятия могут использоваться при построении различных диаграмм или представлений. В то же время каждое из представлений модели обладает собственными семантическими особенностями, которые накладывают отпечаток на семантику базовых понятий языка в целом.

Говоря о сложности описания языка UML, следует отметить присущую всем формальным языкам сложность их строгого задания, которая вытекает из необходимости в той или иной степени использовать естественный язык для спецификации базовых примитивов. В этом случае естественный язык выступает в роли метаязыка, т.е. языка для описания формального языка. Поскольку естественный язык не является формальным, то и его применение для описания формальных языков в той или иной степени страдает неточностью Хотя в задачи языка UML не входит анализ соответствующих логико-лингвистических деталей, эти особенности отразились на структуре описания языка UML, в частности, сделали стиль описания всех его основных понятий полуформальным.

14.2. Общая структура языка UML

С самой общей точки зрения описание языка UML состоит из двух взаимодействующих частей: семантики и нотации.

Семантика языка UML представляет собой некоторую метамодель, которая определяет абстрактный синтаксис и семантику понятий объектного моделирования на языке UML.

Нотация языка UML представляет собой графическую нотацию для визуального представления семантики языка UML.

Абстрактный синтаксис и семантика языка UML описываются с использованием некоторого подмножества нотации UML. В дополнение к этому нотация UML описывает соответствие графической нотации базовым понятиям семантики. Таким образом, с функциональной точки зрения эти две части дополняют друг друга. При этом семантика языка UML описывается на основе некоторой метамодели, имеющей три отдельных представления: абстрактный синтаксис, правила корректного построения выражений и семантику. Рассмотрение семантики языка UML предполагает некоторый полуформальный стиль изложения, который объединяет естественный и формальный языки для представления базовых понятий и правил их расширения.

Семантика определяется для двух видов объектных моделей: структурных моделей и моделей поведения. Структурные модели, называемые также статическими, описывают структуру сущностей или компонентов некоторой системы, включая их классы, интерфейсы, атрибуты и отношения. Модели поведения, называемые иногда динамическими, описывают поведение или функционирование объектов системы, включая их методы, взаимодействие и сотрудничество между ними, а также процесс изменения состояний отдельных компонентов и системы в целом.

Для решения столь широкого диапазона задач моделирования разработана достаточно полная семантика для всех компонентов графической нотации. Требования семантики языка UML конкретизируются при построении отдельных видов диаграмм. Нотация языка UML включает в себя описание отдельных семантических элементов, которые могут применяться при построении диаграмм.

Формальное описание самого языка UML основывается на некоторой общей иерархической структуре модельных представлений, имеющей четыре уровня: метаметамодель; метамодель; модель; объекты пользователя.

Уровень *метаметамодели* образует исходную основу для всех метамодельных представлений. Главное предназначение этого уровня состоит в том, чтобы определить язык для спецификации метамодели. Метаметамодель определяет модель языка UML на самом высоком уровне абстракции и является наиболее компактным ее описанием. С другой стороны, метаметамодель может специфицировать несколько метамоделей, чем достигается потенциальная гибкость включения дополнительных понятий. Примерами понятий этого уровня служат метакласс, метаатрибут, метаоперация.

Следует отметить, что семантика метаметамодели не входит в описание языка UML. С одной стороны, это делает язык UML более простым для изучения, поскольку не требуются знания общей теории формальных языков и формальной логики. С другой стороны, наличие метаметамодели придает языку UML статус научности, который необходим ему для того, чтобы быть непротиворечивым формальным языком. Если эти особенности могут представляться мало интересными для многих программистов, то разработчики инструментальных средств никак не могут их игнорировать.

Метамодель является экземпляром или конкретизацией метаметамодели. Главная задача этого уровня — определить язык для спецификации моделей. Данный уровень является более конструктивным, чем предыдущий, поскольку обладает более развитой семантикой базовых понятий. Все основные понятия языка UML — это понятия уровня метамодели. Примеры таких понятий: класс, атрибут, операция, компонент, ассоциация и многие другие.

Модель в контексте языка UML является экземпляром метамодели в том смысле, что любая конкретная модель системы должна использовать только понятия метамодели, конкретизировав их применительно к своей ситуации. Этот уровень служит для описания информации о конкретной предметной области. Однако если для построения модели используются понятия языка UML, то необходима полная согласованность понятий уровня модели с базовыми понятиями языка UML уровня метамодели. Примерами понятий уровня модели могут служить имена полей проектируемой базы данных — имя и фамилия сотрудника, возраст, должность, адрес, телефон. При этом данные понятия используются лишь как имена соответствующих информационных атрибутов.

Конкретизация понятий модели происходит на уровне *объектов пользователя*. В настоящем контексте объект является экземпляром модели, поскольку содержит конкретную информацию относительно того, чему в действительности соответствуют те или иные понятия модели. Примером объекта может служить следующая запись в проектируемой базе данных: «Илья Петров, 18 лет, программист, ул. Пионерская, д. 5, к. 1, кв. 23, тел. 123-45-67».

Описание семантики языка UML предполагает рассмотрение базовых понятий только уровня метамодели, который представляет собой лишь пример или частный случай уровня метаметамодели. Метамодель UML является по своей сути скорее логической моделью, чем физической или моделью реализации. Особенность логической модели заключается в том, что она концентрирует внимание на декларативной или концептуальной семантике, опуская детали конкретной физической реализации моделей. При этом отдельные реализации, использующие данную логическую метамодель, должны быть согласованы с ее семантикой, а также поддерживать возможности импорта и экспорта отдельных логических моделей.

В то же время логическая метамодель может быть реализована различными способами для обеспечения требуемого уровня производительности и надежности соответствующих инструментальных средств. В этом заключается недостаток логической модели, которая не содержит на уровне семантики требований, обязательных для ее эффективной последующей реализации. Однако согласованность метамодели с конкретными моделями реализации является обязательной для всех разработчиков программных средств, обеспечивающих поддержку языка UML.

14.3. Общие сведения о пакетах в языке UML

Метамодель языка UML имеет довольно сложную структуру, которая включает в себя около 90 метаклассов, более 100 метаассоциаций и почти 50 стереотипов, число которых возрастает с появлением новых версий языка. Чтобы справиться с этой сложностью языка UML, все его элементы организованы в логические пакеты. Поэтому рассмотрение языка UML на метамодельном уровне заключается в описании трех его наиболее общих логических блоков, или пакетов: «Основные элементы», «Элементы поведения» и «Общие механизмы».

Говоря о пакетах в контексте общего описания языка, мы, по сути дела, рассматриваем графическую нотацию языка UML. Дело в том, что для описания языка UML используются средства самого языка, и одним из таких средств является пакет. В общем случае пакет служит для группировки элементов модели. При этом сами элементы модели, которыми могут быть произвольные сущности, отнесенные к одному пакету, выступают в роли единого целого.

