ВОЕННО-КОСМИЧЕСКАЯ АКАДЕМИЯ
имени А.Ф. Можайского

В.А. Лохвицкий

ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Санкт-Петербург
2014

Рецензенты:
кандидат технических наук, доцент **Г.А. Брякалов**;
кандидат технических наук **В.С. Забузов**

Лохвицкий В.А

Технологии разработки программного обеспечения: учеб. пособие / В.А. Лохвицкий. – СПб.: ВКА имени А.Ф. Можайского, 2014. – 123 с.

В данном учебном пособии содержатся основные материалы дисциплины «Технологии разработки программного обеспечения». В нем рассмотрены основные понятия программной инженерии, методы проектирования, кодирования, тестирования, отладки и оценивания качества программных средств и систем.

Учебное пособие предназначено для курсантов и слушателей вычислительных специальностей ВКА имени А.Ф. Можайского, получивших подготовку в рамках дисциплин «Информатика», «Программирование», «Теория языков программирования и методы трансляции», «Структуры и алгоритмы обработки данных».

Материал написан на основе специального курса лекций. В курсе изложены сведения, необходимые для понимания принципов, методов и средств проектирования, разработки, тестирования, отладки, а также оценивания и управления качеством программных средств и систем.

Подписано к печ. 25.11.2014 Формат печатного листа 445×300/8
Гарнитура Times New Roman Авт. л. 7,5
Уч.-печ. л. 16,0 Заказ 2918 Бесплатно

Типография ВКА имени А.Ф. Можайского

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Технологией программирования называют совокупность методов и средств, используемых в процессе разработки программного обеспечения. Как любая другая технология, эта представляет собой набор технологических инструкций, включающих:

- указание последовательности выполнения технологических операций;
- перечисление условий, при которых выполняется та или иная операция;
- описания самих операций, где для каждой операции определены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии и методы оценки и т.п.

Кроме набора операций и их последовательности, технология также определяет способ описания проектируемой системы, точнее модели, используемой на конкретном этапе разработки.

Различают технологии:

1) используемые на конкретных этапах разработки или для решения отдельных задач этих этапов;

2) охватывающие несколько этапов (или весь процесс разработки).

В основе первых, как правило, лежит ограниченно применимый *метод,* позволяющий решить конкретную задачу. В основе вторых обычно лежит базовый метод или *подход,* определяющий совокупность методов, используемых на разных этапах разработки, или *методологию.*

Отличительной особенностью современного этапа развития технологии программирования, является создание и внедрение автоматизированных технологий разработки и сопровождения программного обеспечения, которые были названы CASE-технологиями (Computer-Aided Software/System Engineering – разработка программного обеспечения/программных систем с использованием компьютерной поддержки). Без средств автоматизации разработка достаточно сложного программного обеспечения на настоящий момент становится трудноосуществимой: память человека уже не в состоянии фиксировать все детали, которые необходимо учитывать при разработке программного обеспечения. На сегодня существуют CASE-технологии, поддерживающие как структурный, так и объектный (в том числе и компонентный) подходы к программированию.

Появление нового подхода не означает, что отныне все программное обеспечение будет создаваться из программных компонентов, но анализ существующих проблем разработки сложного программного обеспечения показывает, что он будет применяться достаточно широко.

1. ОСНОВНЫЕ ПОНЯТИЯ ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1. Особенности и проблемы создания современных программных продуктов

Процесс создания программного средства (**ПС**), как и любая другая интеллектуальная деятельность, основан на человеческих суждениях и умозаключениях, то есть является творческим. Вследствие этого все попытки автоматизировать процесс создания ПС имеют лишь ограниченный успех. CASE-средства могут помочь лишь в реализации некоторых этапов процесса разработки ПС. Главная причина ограниченного применения автоматизированных средств – огромное многообразие видов деятельности, связанных с разработкой программных продуктов. Кроме того, разработчики используют разные подходы к созданию ПС. Также различаются характеристики и возможности создаваемых систем. Поэтому даже в одном коллективе разработчиков при создании разных ПС могут использоваться различные подходы и технологии.

Несмотря на то, что наблюдается огромное разнообразие подходов, методов и технологий создания ПС, существуют фундаментальные базовые процессы, без реализации которых не может обойтись ни одна технология разработки ПС. К ним относятся:

1. ***Разработка спецификации ПС.*** Это фундамент любого программного средства. Спецификация определяет все функции и действия, которые будет выполнять разрабатываемая система.

2. ***Проектирование и реализация ПС.*** Это процесс непосредственного создания ПС на основе спецификации.

3. ***Аттестация ПС.*** Разработанное программное средство должно быть аттестовано на соответствие требованиям заказчика.

4. ***Эволюция ПС.*** Любые программные системы должны модифицироваться в соответствии с изменениями требований заказчика.

Хотя не существует «идеального» процесса создания ПС, во многих организациях-разработчиках пытаются его усовершенствовать. Совершенствовать процесс можно разными путями. Например, путем стандартизации, которая уменьшит разнородность используемых технологий и сделает экономически выгодной автоматизацию разработок.

1.1.1. Классификация программ

Все программы по характеру использования и категориям пользователей можно разделить на два класса (рис. 1.1) – утилитарные программы и программные продукты (изделия).

Рис. 1.1. Классификация программ по категориям пользователей

Утилитарные программы («программы для себя») предназначены для удовлетворения нужд их разработчиков. Чаще всего утилитарные программы выполняют функцию сервиса в технологии обработки данных либо являются программами решения функциональных задач, не предназначенных для широкого распространения.

Программные продукты (изделия) предназначены для удовлетворения потребностей пользователей, широкого распространения и продажи.

В настоящее время существуют и другие варианты легального распространения программных продуктов, которые появились с использованием глобальных или региональных телекоммуникаций.

По характеру использования программы можно условно разделить на две основные группы:

- ***freeware*** – бесплатные программы, свободно распространяемые, поддерживаются самим пользователем, который правомочен вносить в них необходимые изменения;
- ***shareware*** – некоммерческие (условно-бесплатные) программы, которые могут использоваться, как правило, бесплатно. При условии регулярного использования подобных продуктов осуществляется взнос определенной суммы.

Ряд производителей использует *OEM-программы* (Original Equipment Manufacturer), т.е. встроенные программы, устанавливаемые на компьютеры или поставляемые вместе с вычислительной техникой.

1.1.2. Понятие «программный продукт»

Программный продукт должен быть соответствующим образом подготовлен к эксплуатации, иметь необходимую техническую документацию, предоставлять сервис и гарантию надежной работы программы, иметь товарный знак изготовителя, а также желательно наличие кода государственной регистрации. Только при таких условиях созданный программный комплекс может быть назван программным продуктом.

Программный продукт – комплекс взаимосвязанных программ для решения определенной проблемы (задачи) массового спроса, подготовленный к реализации как любой вид промышленной продукции.

Путь от «программ для себя» до программных продуктов достаточно долгий. Он связан с изменениями технической и программной среды разработки и эксплуатации программ, с появлением и развитием самостоятельной отрасли – информационного бизнеса, для которой характерны разделение труда организаций-разработчиков программ, их дальнейшая специализация, формирование рынка программных средств и информационных услуг.

Программные продукты могут создаваться как:

- индивидуальная разработка под заказ;
- разработка для массового распространения среди пользователей.

При *индивидуальной разработке* организация-разработчик создает оригинальный программный продукт, учитывающий специфику обработки данных для конкретного заказчика.

При разработке *для массового распространения* организация-разработчик, с одной стороны, должна обеспечить универсальность выполняемых функций обработки данных, с другой стороны, гибкость и настраиваемость программного продукта на условия конкретного применения. Отличительной особенностью программных продуктов должна быть их системность – функциональная полнота и законченность реализуемых функций обработки, которые применяются в совокупности.

Программный продукт разрабатывается на основе промышленной технологии выполнения проектных работ с применением современных инструментальных средств программирования. Специфика заключается в уникальности процесса разработки алгоритмов и программ, зависящего от характера обработки информации и используемых инструментальных средств. На создание программных продуктов затрачиваются значительные ресурсы – трудовые, материальные, финансовые. Кроме этого, требуется высокая квалификация разработчиков.

Как правило, программные продукты требуют сопровождения, которое осуществляется специализированными организациями-распространителями про-

грамм (дистрибьюторами), реже – организациями-разработчиками. Сопровождение программ массового применения сопряжено с большими трудозатратами на исправление обнаруженных ошибок, создание новых версий программ и т.п.

Сопровождение программного продукта – поддержка работоспособности программного продукта, переход на его новые версии, внесение изменений, исправление обнаруженных ошибок и т.п.

Программные продукты в отличие от традиционных программных изделий не имеют строго регламентированного набора качественных характеристик, задаваемых при создании программ, либо эти характеристики невозможно заранее точно указать или оценить, т.к. одни и те же функции обработки, обеспечиваемые программным средством, могут иметь различную глубину проработки. Даже время и затраты на разработку программных продуктов не могут быть определены с большой степенью точности заранее.

1.1.3. Основные характеристики программных продуктов

Основными характеристиками программ являются:

- алгоритмическая сложность (логика алгоритмов обработки информации);
- состав и глубина проработки реализованных функций обработки;
- полнота и системность функций обработки;
- объем файлов программ;
- требования к операционной системе и техническим средствам обработки со стороны программного средства;
- объем дисковой памяти;
- размер оперативной памяти для запуска программ;
- тип процессора;
- версия операционной системы;
- требования наличия вычислительной сети и др.

Программные продукты имеют многообразие показателей качества, которые отражают следующие аспекты:

- насколько хорошо (просто, надежно, эффективно) можно использовать программный продукт;
- насколько легко эксплуатировать программный продукт;
- можно ли использовать программный продукт при изменении условия его применения и др.

Дерево характеристик качества программных продуктов представлено на рис. 1.2.

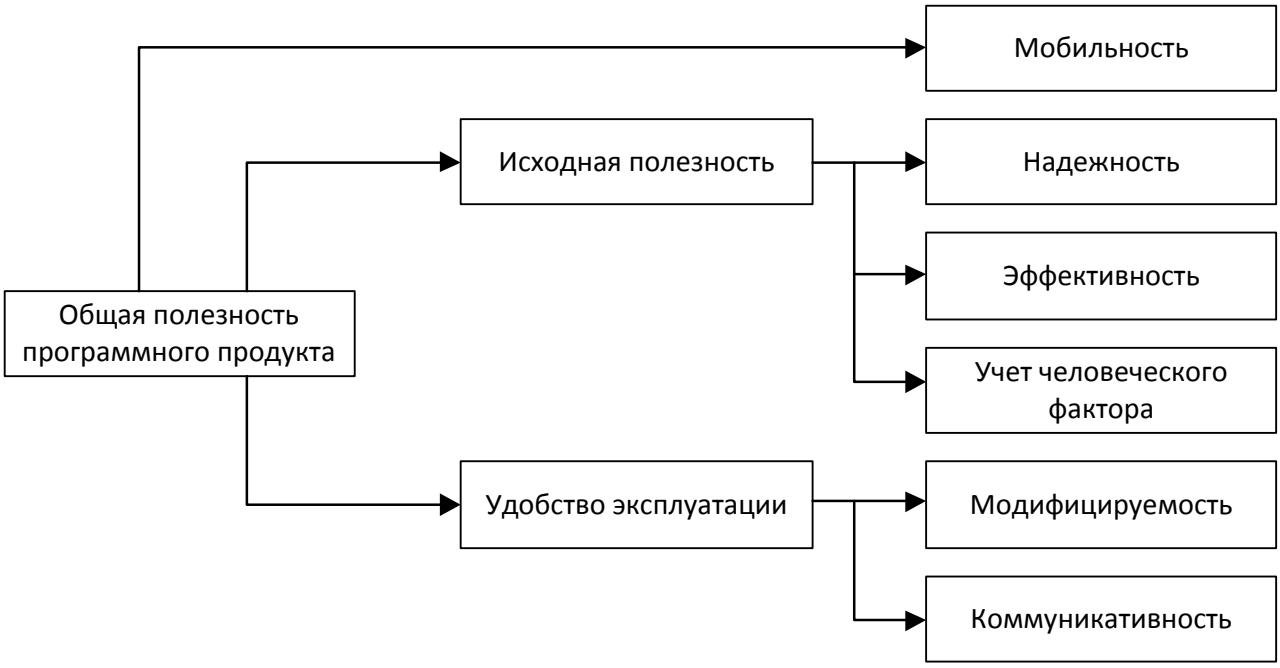

Рис. 1.2. Основные характеристики качества программных продуктов

Мобильность *программных продуктов* означает их независимость от технического комплекса системы обработки данных, операционной среды, сетевой технологии обработки данных, специфики предметной области и т.п. Мобильный (многоплатформный) программный продукт может быть установлен на различных моделях компьютеров и операционных систем, без ограничений на его эксплуатацию в условиях вычислительной сети. Функции обработки такого программного продукта пригодны для массового использования без каких-либо изменений.

Надежность *работы программного продукта* определяется бессбойностью и устойчивостью в работе программ, точностью выполнения предписанных функций обработки, возможностью диагностики возникающих в процессе работы программ ошибок.

Эффективность *программного продукта* оценивается как с позиций прямого его назначения – требований пользователя, так и с точки зрения расхода вычислительных ресурсов, необходимых для его эксплуатации. Расход вычислительных ресурсов оценивается через объем внешней памяти для размещения программ и объем оперативной памяти для запуска программ.

Учет человеческого фактора означает обеспечение дружественного интерфейса для работы конечного пользователя, наличие контекстно-зависимой подсказки или обучающей системы в составе программного средства, хорошей документации для освоения и использования заложенных в программном средстве функциональных возможностей, анализа и диагностики возникших ошибок и др.

Модифицируемость *программных продуктов* означает способность к внесению изменений, например расширение функций обработки, переход на другую техническую базу обработки и т.п.

Коммуникативность *программных продуктов* основана на максимально возможной их интеграции с другими программами, обеспечении обмена данными в общих форматах представления (экспорт/импорт баз данных, внедрение или связывание объектов обработки и др.).

В условиях существования рынка программных продуктов важными характеристиками являются:

- стоимость;
- количество продаж;
- время нахождения на рынке (длительность продаж);
- известность организации-разработчика и программы;
- наличие программных продуктов аналогичного назначения.

Спецификой программных продуктов (в отличие от большинства промышленных изделий) является также и то, что их эксплуатация должна выполняться на правовой основе – лицензионных соглашений между разработчиком и пользователями с соблюдением авторских прав разработчиков программных продуктов.

1.1.4. Специфика разработки программных средств

Разработка программных средств (**ПС**) имеет ряд специфических особенностей:

1. Наличие противоречия между неформальным характером определения требований к программному средству (постановки задачи) и формализованным процессом разработки программного средства

2. Разработка ПС носит творческий характер (на каждом шаге приходится делать некоторый выбор, принимать решение), а не сводится к выполнению какой-либо последовательности регламентированных действий. Тем самым такая разработка ближе к процессу *проектирования* сложных устройств, но никак не к их массовому производству. Этот творческий характер разработки ПС сохраняется до самого ее конца.

3. Следует отметить также особенность продукта разработки. Он представляет собой некоторую совокупность текстов (т.е. статических объектов), смысл же (семантика) этих текстов выражается процессами обработки данных и действиями пользователей, запускающих эти процессы (т.е. является динамическим). Это предопределяет выбор разработчиком ряда специфичных приемов, методов и средств.

4. Продукт разработки имеет и другую специфическую особенность: ПС при своем использовании (эксплуатации) не расходуется и не расходует используемых ресурсов.

1.2. Жизненный цикл программного средства

Под *жизненным циклом ПС* понимают весь период его разработки и эксплуатации (использования), начиная от момента возникновения замысла ПС и кончая прекращением всех видов его использования.