Пакеты, как и другие элементы модели, могут быть вложены в другие пакеты. Важной особенностью языка UML является тот факт, что все виды элементов модели UML могут быть организованы в пакеты.

Пакет — основной способ организации элементов модели в языке UML. Каждый пакет владеет всеми своими элементами, т.е. теми элементами, которые включены в него. Про соответствующие элементы пакета говорят, что они принадлежат пакету или входят в него. При этом каждый элемент может принадлежать только одному пакету. В свою очередь, одни пакеты могут быть вложены в другие пакеты. В этом случае первые называются подпакетами, поскольку все элементы подпакета принадлежат более общему пакету. Тем самым для элементов модели задается отношение вложенности пакетов, которое представляет собой иерархию.

Отношение пакет—подпакет наиболее естественно ассоциировать с более общим отношением множество—подмножество. Поскольку пакет можно рассматривать в качестве частного случая множества, такая интерпретация помогает использовать графические средства для представления соответствующих отношений между пакетами.

Известно, что для графического представления иерархий могут использоваться графы специального вида, которые называются деревьями. Однако в языке UML эти графические обозначения настолько модифицированы, что соответствующие ассоциации с общетеоретическими понятиями могут представлять определенную трудность для начинающих разработчиков. Тем не менее, очень важно уметь ассоциировать специальные конструкции языка UML с соответствующими понятиями теории множеств и системного моделирования, что, в некотором смысле, формирует стиль мышления системного аналитика. В противном случае не исключены досадные ошибки не только на начальном этапе концептуализации предметной области, но и в процессе построения различных представлений систем.

В языке UML для визуализации пакетов разработана специальная символика, или графическая нотация. Для графического изображения пакетов на диаграммах применяется специальный графический символ — большой прямоугольник с небольшим прямоугольником, присоединенным к левой части верхней стороны большого (рис. 14.2). Визуально символ пакета напоминает пиктограмму папки в популярном графическом интерфейсе. Внутри большого прямоугольника может записываться информация, относящаяся к данному пакету. Если такой информации нет, то внутри большого прямоугольника записывается имя пакета, которое должно быть уникальным в пределах рассматриваемой модели (см. рис. 14.2, *а*). При наличии такой информации имя пакета записывается в верхнем маленьком прямоугольнике (см. рис. 14.2, *б*).

Говоря об имени пакета, следует остановиться на общем соглашении об именах в языке UML. В данном случае именем пакета может быть строка (или несколько строк) текста, содер-

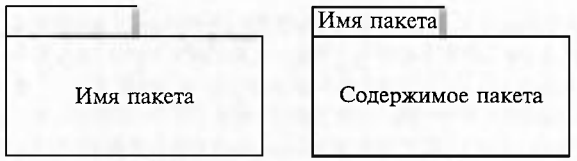


Рис. 14.2. Графическое изображение пакетов в языке UML:
а — при отсутствии информации, относящейся к данному пакету; *б* — при наличии такой информации

жащая любое число букв, цифр и некоторых специальных знаков. С целью удобства спецификации пакетов принято в качестве их имен использовать одно или несколько существительных, например «контроллер», «графический интерфейс», «форма ввода данных».

Перед именем пакета может помещаться строка текста, содержащая некоторое ключевое слово. Подобными ключевыми словами являются заранее определенные в языке UML слова, которые получили название стереотипов. Такими стереотипами для пакетов являются слова facade (фасад), framework (каркас), stub (заглушка) и topLevel (верхний уровень). В качестве содержимого пакета могут выступать имена его отдельных элементов и их свойства.

Сами по себе пакеты могут найти ограниченное применение, поскольку содержат лишь информацию о входящих в их состав элементах модели. Не менее важно представить графически отношения, которые могут иметь место между отдельными пакетами. Как и в теории графов, для визуализации отношений в языке UML применяются отрезки линий, внешний вид которых имеет смысловое содержание.

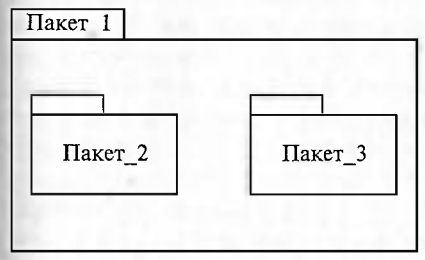


Рис. 14.3. Графическое изображение вложенности пакетов друг в друга

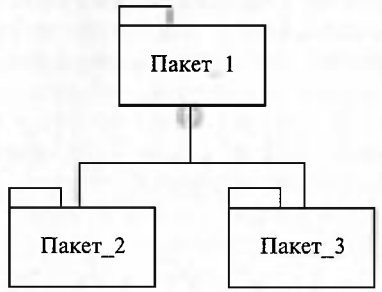


Рис. 14.4. Графическое изображение вложенности пакетов друг в друга с помощью явной визуализации отношения включения

Одним из типов отношений между пакетами является отношение вложенности, или включения, пакетов друг в друга. С одной стороны, в языке UML это отношение может быть изображено без использования линий, т.е. простым размещением одного пакета-прямоугольника внутри другого пакета-прямоугольника (рис. 14.3). Так, в данном случае пакет с именем Пакет_1 содержит в себе два подпакета: Пакет_2 и Пакет_3.

С другой стороны, это же отношение может быть изображено с помощью отрезков линий аналогично графическому представлению дерева (рис. 14.4). В этом случае наиболее общий пакет (метапакет, или контейнер) изображается в верхней части рисунка, а его подпакеты — уровнем ниже. Метапакет соединяется с подпакетами сплошной линией, на конце которой, примыкающей к метапакету, изображается специальный символ — знак «плюс» в кружочке. Этот символ означает, что подпакеты являются «собственностью» или частью контейнера и кроме них контейнер не содержит никаких других подпакетов.

На графических диаграммах между пакетами могут указываться и другие типы отношений.

14.4. Основные пакеты метамодели языка UML

Как уже отмечалось в подразд. 14.3, основой представления языка UML на метамодельном уровне является описание трех его логических блоков, или пакетов: «Основные элементы», «Элементы поведения» и «Общие механизмы» (рис. 14.5).