Жизненный цикл охватывает довольно сложный процесс создания и использования ПС. Этот процесс может быть организован по-разному для разных классов ПС и в зависимости от особенностей коллектива разработчиков.

В настоящее время можно выделить пять основных подходов к организации процесса создания и использования ПС:

1. ***Водопадный подход.*** При таком подходе разработка ПС состоит из цепочки этапов. На каждом этапе создаются документы, используемые на последующем этапе. В исходном документе фиксируются требования к ПС. В конце этой цепочки создаются программы, включаемые в ПС.

2. ***Исследовательское программирование.*** Этот подход предполагает быструю (насколько это возможно) реализацию рабочих версий программ ПС, выполняющих лишь в первом приближении требуемые функции. После экспериментального применения реализованных программ производится их модификация с целью сделать их более полезными для пользователей. Этот процесс повторяется до тех пор, пока ПС не будет достаточно приемлемо для пользователей. Такой подход применялся на ранних этапах развития программирования, когда технологии программирования не придавалось большого значения (использовалась «интуитивная» технология). В настоящее время этот подход применяется для разработки таких ПС, для которых пользователи не могут точно сформулировать требования (например, для разработки систем искусственного интеллекта).

3. ***Прототипирование.*** Этот подход моделирует начальную фазу исследовательского программирования вплоть до создания рабочих версий программ, предназначенных для проведения экспериментов с целью установить требования к ПС. В дальнейшем должна последовать разработка ПС по установленным требованиям в рамках какого-либо другого подхода (например, водопадного).

4. ***Формальные преобразования.*** Этот подход включает разработку формальных спецификаций ПС и превращение их в программы путем корректных преобразований. На этом подходе базируется компьютерная технология (CASE-технология) разработки ПС.

5. ***Сборочное программирование.*** Этот подход предполагает, что ПС конструируется, главным образом, из компонент, которые уже существуют. Должно быть некоторое хранилище (библиотека) таких компонент, каждая из которых может многократно использоваться в разных ПС. Такие компоненты называются *повторно используемыми* (reusable). Процесс разработки ПС при данном подходе состоит скорее из сборки программ из компонент, чем из их программирования.

1.3. Стратегии разработки программных средств

1.3.1. Основные понятия

На начальном этапе развития вычислительной техники ПС разрабатывались с использованием так называемой «интуитивной» технологии разработки. Модель такого процесса разработки ПС иллюстрирует рис. 1.3.

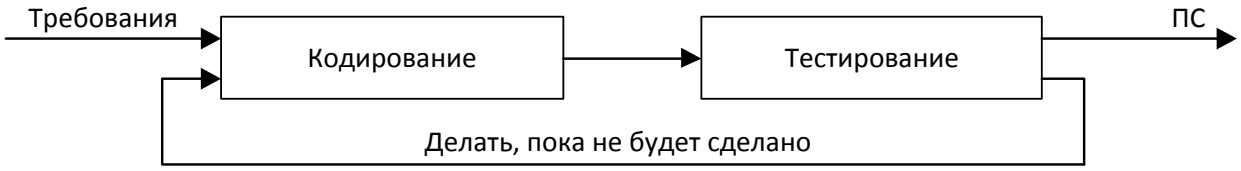

Рис. 1.3. Модель разработки ПС с применением «интуитивной» технологии

Очевидно, что **недостатками** данной модели являются:

- неструктурированность процесса разработки ПС;
- ориентация на индивидуальные знания и умения программиста;
- сложность управления и планирования проекта;
- большая длительность и стоимость разработки;
- низкое качество программных продуктов;
- высокий уровень рисков проекта.

Для устранения или сокращения вышеназванных недостатков к настоящему времени созданы и широко используются три основные стратегии разработки программного обеспечения (**ПО**):

1. **Однократный проход**. Данная стратегия предполагает линейную последовательность этапов конструирования.

2. **Инкрементная стратегия**. В начале процесса определяются все пользовательские и системные требования, оставшаяся часть конструирования выполняется в виде последовательности версий. Первая версия реализует часть запланированных возможностей, следующая версия реализует дополнительные возможности и т.д., пока не будет разработана полнофункциональная система.

3. **Эволюционная стратегия**. Система также строится в виде последовательности версий, но в начале процесса определены не все требования. Требования уточняются в результате разработки версий.

Характеристики стратегий конструирования ПО в соответствии с требованиями стандарта IEEE/EIA 12207.2 приведены в табл. 1.1.

Таблица 1.1

Характеристики стратегий конструирования ПО

Стратегия конструирования	В начале процесса определены все требования?	Множество циклов конструирования?	Промежуточное ПО распространяется?
Однократный проход	Да	Нет	Нет
Инкрементная (запланированное улучшение продукта)	Да	Да	Может быть
Эволюционная	Нет	Да	Да

Каждая из стратегий разработки имеет свои достоинства и уязвимые места, определяемые степенью соответствия выбранной стратегии возможностям реализации конкретного проекта. Следует подчеркнуть, что одни и те же свойства стратегии при ее правильном выборе могут рассматриваться как преимущества, а если стратегия выбрана неверно – становиться недостатком.

Три базовые стратегии могут быть реализованы с помощью различных моделей жизненного цикла (**ЖЦ**).

1.3.2. Каскадная стратегия разработки программных средств

Каскадная стратегия представляет собой однократный проход этапов разработки. Данная стратегия основана на полном определении всех требований к разрабатываемому программному средству или системе в начале процесса разработки. Каждый этап разработки начинается после завершения предыдущего этапа. Возврат к уже выполненным этапам не предусматривается. Промежуточные продукты разработки в качестве версии программного средства (системы) не распространяются.

Представителями моделей, реализующих каскадную стратегию, являются каскадная (водопадная) и V-образная модели.

Основными достоинствами каскадной стратегии, проявляемыми при разработке соответствующего ей проекта, являются:

- стабильность требований в течение ЖЦ разработки;

- необходимость только одного прохода этапов разработки, что обеспечивает простоту применения стратегии;
- простота планирования, контроля и управления проектом;
- доступность для понимания заказчиками.

К **основным недостаткам** каскадной стратегии следует отнести:

- сложность полного формулирования требований в начале процесса разработки и невозможность их динамического изменения на протяжении ЖЦ;
- линейность структуры процесса разработки. Современные ПС или системы обычно слишком велики и сложны, чтобы все работы по их созданию выполнять однократно. В результате возврат к предыдущим шагам для решения возникающих проблем приводит к увеличению финансовых затрат и нарушению графика работ;
- непригодность промежуточных продуктов для использования;
- недостаточное участие пользователя в процессе разработки ПС – только в самом начале (при разработке требований) и в конце (во время приемочных испытаний), что приводит к невозможности предварительной оценки пользователем качества программного средства или системы.

Использование каскадной стратегии наиболее эффективно в следующих случаях:

- при разработке проектов с четкими, неизменяемыми в течение ЖЦ требованиями и понятной реализацией;
- при разработке проектов невысокой сложности, например:
- создания программного средства или системы такого же типа, как уже однажды разработанные;
- создания новой версии уже существующего программного средства или системы;
- переноса уже существующего продукта на новую платформу;
- при выполнении больших проектов в качестве составной части моделей ЖЦ, реализующих другие стратегии разработки.

Резюме

Каскадная стратегия представляет собой однократный проход этапов разработки. Данная стратегия основана на полном определении всех требований к программному средству или системе в начале процесса разработки. Возврат к уже выполненным этапам не предусматривается. Промежуточные результаты в качестве версии программного средства (системы) не распространяются. Каскадная стратегия имеет достоинства и недостатки, определяемые правильностью выбора данной стратегии по отношению к конкретному проекту.

1.3.3. Инкрементная стратегия разработки программных средств

Инкрементная стратегия представляет собой многократный проход этапов разработки с запланированным улучшением результата (рис. 1.4).

Данная стратегия основана на полном определении всех требований к разрабатываемому программному средству (системе) в начале процесса разработки. Однако полный набор требований реализуется постепенно в соответствии с планом в последовательных циклах разработки. Результат каждого цикла называется инкрементом.