Эти пакеты, в свою очередь, подразделяются на отдельные подпакеты. Например, пакет «Основные элементы» состоит из подпакетов «Элементы ядра», «Вспомогательные элементы», «Механизмы расширения» и «Типы данных» (рис. 14.6). При этом пакет «Элементы ядра» описывает базовые понятия и принципы включения в структуру метамодели основных понятий языка, таких как метаклассы, метаассоциации и метаатрибуты. Пакет

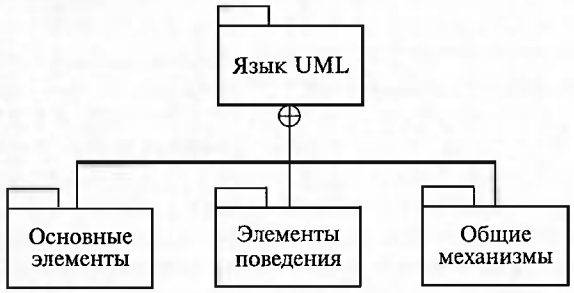


Рис. 14.5. Основные пакеты метамодели языка UML

Рис. 14.6. Подпакеты пакета «Основные элементы» языка UML

«Вспомогательные элементы» определяет дополнительные конструкции, которые расширяют базовые элементы для описания зависимостей, шаблонов, физических структур и элементов представлений. Пакет «Механизмы расширения» задает правила уточнения и расширения семантики базовых элементов моделей. Пакет «Типы данных» определяет основные структуры данных для языка UML.

Пакет «Основные элементы». Ниже дается краткая характеристика элементов каждого из перечисленных подпакетов, входящих в состав пакета «Основные элементы».

Пакет «Элементы ядра» является наиболее фундаментальным из всех подпакетов, которые входят в пакет «Основные элементы» языка UML. Этот пакет определяет основные абстрактные и конкретные компоненты, необходимые для разработки объектных моделей. При этом абстрактные компоненты метамодели не имеют экземпляров или примеров и используются исключительно для уточнения других компонентов модели. Конкретные компоненты метамодели имеют экземпляры и отражают особенности представления лиц, которые разрабатывают объектные модели.

Следует отметить присущую развитым языкам представления знаний в целом и языку UML в частности неоднозначность выразительных возможностей. Речь идет о том, что одна и та же моделируемая сущность или система может быть представлена средствами языка UML по-разному. При этом разные разработчики могут построить объектные модели одной и той же системы, существенно отличающиеся не только формой своего представления, но и составом используемых в модели компонентов.

Пакет «Элементы ядра» специфицирует базовые конструкции, требуемые для описания исходной метамодели, и определяет архитектурный «скелет» для присоединения дополнительных конструкций языка, таких как метаклассы, метаассоциации и метаатрибуты. Хотя пакет «Элементы ядра» содержит семантику, дос-

таточную для определения всей оставшейся части языка UML, он не является метаметамоделью UML.

В этот пакет входят основные метаклассы языка UML: класс (Class); атрибут (Attribute); ассоциация (Association); ассоциация-класс (AssociationClass); конец ассоциации (AssociationEnd); свойство поведения (BehavioralFeature); классификатор (Classifier); ограничение (Constraint); тип данных (DataType); зависимость (Dependency); элемент (Element); право на элемент (ElementOwnership); свойство (Feature); обобщение (Generalization); элемент отношения обобщения (GeneralizableElement); интерфейс (Interface); метод (Method); элемент модели (ModelElement); пространство имен (Namespace); операция (Operation), параметр (Parameter); структурное свойство (StructuralFeature); правила построения выражений (Well-formedness rules).

Пакет «Вспомогательные элементы» специфицирует дополнительные конструкции языка UML, которые расширяют пакет «Элементы ядра». Вспомогательные элементы обеспечивают понятийный базис для зависимостей, шаблонов, физических структур и элементов представлений.

В этот пакет входят следующие метаклассы: связывание (Binding); комментарий (Comment); компонент (Component); узел (Node); презентация (Presentation); уточнение (Refinement); цепочка зависимостей (Trace); потребление (Usage); элемент представления (ViewElement); зависимость (Dependency); элемент модели (ModelElement); правила построения выражений (Well-formedness rules). Три последних метакласса взяты из пакета «Элементы ядра» и используются для спецификации остальных.

Хотя этот пакет имел самостоятельное значение в начальных версиях языка UML, в проектах последней версии его элементы объединились с пакетом «Элементы ядра». Причиной этого послужило требование строгого вхождения каждого элемента в один пакет.

Пакет «Механизмы расширения» специфицирует порядок включения в модель элементов с уточненной семантикой, а также модификацию отдельных компонентов языка UML для более точного отражения специфики моделируемых систем. Механизм расширения определяет семантику для стереотипов, ограничений и помеченных значений. Хотя язык UML обладает богатым множеством понятий и нотаций для моделирования типичных программных систем, реально разработчик может столкнуться с необходимостью включить в модель дополнительные свойства или нотации, которые не определены явно в языке UML. При этом разработчики часто сталкиваются с необходимостью включения в модель графической информации, такой, например, как дополнительные значки и украшения.

Для этой цели в языке UML предусмотрены три механизма расширения, которые могут использоваться совместно или раз-

дельно для определения новых элементов модели с отличающимися от специфицированных в метамодели языка UML элементов семантикой, нотацией и свойствами. Такими механизмами являются ограничение (Constraint), стереотип (Stereotype) и помеченное значение (Tagged Value).

Механизмы расширения языка UML предназначены для выполнения следующих задач:

уточнение существующих модельных элементов при разработке моделей на языке UML;

определение стандартных компонентов в спецификации самого языка UML, которые либо не являются достаточно интересными, либо сложны для непосредственного определения в качестве элементов метамодели UML;

определение таких расширений языка UML, которые зависят от специфики моделируемого процесса или от языка реализации программного кода;

присоединение произвольной семантической или несемантической информации к элементам модели.

Наиболее важные из встроенных механизмов расширения основываются на понятии «стереотип». Стереотипы обеспечивают некоторый способ классификации модельных элементов на уровне объектной модели и возможность добавления в язык UML «виртуальных» метаклассов с новыми атрибутами и семантикой. Другие встроенные механизмы расширения основываются на понятии «список свойств», который содержит помеченные значения и ограничения.

Эти механизмы обеспечивают пользователю возможность включения дополнительных свойств и семантики непосредственно в отдельные элементы модели.

Пакет «Типы данных» специфицирует различные типы данных, которые могут использоваться в языке UML. Этот пакет имеет более простые по сравнению с другими пакетами внутреннюю структуру и описание, поскольку предполагается, что семантика соответствующих понятий хорошо известна.

В метамодели UML типы данных используются для объявления типов атрибутов классов. Они записываются в форме строк текста на диаграммах и не имеют отдельного значка «тип данных». Благодаря этому происходит уменьшение размеров диаграмм без потери информации. Однако каждая из одинаковых записей для некоторого типа данных должна соответствовать одному и тому же типу данных в модели. При этом типы данных, используемые в описании языка UML, могут отличаться от типов данных, которые определяет разработчик для своей модели на языке UML. Типы данных в последнем случае будут являться частным случаем или экземплярами метакласса «типы данных», который определен в метамодели.