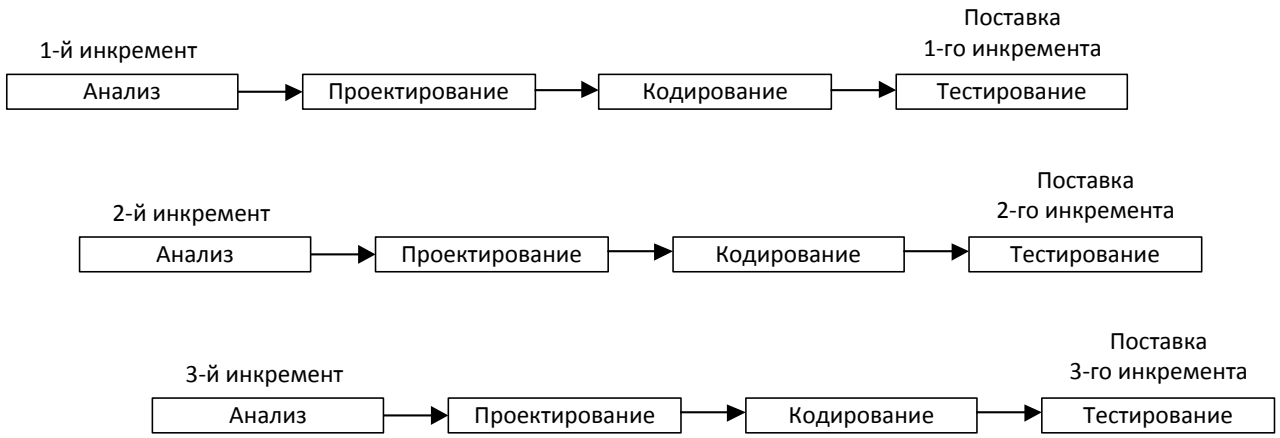

Рис. 1.4. Инкрементная модель разработки ПС

Первый инкремент реализует базовые функции программного средства. В последующих инкрементах функции программного средства постепенно расширяются, пока не будет реализован весь набор требований. Различия между инкрементами соседних циклов в ходе разработки постепенно уменьшаются.

Результат каждого цикла разработки может распространяться в качестве очередной поставляемой версии программного средства или системы.

Особенностью инкрементной стратегии является большое количество циклов разработки при незначительной продолжительности цикла и небольших отличиях между инкрементами соседних циклов. Например, данная стратегия разработки ПС и систем используется в компании Microsoft. Здесь на каждую версию программного средства разрабатывается около тысячи инкрементов.

Период разработки инкремента составляет одни сутки (например, днем инкремент разрабатывается, ночью тестируется). В ряде организаций используется недельный период разработки инкремента (чаще всего пять дней – разработка, два дня – тестирование).

Инкрементная стратегия обычно основана на объединении элементов каскадной модели и прототипирования (макетирования). При этом использование

прототипирования позволяет существенно сократить продолжительность разработки каждого инкремента и всего проекта в целом.

Под *прототипом* понимается легко поддающаяся модификации и расширению рабочая модель разрабатываемого программного средства (или системы), позволяющая пользователю получить представление об его ключевых свойствах до полной реализации.

Современной реализацией инкрементной стратегии является экстремальное программирование.

Основными достоинствами инкрементной стратегии являются:

- возможность получения функционального продукта после реализации каждого инкремента;
- короткая продолжительность создания инкремента; это приводит к сокращению сроков начальной поставки, позволяет снизить затраты на первоначальную и последующие поставки программного продукта;
- предотвращение реализации громоздких спецификаций требований; стабильность требований во время создания определенного инкремента; возможность учета изменившихся требований;
- снижение рисков по сравнению с каскадной стратегией;
- включение в процесс пользователей, что позволяет оценить функциональные возможности продукта на более ранних этапах разработки и в конечном итоге приводит к повышению качества программного продукта, снижению затрат и времени на его разработку.

К **основным недостаткам** инкрементной стратегии следует отнести:

- необходимость полного функционального определения системы или программного средства в начале ЖЦ для обеспечения планирования инкрементов и управления проектом;
- возможность текущего изменения требований к системе или программному средству, которые уже реализованы в предыдущих инкрементах;
- сложность планирования и распределения работ;
- проявление человеческого фактора, связанного с тенденцией к оттягиванию решения трудных проблем на поздние инкременты, что может нарушить график работ или снизить качество программного продукта.

Использование инкрементной данной стратегии разработки ПО наиболее эффективно в следующих случаях:

- при разработке проектов, в которых большинство требований можно сформулировать заранее, но часть из них может быть уточнена через определенный период времени;

- при разработке сложных проектов с заранее сформулированными требованиями, для которых разработка системы или программного средства за один цикл связана с большими трудностями;
- при необходимости быстро поставить на рынок продукт, имеющий базовые функциональные свойства;
- при разработке проектов с низкой или средней степенью рисков;
- при выполнении проекта с применением новых технологий.

Резюме

Инкрементная стратегия представляет собой многократный проход этапов разработки с запланированным улучшением результата. Данная стратегия основана на полном определении всех требований к разрабатываемому программному средству или системе в начале процесса разработки. Однако полный набор требований реализуется постепенно в соответствии с планом в последовательных циклах разработки. При инкрементной стратегии часто используется прототипирование. Инкрементная стратегия имеет достоинства и недостатки, определяемые правильностью выбора данной стратегии по отношению к конкретному проекту.

1.3.4. Эволюционная стратегия разработки программных средств

Эволюционная стратегия представляет собой многократный проход этапов разработки. Данная стратегия основана на частичном определении требований к разрабатываемому программному средству или системе в начале процесса разработки. Требования постепенно уточняются в последовательных циклах разработки. Результат каждого цикла разработки обычно представляет собой очередную поставляемую версию программного средства или системы.

Следует отметить, что в общем случае для эволюционной стратегии характерно существенно меньшее количество циклов разработки при большей продолжительности цикла по сравнению с инкрементной стратегией. При этом результат каждого цикла разработки (очередная версия программного средства или системы) гораздо сильнее отличается от результата предыдущего цикла.

Как и при инкрементной стратегии, при реализации эволюционной стратегии зачастую используется прототипирование.

В данном случае основной целью прототипирования является обеспечение полного понимания требований. Оно позволяет итеративно уточнять требования к продукту при достижении предельно высокой производительности разработки проекта и одновременном снижении затрат. Использование прототипирования наиболее эффективно в тех случаях, когда в проекте применяются новые концепции или новые технологии, так как в этих случаях достаточно слож-

но полностью и корректно разработать детальные технические требования к системе или программному средству на ранних стадиях цикла разработки.

Для итеративного уточнения требований при применении прототипиро вания в цикле разработки должен участвовать заказчик.

Представителем моделей, реализующих эволюционную стратегию, является, например, спиральная модель ЖЦ (п. 1.4.3).

Основными достоинствами эволюционной стратегии являются:

- возможность уточнения и внесения новых требований в процессе разработки;
- пригодность промежуточного продукта для использования;
- возможность управления рисками;
- обеспечение широкого участия пользователя в проекте, начиная с ранних этапов, что минимизирует возможность разногласий между заказчиками и разработчиками и обеспечивает создание продукта высокого качества;
- реализация преимуществ каскадной и инкрементной стратегий.

К **недостаткам** эволюционной стратегии следует отнести:

- неизвестность точного количества необходимых итераций и сложность определения критериев для продолжения процесса разработки на следующей итерации; это может вызвать задержку реализации конечной версии системы или программного средства;
- сложность планирования и управления проектом;
- необходимость активного участия пользователей в проекте, что на практике не всегда осуществимо;
- необходимость в мощных инструментальных средствах и методах прототипирования;
- возможность отодвигания решения трудных проблем на последующие циклы, что может привести к несоответствию полученных продуктов требованиям заказчиков.