При задании типа данных наиболее часто применяется неформальная конструкция, которая называется перечислением. Речь идет о множестве допустимых значений атрибута, которое наделяется некоторым отношением порядка. При этом упорядоченность значений либо указывается явно заданием первого и последнего элементов списка, либо следует неявно в случае простого типа данных, как, например, для множества натуральных чисел. В пакете «Типы данных» определены способы спецификации перечислений для корректного задания допустимых значений атрибутов.

Для определения различных типов данных в языке UML используются как простые конструкции: целое число (Integer), строка (String), имя (Name), булев (Boolean), время (Time), кратность (Multiplicity), тип видимости (VisibilityKind), диапазон кратности (MultiplicityRange), так и более сложные: выражение (Expression), булево выражение (BooleanExpression), тип агрегирования (AggregationKind), тип изменения (ChangeableKind), геометрия (Geometry), отображение (Mapping), выражение-процедура (ProcedureExpression), тип псевдосостояния (PseudostateKind), выражение времени (TimeExpression), непрерываемый (Uninterpreted).

Пакет «Элементы поведения». Этот пакет является самостоятельным компонентом языка UML и, как следует из его названия, специфицирует динамику поведения в нотации UML. Пакет «Элементы поведения» состоит из четырех подпакетов: «Общее поведение», «Кооперации», «Варианты использования» и «Автоматы» (рис. 14.7). Ниже дается краткая характеристика каждого из этих подпакетов.

Пакет Общее поведение является наиболее фундаментальным из всех подпакетов и определяет базовые понятия ядра, необходимые для всех элементов поведения. В этом пакете специфицирована семантика для динамических элементов, которые включены в другие подпакеты элементов поведения. В пакет «Общее поведение» входят следующие элементы: объект (Object), действие

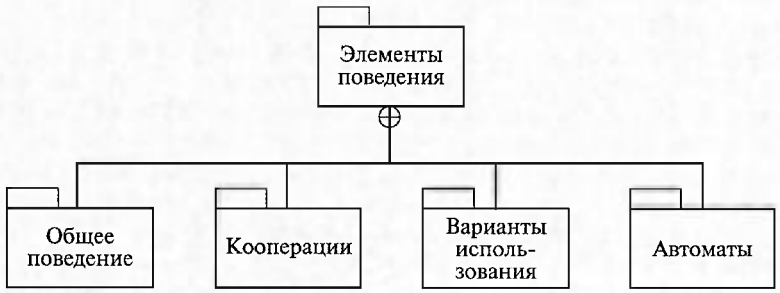


Рис. 14.7. Подпакеты пакета «Элементы поведения» языка UML

(Action), последовательность действий (ActionSequence), аргумент (Argument), экземпляр (Instance), исключение (Exception), связь (Link), сигнал (Signal), значение данных (DataValue), связь атрибутов (AttributeLink), действие вызова (CallAction), действие создания (CreateAction), действие уничтожения (DestroyAction).

Наиболее важным из перечисленных элементов является объект, под которым в языке UML понимается отдельный экземпляр или пример класса, структура и поведение которого полностью определяются порождающим этот объект классом. Предполагается, что все без исключения объекты, порожденные одним и тем же классом, имеют совершенно одинаковые структуру и поведение, хотя каждый них может иметь свое собственное множество связей атрибутов. При этом каждая связь атрибута относится к некоторому экземпляру, обычно к значению данных. Это множество может быть модифицировано согласно спецификации отдельного атрибута в описании класса.

В языке UML под поведением понимается не только процесс изменения атрибутов объектов в результате выполнения операций над их значениями, но и такие процедуры, как создание и уничтожение самих объектов. При этом динамика взаимодействия объектов, которая определяет их поведение, описывается с помощью специальных понятий, таких как «сигналы» и «действия».

Пакет «Кооперации» специфицирует контекст поведения элементов модели при их использовании для выполнения отдельной задачи. В нем задается семантика понятий, которые необходимы для ответа на вопрос: «Как различные элементы модели взаимодействуют между собой с точки зрения структуры?». Этот пакет использует конструкции, определенные в пакетах «Основные элементы» и «Общее поведение».

В частности, в пакет «Кооперации» входят следующие элементы: кооперация (Collaboration), взаимодействие (Interaction), сообщение (Message), роль ассоциации (AssociationRole), роль классификатора (ClassifierRole), роль конца ассоциации (AssociationEndRole). Как можно догадаться из названия пакета, его элементы непосредственно используются при построении диаграмм кооперации. Понятие «кооперация» имеет важное значение для представления взаимодействия элементов модели с точки зрения классификаторов и ассоциаций.

Пакет «Варианты использования» специфицирует поведение модели при включении в нее специальных элементов, которые в языке UML называются актерами и вариантами использования. Эти понятия служат для определения функциональности моделируемой сущности, такой как система. Особенность элементов этого пакета состоит в том, что они используются для первоначального определения поведения сущности без спецификации ее внутренней структуры.

Объединение в языке UML средств концептуализации исходных требований к проектируемой системе и структуризации ее внутренних компонентов с достаточно богатой семантикой применяемых для этого элементов имеет важное значение для построения адекватных моделей сложных систем. Действительно, ограниченность традиционных моделей состоит в том, что они не позволяют одновременно описывать статические или структурные свойства системы и динамику ее поведения. Попытки совместного решения данных проблем сталкиваются с отсутствием единой символики для обозначения близких по смыслу системных понятий. Язык UML удачно выделяет базовые понятия, которые необходимы при построении таких моделей. Более того, если этих понятий окажется недостаточно для разработки какого-то конкретного проекта, то сам разработчик может расширить базовые понятия и даже включить в модель собственные конструкции, согласованные с метамоделью языка UML.

В пакет «Варианты использования» кроме уже упомянутых элементов актер (Actor) и вариант использования (UseCase) входят следующие элементы: расширение (Extension), точка расширения (ExtensionPoint), включение (Include) и экземпляр варианта использования (UseCaseInstance).

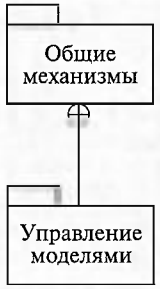
Пакет «Автоматы» специфицирует поведение элементов при построении моделей с использованием систем переходов для конечного множества состояний. В нем определено множество понятий, которые необходимы для представления поведения модели в виде дискретного пространства с конечным числом состояний и переходов.

Формализм автомата, который используется в языке UML, отличается от формализма теории автоматов своей объектной ориентацией. Автоматы являются основным средством моделирования поведения различных элементов языка UML. Например, автоматы могут использоваться для моделирования поведения индивидуальных сущностей, таких как экземпляры классов, а также для спецификации взаимодействий между сущностями, таких как кооперации. Формализм автоматов дополнительно обеспечивает семантический базис для графов деятельности, которые являются частным случаем автомата.

В пакет «Автоматы» входят следующие элементы: состояние (State), переход (Transition), событие (Event), автомат (StateMachine), простое состояние (SimpleState), составное состояние (CompositeState), псевдосостояние (PseudoState), конечное состояние (FinalState) и некоторые другие.

Одним из ключевых при моделировании динамических свойств систем является понятие «состояние». Под состоянием в языке UML понимается абстрактный метакласс, используемый для моделирования ситуации или процесса, в ходе которого имеет место

Рис. 14.8. Состав пакета «Общие механизмы»



(обычно неявное) выполнение некоторого инвариантного условия. Примером такого условия может быть состояние ожидания объектом выполнения некоторого внешнего события, например запроса или передачи управления. С другой стороны, состояние может использоваться для моделирования динамических условий, таких как процесс выполнения некоторой операции. В этом случае момент начала выполнения операции является переходом объекта в соответствующее состояние.

Пакет «Общие механизмы». В этом пакете определены общие механизмы, которые применимы ко всем моделям UML. Пакет состоит из единственного подпакета «Управление моделями» (рис. 14.8), который служит для спецификации способов организации элементов в модели, пакеты и подсистемы. Кратко рассмотрим основные особенности данного подпакета.

Пакет «Управление моделями» (Model Management) специфицирует базовые элементы языка UML, которые необходимы для формирования всех модельных представлений. Именно в нем определяется семантика модели (Model), пакета (Package) и подсистемы (Subsystem). Эти элементы служат своеобразными контейнерами для группировки других элементов модели.

Пакет является метаклассом в языке UML и предназначен, как отмечалось выше, для организации других элементов модели, таких как другие пакеты, классификаторы и ассоциации. Пакет может также содержать ограничения и зависимости между элементами модели в самом пакете. Предполагается, что каждый элемент пакета имеет видимость только внутри данного пакета. Это означает, что за пределами пакета никакой его элемент не может быть использован, если нет дополнительных указаний на импорт или доступ к отдельным элементам пакета. Пакеты со всем своим содержимым определены в некотором пространстве имен, которое задает единственность использования имен всех элементов модели. Другими словами, имя каждого элемента модели должно быть единственным, или уникальным, в некотором пространстве имен, которое, являясь само элементом модели, может быть вложено в более общее пространство имен.

Модель является подклассом пакета и представляет собой абстракцию физической системы, которая предназначена для вполне определенной цели. Именно эта цель предопределяет те компоненты, которые должны быть включены в модель, и те, рассмотрение которых не является обязательным. Другими словами, модель отражает релевантные аспекты физической системы, оказывающие непосредственное влияние на достижение поставлен-

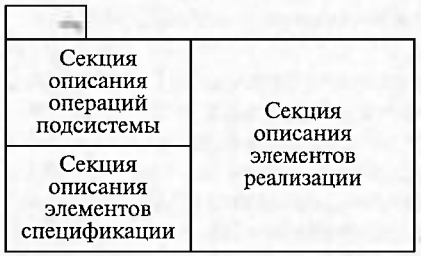


Рис. 14.9. Графическое изображение подсистемы в языке UML

ной цели. В прикладных задачах цель обычно задается в форме исходных требований к системе, которые, в свою очередь, в языке UML записываются в виде вариантов использования системы.

В языке UML для одной и той же физической системы могут быть определены различные модели, каждая из которых специфицирует систему с различных точек зрения. Примерами таких моделей являются логическая модель, модель проектирования, модель вариантов использования и др. При этом каждая такая модель имеет свою собственную точку зрения на физическую систему и свой собственный уровень абстракции. Модели, как и пакеты, могут быть вложенными друг в друга. Со своей стороны, пакет может включать в себя несколько различных моделей одной и той же системы, и в этом состоит один из важнейших механизмов разработки моделей на языке UML.

Подсистема есть просто группировка элементов модели, которые специфицируют некоторое простейшее поведение физической системы. В метамодели UML подсистема является подклассом как пакета, так и классификатора. Элементы подсистемы делятся на две части: спецификацию поведения и его реализацию.

Для графического представления подсистемы применяется специальное обозначение — прямоугольник, как в случае пакета, но дополнительно разделенный на три секции (рис. 14.9). При этом в верхнем маленьком прямоугольнике изображается символ, по своей форме напоминающий вилку и указывающий на подсистему.

Имя подсистемы вместе с необязательным ключевым словом или стереотипом записывается внутри большого прямоугольника. Однако при наличии строк текста внутри большого прямоугольника имя подсистемы может быть записано рядом с обозначением «вилки».

Операции подсистемы записываются в левой верхней секции, ниже указываются элементы спецификации, а справа от вертикальной линии — элементы реализации. При этом последние две секции помечаются соответствующими метками: «Элементы спецификации» и «Элементы реализации». Секция операций никак не помечается.

Если в подсистеме отсутствуют те или иные секции, то они совсем не отображаются на схеме.

14.5. Специфика описания метамодели языка UML

Метамодель языка UML описывается определенным полуформальным языком с использованием трех видов представлений: абстрактного синтаксиса; правил построения выражений; семантики.

Абстрактный синтаксис представляет собой модель для описания некоторой части языка UML, предназначенной для построения диаграмм классов на основе описаний систем на естественном языке. Возможности абстрактного синтаксиса в языке UML довольно ограничены и имеют отношение только к интерпретации обозначений отдельных компонентов диаграмм, связей между компонентами и допустимых дополнительных обозначений. К элементам абстрактного синтаксиса относятся некоторые ключевые слова и значения отдельных атрибутов базовых понятий уровня метамодели, которые имеют фиксированное обозначение в виде текста на естественном языке.

Правила построения выражений используются для задания дополнительных ограничений или свойств, которыми должны обладать те или иные компоненты модели. Поскольку исходным для объектно-ориентированного проектирования является понятие «класс», общими свойствами класса должны обладать все экземпляры, которые в этом смысле должны быть инвариантны по отношению друг к другу. Для задания этих инвариантных свойств классов и отношений необходимо использовать специальные выражения некоторого формального языка, получившего в рамках UML название языка объектных ограничений (OCL — Object Constraint Language). Хотя язык OCL и использует естественный язык для формулировки правил построения выражений, особенности его применения являются темой самостоятельного обсуждения.

Семантика языка UML описывается в основном на естественном языке, но может включать в себя некоторые дополнительные обозначения, вытекающие из связей определяемых понятий с другими понятиями. Семантика понятий раскрывает их смысл или содержание. Сложность описания семантики языка UML заключается именно в метамодельном уровне представлений его основных конструкций. С одной стороны, понятия языка UML имеют абстрактный характер (ассоциация, композиция, агрегация, сотрудничество, состояние). С другой стороны, каждое из этих понятий допускает свою конкретизацию на уровне модели (сотрудник, отдел, должность, стаж).