Использование эволюционной стратегии наиболее эффективно в следующих случаях:

- при разработке проектов, для которых требования слишком сложны, неизвестны заранее, непостоянны или требуют уточнения;
- при разработке сложных проектов, в том числе:
- больших долгосрочных проектов;
- проектов по созданию новых, не имеющих аналогов ПС или систем;
- проектов со средней и высокой степенью рисков;
- проектов, для которых нужна проверка концепции, демонстрация технической осуществимости или промежуточных продуктов;
- при разработке проектов, использующих новые технологии.

Резюме

Эволюционная стратегия представляет собой многократный проход этапов разработки. Данная стратегия основана на частичном определении требований к разрабатываемому программному средству или системе в начале процесса разработки. Требования постепенно уточняются в последовательных циклах разработки. Результат каждого цикла разработки обычно представляет собой очередную поставляемую версию программного средства или системы.

При эволюционной стратегии часто используется прототипирование. Эволюционная стратегия имеет достоинства и недостатки, определяемые правильностью выбора данной стратегии по отношению к конкретному проекту.

1.4. Основные модели разработки программных средств

1.4.1. Каскадная (водопадная) модель

Старейшей парадигмой процесса разработки ПО является классический жизненный цикл (автор *Уинстон Ройс*, 1970).

Очень часто классический жизненный цикл называют *каскадной* или *водопадной моделью*, подчеркивая, что разработка рассматривается как последовательность этапов, причем переход на следующий, иерархически нижний этап происходит только после полного завершения работ на текущем этапе (рис. 1.5).

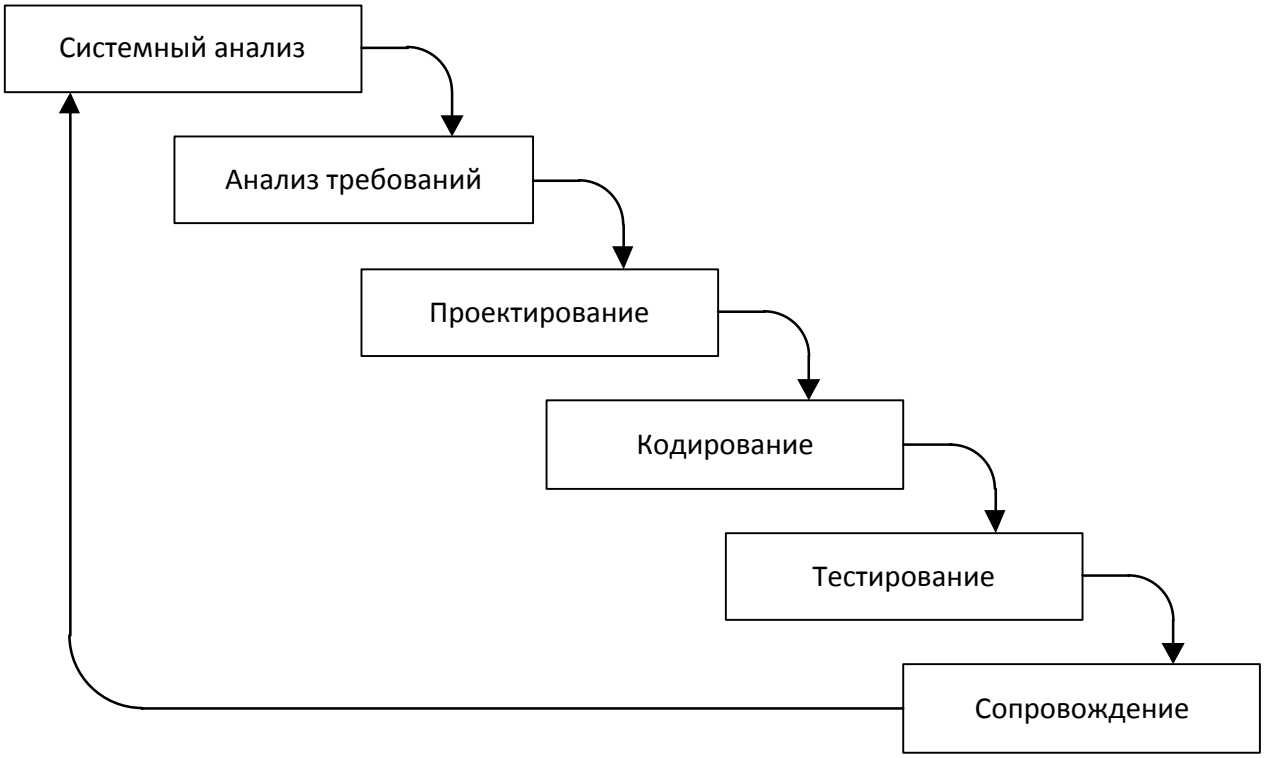

Рис. 1.5. Классический жизненный цикл разработки ПО

Подразумевается, что разработка начинается на системном уровне и проходит через анализ, проектирование, кодирование, тестирование и сопровождение. При этом моделируются действия стандартного инженерного цикла.

Системный анализ задает роль каждого элемента в компьютерной системе, взаимодействие элементов друг с другом. Поскольку ПО является лишь частью большой системы, то анализ начинается с определения требований ко всем системным элементам и назначения подмножества этих требований программному «элементу». Необходимость применения системного подхода наиболее обоснована при проектировании интерфейса ПО с другими элементами (аппаратурой, людьми, базами данных). На этом же этапе начинается решение задачи планирования проекта ПО. В ходе планирования проекта определяются объем проектных работ и их риск, необходимые трудозатраты, формируются рабочие задачи и план-график работ.

Анализ требований относится к программному элементу – программному обеспечению. Уточняются и детализируются его функции, характеристики и интерфейс. Все определения документируются в *спецификации анализа.* Здесь же завершается решение задачи планирования проекта.

Проектирование состоит в создании представлений:

- архитектуры ПО;
- модульной структуры ПО;
- алгоритмической структуры ПО;
- структуры данных;
- входного и выходного интерфейса (входных и выходных форм данных).

Исходные данные для проектирования содержатся в *спецификации анализа,* то есть в ходе проектирования выполняется трансляция требований к ПО во множество проектных представлений. При решении задач проектирования основное внимание уделяется качеству будущего программного продукта.

Кодирование состоит в переводе результатов проектирования в текст на языке программирования.

Тестирование – это выполнение программы для выявления дефектов в функциях, логике и форме реализации программного продукта.

Сопровождение – это внесение изменений в эксплуатируемое ПО. Цели изменений:

- исправление ошибок;
- адаптация к изменениям внешней для программного обеспечения среды;
- усовершенствование программного обеспечения по требованиям заказчика.

Сопровождение ПО состоит в повторном применении каждого из предшествующих шагов (этапов) жизненного цикла к существующей программе, но не в разработке новой программы.

Как и любая инженерная схема, классический жизненный цикл имеет достоинства и недостатки.

К **достоинствам** классического жизненного цикла можно отнести то, что появляется возможность сформировать план и временной график по всем этапам проекта, а также упорядочить ход конструирования.

Недостатки классического жизненного цикла:

- реальные проекты часто требуют отклонения от стандартной последовательности шагов;
- цикл основан на точной формулировке исходных требований к ПО (на практике в начале проекта требования заказчика определены лишь частично);
- результаты проекта доступны заказчику только в конце работы.

1.4.2. Спиральная модель

Спиральная модель – это классический пример применения эволюционной стратегии конструирования.

Спиральная модель (автор *Барри Боэм*, 1988) базируется на лучших свойствах классического жизненного цикла и макетирования, к которым добавляется новый элемент – анализ риска, отсутствующий в этих парадигмах.

Модель определяет четыре действия, представляемые четырьмя квадрантами спирали (рис. 1.6):

- *планирование* – определение целей, вариантов и ограничений;
- *анализ риска* – анализ вариантов и распознавание/выбор риска;
- *конструирование* – разработку продукта следующего уровня;
- *оценивание* – оценку заказчиком текущих результатов конструирования.

Интегрирующий аспект спиральной модели очевиден при учете радиального измерения спирали. С каждой итерацией по спирали (продвижением от центра к периферии) строятся все более полные версии ПО.

В первом витке спирали определяются начальные цели, варианты и ограничения, распознается и анализируется риск. Если анализ риска показывает неопределенность требований, на помощь разработчику и заказчику приходит макетирование (используемое в квадранте конструирования). Для дальнейшего определения проблемных и уточненных требований может быть использовано моделирование. Заказчик оценивает инженерную (конструкторскую) работу и вносит предложения по модификации (квадрант оценки заказчиком). Следующая фаза планирования и анализа риска базируется на предложениях заказчика.

В каждом цикле по спирали результаты анализа риска формируются в виде «продолжать, не продолжать». Если риск слишком велик, проект может быть остановлен.

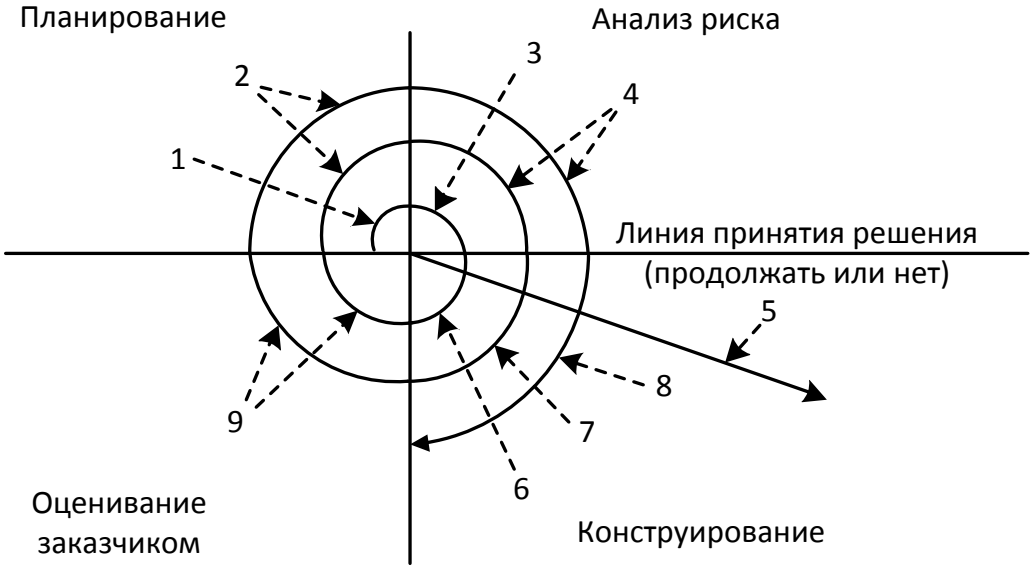

Рис. 1.6. Спиральная модель:
1 – начальный сбор требований и планирование проекта;
2 – та же работа, но на основе рекомендаций заказчика; 3 – анализ риска на основе начальных требований; 4 – анализ риска на основе реакции заказчика;
5 – переход к комплексной системе; 6 – начальный макет системы;
7 – следующий уровень макета; 8 – сконструированная система;
9 – оценивание заказчиком

В большинстве случаев движение по спирали продолжается, с каждым шагом продвигая разработчиков к более общей модели системы. В каждом цикле по спирали требуется конструирование (нижний правый квадрант), которое может быть реализовано классическим жизненным циклом или макетированием. Заметим, что количество действий по разработке (происходящих в правом нижнем квадранте) возрастает по мере продвижения от центра спирали.

Достоинства спиральной модели:

- наиболее реально (в виде эволюции) отображает разработку ПО;
- позволяет явно учитывать риск на каждом витке эволюции разработки;
- включает шаг системного подхода в итерационную структуру разработки;
- использует моделирование для уменьшения риска и совершенствования программного изделия.

Недостатки спиральной модели:

- новизна (отсутствует достаточная статистика эффективности модели);
- повышенные требования к заказчику;
- трудности контроля и управления временем разработки.

1.4.3. V-образная модель

Основное назначение V-образной модели – обеспечение планирования тестирования (испытаний) системы и программного средства на ранних стадиях проекта.

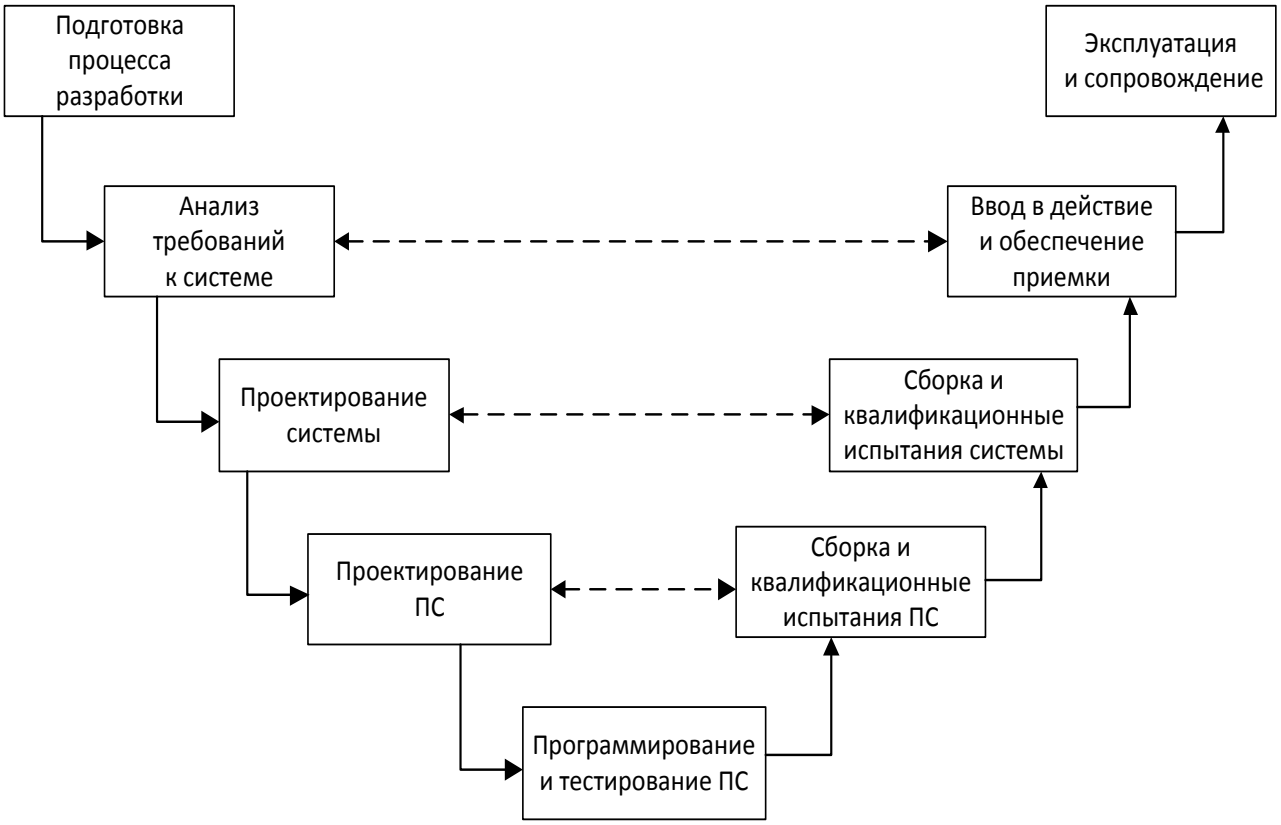

Рис. 1.7. V-образная модель жизненного цикла

Данная модель поддерживает каскадную стратегию однократного выполнения этапов процесса разработки ПС или систем и базируется на предварительном полном формировании требований.

В классической V-образной модели (см. рис. 1.7) каждый шаг начинается после завершения предыдущего шага. Отличием V-образной модели от каскадной является то, что в ней выделены связи между шагами, предшествующими программированию, и соответствующими видами тестирования и испытаний.