Сложность описания семантики языка UML вытекает из этой двойственности понятий. При таком описании необходимо придерживаться традиционных правил изложения. Иллюстрация абстрактных понятий на примере конкретных свойств и отноше-

ний, а также их значений позволяет акцентировать внимание на общих инвариантах этих понятий, что совершенно необходимо для понимания их семантики.

Таким образом, метамодель языка UML может рассматриваться как комбинация графической нотации (специальных обозначений), некоторого формального языка и естественного языка. При этом следует иметь в виду, что существует некоторый теоретический предел, который ограничивает описание метамодели средствами самой метамодели. Именно в подобных случаях используется естественный язык, обладающий наибольшими выразительными возможностями.

Хотя сам термин «естественный язык» далеко не однозначен и порождает целый ряд дополнительных вопросов, будем его трактовать как форму обычного текста на русском и, возможно, английском языках. Как бы ни хотелось некоторым из отечественных разработчиков полностью избежать использования английского языка при описании языка UML, это не удастся. Тем не менее если исключить написание стандартных элементов и некоторых ключевых слов, то во всех остальных случаях под естественным языком можно понимать русский без специальных оговорок.

Для придания формального характера моделям UML использование естественного языка должно строго соответствовать определенным правилам. Например, описание семантики языка UML может включать в себя фразы типа «Сущность А обладает способностью» или «Сущность Б есть сущность В». В каждом из этих случаев мы будем воспринимать смысл фраз, руководствуясь традиционным пониманием предложений русского языка. Однако этого может оказаться недостаточно для более формального представления знаний о рассматриваемых сущностях. Тогда необходимо дополнительно специфицировать семантику этих простых фраз, для чего рекомендуется использовать следующие правила:

явно указывать в тексте экземпляр некоторого метакласса. Речь идет о том, что в естественной речи мы часто опускаем слово «пример» или «экземпляр», говоря просто «класс». Так, фразу «Атрибут “возраст” класса “сотрудник” имеет значение 30 лет» следует записать более точно: «Атрибут “возраст” экземпляра класса “сотрудник” имеет значение 30 лет»;

в каждый момент времени использовать только то значение слова, которое приписано имени соответствующей конструкции языка UML. Все дополнительные особенности семантики должны быть указаны явным образом без каких бы то ни было неявных предположений;

термины языка UML должны включать в себя только один из допустимых префиксов, таких как «под», «супер» или «мета». При этом сам термин с префиксом записывается одним словом.

В дополнение к этому должны выполняться следующие правила выделения текста:

если используются ссылки на конструкции языка UML, а не на их представления в метамодели, то следует применять обычный текст без какого бы то ни было выделения;

имя метакласса является элементом нотации языка UML и представляет собой существительное и, возможно, присоединенное к нему прилагательное. В этом случае имя метакласса на английском языке записывается одним словом с выделением каждой составной части имени заглавной буквой (например, ModelElement, StructuralFeature);

имена метаассоциаций и ассоциаций классов записываются аналогичным образом (например, ElementReference);

имена других элементов языка UML также записываются одним словом, но должны начинаться со строчной буквы (например, ownedElement, allContents);

имена метаатрибутов, которые принимают булевы значения, всегда начинаются с префикса is (например, isAbstract);

перечисляемые типы должны всегда заканчиваться словом «Kind» (например, AggregationKind);

при ссылках в тексте на метаклассы, метаассоциации, метаатрибуты должны всегда использоваться в точности те их имена, которые указаны в модели;

имена стандартных обозначений (стереотипов) заключаются в кавычки и начинаются со строчной буквы (например, «type»);

при описании семантики языка UML все имена его стандартных элементов (метаклассов, метаассоциаций, метаатрибутов) допускается записывать на русском языке с дополнительным указанием оригинального имени на английском. Если имена стандартных элементов состоят из нескольких слов, то эти слова записывают раздельно (например, «класс ассоциации», «элемент модели», «пространство имен»);

при разработке конкретных моделей систем в форме диаграмм языка UML целесообразно применять оригинальные англоязычные термины, придерживаясь описанных выше правил (кроме, возможно, пояснительного текста на русском). Причина этой рекомендации вполне очевидна: последующая инструментальная реализация модели может оказаться невозможной, если не следовать оригинальным правилам записи текста при использовании языка UML.

Приведенные дополнительные рекомендации не противоречат оригинальным правилам языка UML, а только уточняют рамки использования естественного языка при построении моделей и описании самого языка. Поскольку описание семантики любого формального языка связано с проблемой его интерпретации, полностью обойтись без естественного языка не представляется воз-

можным. Если вопросы использования оригинальных терминов при построении логических и физических моделей не вызывают сомнений у большинства программистов, то процесс построения концептуальных моделей сложных систем формализован в меньшей степени. Именно по этой причине исходные требования к системе формулируются на естественном для разработчиков языке (в нашем случае на русском).

В рамках языка UML все представления о модели сложной системы фиксируются в виде специальных графических конструкций, называемых диаграммами. В терминах языка UML определены следующие виды диаграмм:

диаграммы вариантов использования (use case diagrams);

диаграммs классов (class diagrams);

диаграммы поведения (behavior diagrams), подразделяют на диаграммы состояний (statechart diagrams), диаграммы деятельности (activity diagrams), а также диаграммы взаимодействия (interaction diagrams), которые, в свою очередь, подразделяются на диаграммы последовательностей (sequence diagrams) и диаграммы кооперации (collaboration diagrams);

диаграммы реализации (implementation diagrams), подразделяющиеся на диаграммы компонентов (component diagrams) и диаграммы развертывания (deployment diagrams).

Из перечисленных выше диаграмм некоторые служат для обозначения двух и более подвидов диаграмм. При этом в качестве самостоятельных представлений в языке UML используются следующие диаграммы: вариантов использования; классов; состояний; деятельности; последовательностей; кооперации; компонентов; развертывания.

Перечень этих диаграмм и их названия являются каноническими в том смысле, что представляют собой неотъемлемую часть графической нотации языка UML. Более того, процесс ООАП неразрывно связан с процессом построения указанных диаграмм. При этом совокупность построенных таким образом диаграмм является самодостаточной в том смысле, что в них содержится вся информация, которая необходима для реализации проекта сложной системы.

Каждая из диаграмм детализирует и конкретизирует различные представления о модели сложной системы в терминах языка UML. При этом диаграмма вариантов использования представляет собой наиболее общую концептуальную модель сложной системы, которая является исходной для построения всех остальных диаграмм. Диаграмма классов является по своей сути логической моделью, отражающей статические аспекты структурного построения сложной системы.

Диаграммы поведения также являются разновидностями логической модели, которые отражают динамические аспекты функ-

Рис. 14.10. Интегрированная модель сложной системы в нотации UML

ционирования сложной системы. И, наконец, диаграммы реализации служат для представления физических компонентов сложной системы и поэтому относятся к ее физической модели. Таким образом, интегрированная модель сложной системы в нотации UML (рис. 14.10) представляется в виде совокупности указанных выше диаграмм.

14.6. Особенности изображения диаграмм языка UML

Большинство перечисленных выше диаграмм являются в своей основе графами специального вида, состоящими из вершин в форме геометрических фигур, связанных между собой ребрами или дугами. Поскольку информация, которую содержит в себе граф, имеет в основном топологический характер, ни геометрические размеры, ни расположение элементов диаграмм (за некоторыми исключениями, как, например, в случае диаграммы последовательностей с метрической осью времени) не имеют принципиального значения.

Для диаграмм языка UML существуют три типа визуальных обозначений, которые важны с точки зрения заключенной в них информации:

связи, представляемые различными линиями на плоскости. Связи в языке UML играют роль дуг и ребер в теории графов, но имеют менее формальный характер;

текст, содержащийся внутри отдельных геометрических фигур на плоскости. Форма этих фигур (прямоугольник, эллипс) соответствует некоторым элементам языка UML (класс, вариант использования) и имеет фиксированную семантику;

графические символы, изображаемые вблизи от тех или иных визуальных элементов диаграмм.

Все диаграммы в языке UML изображаются с использованием фигур на плоскости. Некоторые из фигур (например, кубы) могут представлять собой двумерные проекции трехмерных геометрических тел, но и они рисуются как фигуры на плоскости. Правда, в ближайшее время предполагают включить в язык UML пространственные диаграммы. В рассматриваемой версии языка такая возможность отсутствует.

В языке UML используются четыре основных вида графических конструкций:

значки, или *пиктограммы*, представляющие собой графические фигуры с фиксированными размерами и формой. Нельзя увеличивать размеры таких фигур, чтобы разместить внутри них дополнительные символы. Значки могут размещаться как внутри других графических конструкций, так и вне их. Примерами значков могут служить окончания связей элементов диаграмм или некоторые другие дополнительные обозначения (украшения);

графические символы на плоскости — двумерные символы, изображаемые с помощью некоторых геометрических фигур. Они могут иметь различную высоту и ширину с целью размещения внутри этих фигур других конструкций языка UML. Наиболее часто внутри таких символов помещаются строки текста, которые уточняют семантику или фиксируют отдельные свойства соответствующих элементов языка UML. Информация, содержащаяся внутри фигур, имеет важное значение для конкретной модели проектируемой системы, поскольку регламентирует реализацию соответствующих элементов в программном коде;

пути, представляющие собой последовательности отрезков линий, соединяющих отдельные графические символы. Концевые точки отрезков должны обязательно соприкасаться с геометрическими фигурами, служащими для обозначения вершин диаграмм, как принято в теории графов. С концептуальной точки зрения путям в языке UML придается особое значение, поскольку они являются простыми топологическими сущностями. С другой стороны, отдельные части пути или сегменты могут не существовать сами по себе вне содержащего их пути. Пути всегда соприкасаются с другими графическими символами по обеим границам соответствующих отрезков линий. Другими словами, пути не могут обрываться на диаграмме линией, которая не соприкасается ни с одним из графических символов. Путь может иметь в качестве окончания, или терминатора, специальную графическую фигуру — значок, который изображается на одном из концов линии, являющейся сегментом этого пути;

строки текста, служащие для представления различных видов информации в некоторой грамматической форме. Предполагается, что каждая строка текста соответствует синтаксису в нотации языка UML, посредством которого может быть реализован грамматический разбор этой строки. Такой разбор необходим для получения полной информации о модели. Например, строки текста в различных секциях обозначения класса могут соответствовать атрибутам этого класса или его операциям. На использование строк накладывается важное условие: семантика всех допустимых символов должна быть заранее определена в языке UML или служить предметом его расширения в конкретной модели.

При графическом изображении диаграмм следует придерживаться следующих основных рекомендаций:

Каждая диаграмма должна служить законченным представлением соответствующего фрагмента моделируемой предметной области. Речь идет о том, что в процессе разработки диаграммы необходимо учесть все сущности, важные с точки зрения контекста данной модели и диаграммы. Отсутствие тех или иных элементов на диаграмме служит признаком неполноты модели и может потребовать ее последующей доработки.

Все сущности на диаграмме модели должны быть одного концептуального уровня. Здесь имеется в виду не только согласованность имен одинаковых элементов, но и возможность вложения отдельных диаграмм друг в друга для достижения полноты представлений. В случае достаточно сложных моделей систем желательно придерживаться стратегии последовательного уточнения или детализации отдельных диаграмм.

Вся информация о сущностях должна быть явно представлена на диаграммах. Речь идет о том, что, хотя в языке UML при отсутствии некоторых символов на диаграмме могут быть использованы их значения по умолчанию (например, в случае неявного указания видимости атрибутов и операций классов), необходимо стремиться к явному указанию свойств всех элементов диаграмм.

Диаграммы не должны содержать противоречивой информации. Противоречивость модели может служить причиной серьезнейших проблем при ее реализации и последующем использовании на практике. Например, наличие замкнутых путей при изображении отношений агрегирования или композиции приводит к ошибкам в программном коде, который будет реализовывать соответствующие классы. Наличие элементов с одинаковыми именами и различными атрибутами свойств в одном пространстве имен также приводит к неоднозначной интерпретации и может служить источником проблем.

Наличие в инструментальных CASE-средствах встроенной поддержки визуализации различных диаграмм языка UML позволяет в некоторой степени исключить ошибочное использование тех или иных графических символов, а также контролировать уникальность имен элементов диаграмм. Однако, поскольку вся ответствен-

ность за окончательный контроль непротиворечивости модели лежит на разработчике, необходимо помнить, что неформальный характер языка UML может служить источником потенциальных ошибок, которые в полном объеме вряд ли будут выявлены инструментальными средствами. Именно это обстоятельство требует от разработчиков глубокого знания нотации и семантики всех элементов языка UML.

Диаграммы не следует перегружать текстовой информацией. Принято считать, что визуализация модели является наиболее эффективной, если она содержит минимум пояснительного текста. Как правило, наличие больших фрагментов развернутого текста служит признаком недостаточной проработанности модели или ее неоднородности, когда в рамках одной модели представляется различная по характеру информация. Поскольку общая декомпозиция модели на отдельные типы диаграмм способна удовлетворить самые детальные представления разработчиков о системе, важно уметь правильно отображать те или иные сущности и аспекты моделирования в соответствующие элементы канонических диаграмм.