Таким образом, в конце каждого этапа жизненного цикла разработки, а зачастую и в процессе выполнения этапа, осуществляется проверка взаимной корректности требований различных уровней. Данная модель позволяет более оперативно проверять корректность разработки, однако, как и в каскадной модели предполагается, что на каждом этапе разрабатываются документы, описывающие поведение всей системы в целом.

При подходящем использовании V-образная модель обладает следующими **достоинствами**:

- планированием тестирования и испытаний на ранних стадиях разработки системы и программного средства;
- упрощением аттестации и верификации промежуточных результатов разработки;
- упрощением управления и контроля хода процесса разработки.

К **недостаткам** V-образной модели можно отнести:

- поздние сроки тестирования требований в жизненном цикле, что оказывает существенное влияние на график выполнения проекта при необходимости изменения требований;
- отсутствие, как и в остальных каскадных моделях, действий, направленных на анализ рисков.

1.4.4. Макетирование программных средств

Достаточно часто заказчик не может сформулировать подробные требования по вводу, обработке или выводу данных для будущего программного продукта. С другой стороны, разработчик может сомневаться в приспосабливаемости продукта под операционную систему, форме диалога с пользователем или в эффективности реализуемого алгоритма. В этих случаях целесообразно использовать макетирование.

Основная цель макетирования – снять неопределенности в требованиях заказчика.

Макетирование (прототипирование) – это процесс создания модели требуемого программного продукта.

Модель может принимать одну из трех форм:

- бумажный макет или макет на основе ПК (изображает или рисует человеко-машинный диалог);
- работающий макет (выполняет некоторую часть требуемых функций);
- существующую программу (характеристики которой затем должны быть улучшены).

Как показано на рис. 1.8, макетирование основывается на многократном повторении итераций, в которых участвуют заказчик и разработчик.

Макетирование начинается со сбора и уточнения требований к создаваемому программному средству (рис. 1.9). Разработчик и заказчик встречаются и определяют все цели ПО, устанавливают, какие требования известны, а какие предстоит доопределить.

Затем выполняется быстрое проектирование. В нем внимание сосредоточивается на тех характеристиках ПО, которые должны быть видимы пользователю.

Рис. 1.8. Макетирование программного средства

Быстрое проектирование приводит к построению макета.

Макет оценивается заказчиком и используется для уточнения требований к программному средству.

Итерации повторяются до тех пор, пока макет не выявит все требования заказчика и, тем самым, не даст возможность разработчику понять, что должно быть сделано.

Главным **достоинством** макетирования является то, что такой подход к разработке обеспечивает определение полных требований к ПО.

Основным **недостатком** макетирования можно считать тот факт, что часто либо заказчик, либо разработчик ошибочно принимают макет за программный продукт.

Поясним суть недостатков. Когда заказчик видит работающую версию ПО, он перестает сознавать, что детали макета скреплены «жевательной резинкой и проволокой»; он забывает, что в погоне за работающим вариантом оставлены нерешенными вопросы качества и удобства сопровождения ПО. Когда заказчику говорят, что продукт должен быть перестроен, он начинает возмущаться и требовать, чтобы макет «в три приема» был превращен в рабочий продукт. Очень часто это отрицательно сказывается на управлении разработкой ПО.

С другой стороны, для быстрого получения работающего макета разработчик часто идет на определенные компромиссы. Могут использоваться не самые подходящие язык программирования или операционная система. Для простой демонстрации возможностей может применяться неэффективный алгоритм. Спустя некоторое время разработчик забывает о причинах, по которым эти средства не подходят. В результате далеко не идеальный выбранный вариант интегрируется в систему.

Рис.1.9. Последовательность действий при макетировании

Очевидно, что преодоление этих недостатков требует борьбы с распространенным соблазном – принять желаемое за действительное.

1.4.5. Быстрая разработка приложений

Модель быстрой разработки приложений (Rapid Application Development) – это один из примеров применения инкрементной стратегии конструирования (рис. 1.10).

RAD-модель обеспечивает экстремально короткий цикл разработки, который достигается на основе использования компонентно-ориентированного конструирования. Если требования полностью определены, а проектная область ограничена, RAD-процесс позволяет группе создать полностью функциональную систему за очень короткое время (60–90 дней).

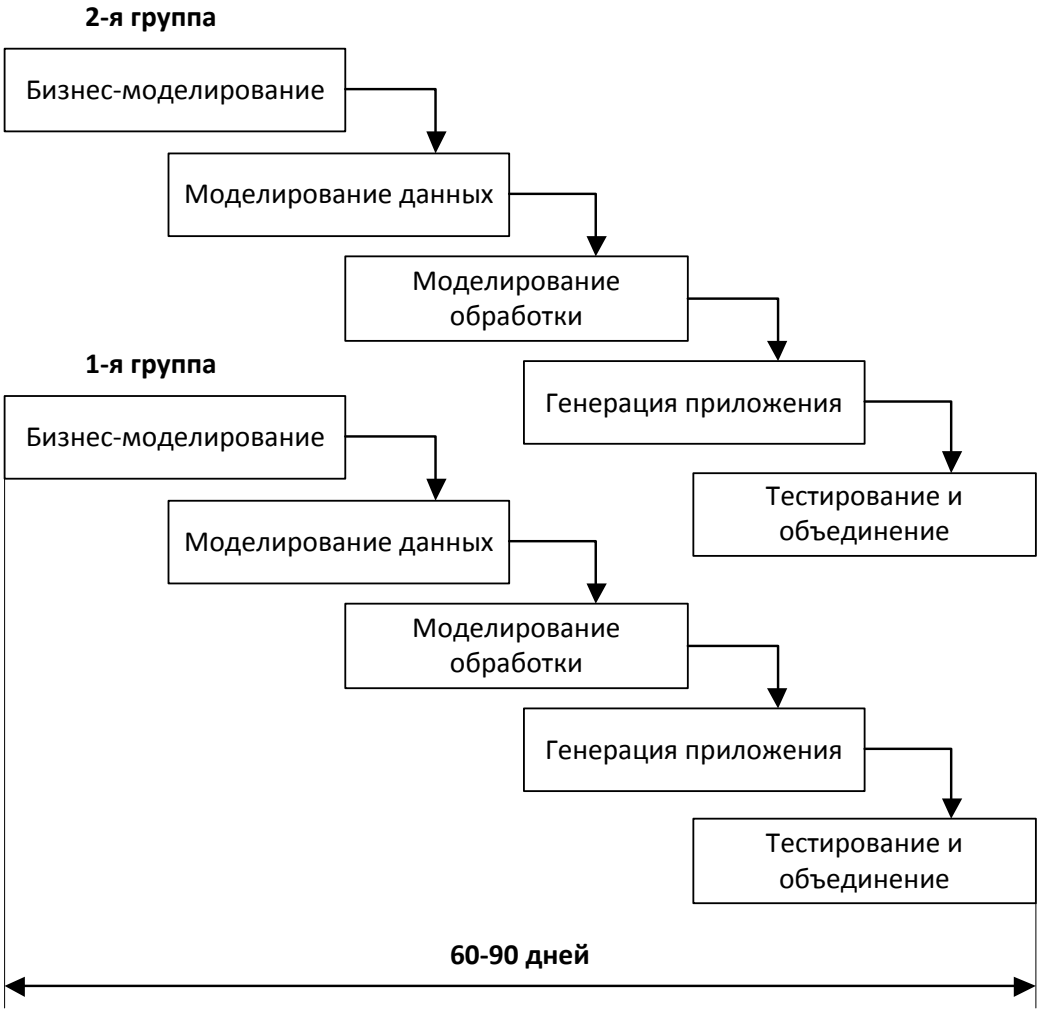

Рис. 1.10. Модель быстрой разработки приложений

RAD-подход ориентирован на разработку информационных систем и выделяет ряд этапов, к числу которых относятся следующие:

1. *Бизнес-моделирование.* Моделируется информационный поток между бизнес-функциями.

2. *Моделирование данных.* Информационный поток, определенный на этапе бизнес-моделирования, отображается в набор объектов данных, которые требуются для поддержки бизнеса. Идентифицируются характеристики (свойства, атрибуты) каждого объекта, определяются отношения между объектами.