Каждая диаграмма должна быть самодостаточной для правильной интерпретации всех ее элементов и понимания семантики всех используемых графических символов. Любые пояснительные тексты, которые не являются собственными элементами диаграммы (например комментариями), не должны приниматься во внимание разработчиками. В то же время отдельные достаточно общие фрагменты диаграмм могут уточняться или детализироваться на других диаграммах этого же типа, образуя вложенные или подчиненные диаграммы. Таким образом, модель системы на языке UML представляет собой пакет иерархически вложенных диаграмм, детализация которых должна быть достаточной для последующей генерации программного кода, реализующего проект соответствующей системы.

Число типов диаграмм для конкретной модели приложения не является строго фиксированным. Для простых приложений нет необходимости строить все без исключения типы диаграмм. Некоторые из них могут отсутствовать в проекте системы, и этот факт не будет считаться ошибкой разработчика. Например, модель системы может не содержать диаграмму развертывания для приложения, выполняемого локально на компьютере пользователя. Важно понимать, что перечень диаграмм зависит от специфики конкретного проекта системы.

Любая из моделей системы должна содержать только те элементы, которые определены в нотации языка UML. Существует требование начинать разработку проекта, используя только те конструкции, которые уже определены в метамодели UML. Как показывает практика, этих конструкций вполне достаточно для

представления большинства типовых проектов программных систем. Только в случае отсутствия необходимых базовых элементов языка UML следует использовать механизмы их расширения для адекватного представления конкретной модели системы. При этом не допускается какое бы то ни было переопределение семантики тех элементов, которые отнесены к базовой нотации метамодели языка UML.

Процесс построения отдельных типов диаграмм имеет свои особенности, которые тесно связаны с семантикой элементов этих диаграмм. Сам процесс ООАП в контексте языка UML получил специальное название — рациональный унифицированный процесс (RUP — Rational Unified Process). Концепция RUP и основные его элементы разработаны А. Джекобсоном в ходе его работы над языком UML.

Суть концепции RUP заключается в последовательной декомпозиции или разбиении процесса ООАП на отдельные этапы, на каждом из которых осуществляется разработка соответствующих типов канонических диаграмм модели системы. При этом на начальных этапах RUP строятся логические представления статической модели структуры системы, затем — логические представления модели поведения и лишь после этого — физические представления модели системы. Как нетрудно заметить, в результате RUP должны быть построены канонические диаграммы на языке UML, при этом последовательность их разработки в основном совпадает с их последовательностью нумерации.

Контрольные вопросы

1. Что представляет собой язык UML?
2. Какие принципы моделирования положены в основу языка UML?
3. Перечислите основные задачи языка UML.
4. Что понимается под возможностью расширения и специализации исходных понятий языка UML?
5. Чем объясняется необходимость независимости языка UML от других языков программирования?
6. Из каких частей состоит описание языка UML?
7. Какие уровни входят в состав иерархической структуры языка UML?
8. Объясните взаимосвязь между метамоделью, моделью и объектом.
9. Объясните понятие «пакет», используемое в языке UML.
10. Нарисуйте и поясните графическое обозначение пакета.
11. Что из себя представляет вложенность пакетов?
12. Перечислите основные пакеты метамодели языка UML.
13. Какие пакеты входят в пакет «Основные элементы»?
14. Каково назначение пакета «Вспомогательные элементы»? Используется он в настоящее время или нет?
15. Для чего предназначен пакет «Элементы ядра»?
16. Какие типы данных могут быть использованы в языке UML?

17. Какие пакеты входят в пакет «Элементы поведения»?
18. Каково назначение пакета «Общее поведение»?
19. Для чего предназначен пакет «Автоматы»?
20. Какие пакеты входят в пакет «Общие механизмы»?
21. Каково назначение пакета «Управление моделями»?
22. Перечислите и объясните виды представлений, входящих в метамодель языка UML.
23. Какие правила используются:
а) для специфицирования семантики естественных языков;
б) для выделения текста?
24. Какие типы диаграмм определены в языке UML?
25. Какие диаграммы используются в качестве самостоятельных представлений?
26. Нарисуйте интегрированную модель сложной системы.
27. Какие виды визуальных обозначений используются в языке UML?
28. Какие виды графических конструкций используются в языке UML?
29. Какие рекомендации следует учитывать при графическом изображении диаграмм?

СПИСОК ЛИТЕРАТУРЫ

1. *Вендров А. М.* Проектирование программного обеспечения экономических информационных систем. — М.: Финансы и статистика, 2000. — 352 с.
2. *Канер С., Фолк Д., Кек Нгуен Е.* Тестирование программного обеспечения: Пер. с англ. — Киев: ДиаСофт, 2000. — 544 с.
3. *Соммервилл И.* Инженерия программного обеспечения. — М.: СПб.: Киев: Изд. дом «Вильямс», 2002. — 624 с.
4. *Фридман А. Л.* Основы объектно-ориентированной разработки программных систем. — М.: Финансы и статистика, 2000. — 200 с.
5. *Boehm B.* Software Engineering Economics. — London: Prentice-Hall International Inc., 1981. — 630 p.
6. *Booch G., Caswell P., Deborah L.* Software Metrics: Establishing a Company-Wide Program. — London: Prentice-Hall International Inc., 1987. — 543 p.
7. *Humphrey W., Watts S.* Managing the Software Process. — Santa Clara: Addison-Wesley Publishing Company Inc., 1990. — 600 p.
8. *Myers G., Glenford J.* The Art of Software Testing. — New Jersey: A Wiley-Interscience Publication, 1979. — 540 p.

ОГЛАВЛЕНИЕ

Учебное издание

Рудаков Александр Викторович

Технология разработки программных продуктов

Учебное пособие

2-е издание, стереотипное

Редактор *Е. М. Зубкович*
Технический редактор *Е. Ф. Коржуева*
Компьютерная верстка: *О. В. Пешкетова*
Корректоры *И. В. Могилевец, Н. И. Смородина*

Диапозитивы предоставлены издательством

Изд. № 102107932. Подписано в печать 28.07.2006. Формат 60×90/16.
Гарнитура «Таймс». Печать офсетная. Бумага тип. № 2. Усл. печ. л. 13,0.
Тираж 2 500 экз. Заказ № 17235.

Издательский центр «Академия». www.academia-moscow.ru
Санитарно-эпидемиологическое заключение № 77.99.02.953.Д.004796.07.04 от 20.07.2004.
117342, Москва, ул. Бутлерова, 17-Б, к. 360. Тел./факс: (495)330-1092, 334-8337.

Отпечатано в ОАО «Саратовский полиграфический комбинат».
410004, г. Саратов, ул. Чернышевского, 59.

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНЫХ ПРОДУКТОВ

ISBN 5-7695-3281-5

Издательский центр
«Академия»
www.academia-moscow.ru