3. *Моделирование обработки.* Определяются преобразования объектов данных, обеспечивающие реализацию бизнес-функций. Создаются описания обработки для добавления, модификации, удаления или нахождения (исправления) объектов данных.

4. *Генерация приложения.* Предполагается использование методов, ориентированных на языки программирования 4-го поколения. Вместо создания ПО с помощью языков программирования 3-го поколения, RAD-процесс работает с

повторно используемыми программными компонентами или создает повторно используемые компоненты.

5. *Тестирование и объединение.* Поскольку используемые компоненты применяются повторно, многие программные элементы уже протестированы. Это уменьшает время тестирования (хотя все новые элементы должны быть протестированы).

Применение RAD возможно в том случае, когда каждая главная функция может быть завершена за 3 месяца. Каждая главная функция адресуется отдельной группе разработчиков, а затем интегрируется в единую систему.

Применение RAD имеет как недостатки, так и ограничения.

1. Для больших проектов в RAD требуются существенные людские ресурсы (необходимо создать достаточное количество групп).

2. RAD применима только для таких приложений, которые могут декомпозироваться на отдельные модули и в которых производительность не является критической величиной.

3. RAD не применима в условиях высоких технических рисков (то есть при использовании новой технологии).

Вопросы для самопроверки

1. В чем заключаются основные особенности разработки программных средств (систем)?
2. Каковы этапы классического жизненного цикла программного средства?
3. Каковы недостатки водопадной модели разработки программного обеспечения?
4. Каковы отличия каскадной модели с обратными связями от классической каскадной модели?
5. В чем основные достоинства спиральной модели?
6. Поясните суть использования V-лбразной модели при разработке программных средств.
7. В чем заключаются основные достоинства использования макетирования при разработке программных средств?
8. Каковы основные стратегии конструирования программного средства?
9. В чем основные отличия модели быстрой разработки приложений от инкрементной модели?
10. Объясните достоинства и недостатки модели быстрой разработки приложений.

2. МОДУЛЬНОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ СРЕДСТВ

2.1. Основные понятия

Модульное проектирование является одним из первых подходов к разработке структуры ПС и уже несколько десятилетий сохраняет свои позиции в качестве классического подхода и в качестве основы для современных технологий разработки ПС.

При разработке модульных ПС могут использоваться как методы структурного проектирования, так и методы объектно-ориентированного проектирования, целью которых является формирование структуры создаваемой программы – ее разделение по некоторым установленным правилам на структурные компоненты (модуляризация) с последующей иерархической организацией данных компонентов. Для различных языков программирования такими компонентами могут быть подпрограммы, внешние модули, объекты.

Классическое определение идеальной модульной программы формулируется следующим образом.

Модульная программа – это программа, в которой любую часть логической структуры можно изменить, не вызывая изменений в ее других частях.

Признаки модульности программ:

1. Программные модули являются независимыми, что означает – модуль можно изменять или модифицировать без последствий в других модулях.

2. Условие «один вход – один выход». Модульная программа состоит из модулей, имеющих одну точку входа и одну точку выхода. В общем случае, может быть более одного входа, но важно, чтобы точки входов были определены и другие модули не могли входить в данный модуль в произвольной точке.

Достоинства модульного проектирования:

- упрощение разработки ПС;
- исключение чрезмерной детализации обработки данных;
- упрощение сопровождения ПС;
- облегчение чтения и понимания программ;
- облегчение работы с данными, имеющими сложную структуру.

Недостатки модульности:

- модульный подход требует большего времени работы центрального процессора (в среднем на 5–10 %) за счет времени обращения к модулям;
- модульность программы приводит к увеличению ее объема (в среднем на 5–10 %);
- модульность требует дополнительной работы программиста и определенных навыков проектирования ПС.

Классические методы структурного проектирования модульных ПС делятся на три основные группы:

- методы нисходящего проектирования;
- методы расширения ядра;
- методы восходящего проектирования.

На практике обычно применяются различные сочетания этих методов.

2.2. Показатели качества декомпозиции программы на модули

Для оценки корректности и эффективности структурного разбиения программы на модули необходимо оценить характеристики получившихся модулей. Существуют различные меры оценки характеристик модулей. Ниже рассматриваются две из них – связность и сцепление.

2.2.1. Связность модуля

Связность модуля – это мера прочности соединения функциональных и информационных объектов внутри одного модуля. Она характеризует степень взаимосвязи элементов, реализуемых одним модулем. Размещение сильно связанных элементов в одном модуле уменьшает межмодульные связи и, соответственно, взаимовлияние модулей. В то же время помещение сильно связанных элементов в разные модули не только усиливает межмодульные связи, но и усложняет понимание их взаимодействия. Объединение слабо связанных элементов также уменьшает технологичность модулей, так как такими элементами мысленно манипулировать сложнее.

Различают следующие виды связности (в порядке убывания уровня):

- функциональную;
- последовательную;
- информационную (коммуникативную);
- процедурную;
- временную;
- логическую;
- случайную.

При *функциональной связности* все объекты модуля предназначены для выполнения одной функции. Модуль, элементы которого связаны функционально, имеет четко определенную цель – при его вызове выполняется одна задача, например, подпрограмма поиска минимального элемента массива. Такой модуль имеет максимальную связность, следствием которой являются его хорошие технологические качества: простота тестирования, модификации и сопровождения. Именно с этим связано одно из требований структурной декомпозиции «один модуль – одна функция».

При ***последовательной*** *связности* функций выход одной функции служит исходными данными для другой функции. Как правило, такой модуль имеет одну точку входа, т.е. реализует одну подпрограмму, выполняющую две функции. Считают, что данные, используемые последовательными функциями, также связаны последовательно. Модуль с последовательной связностью функций можно разбить на два или более модулей, как с последовательной, так и с функциональной связностью. Такой модуль выполняет несколько функций, и, следовательно, его технологичность хуже: сложнее организовать тестирование, а при выполнении модификации мысленно приходится разделять функции модуля.

Информационно *связанными* считают функции, обрабатывающие одни и те же данные. При использовании структурных языков программирования раздельное выполнение функций можно осуществить только, если каждая функция реализуется своей подпрограммой.

Процедурно *связаны* функции или данные, которые являются частями одного процесса. Обычно модули с процедурной связностью функций получают в случае, если в модуле объединены функции альтернативных частей программы. При процедурной связности отдельные элементы модуля связаны крайне слабо, так как реализуемые ими действия связаны лишь общим процессом, следовательно, технологичность данного вида связи ниже, чем предыдущего.

Временная связность функций подразумевает, что эти функции выполняются параллельно или в течение некоторого периода времени, и означает, что они используются в некотором временном интервале. Например, временную связность имеют функции, выполняемые при инициализации некоторого процесса. Отличительной особенностью временной связности является то, что действия, реализуемые такими функциями, обычно могут выполняться в любом порядке. Содержание модуля с временной связностью функций имеет тенденцию меняться: в него могут включаться новые действия и/или исключаться старые. Большая вероятность модификации функции еще больше уменьшает показатели технологичности модулей данного вида по сравнению с предыдущим.

Логическая связь базируется на объединении данных или функций в одну логическую группу. В качестве примера можно привести функции обработки текстовой информации или данные одного и того же типа. Модуль с логической связностью функций часто реализует альтернативные варианты одной операции, например, сложение целых чисел и сложение вещественных чисел. Из такого модуля всегда будет вызываться одна какая-либо его часть, при этом вызывающий и вызываемый модули будут связаны по управлению. Понять логику работы модулей, содержащих логически связанные компоненты, как правило, сложнее, чем модулей, использующих временную связность, следовательно, показатели их технологичности окажутся еще ниже.

В том случае, если связь между элементами мала или отсутствует, считают, что они имеют ***случайную связность.***

Модуль, элементы которого связаны случайно, имеет самые низкие показатели технологичности, так как элементы, объединенные в нем, вообще не связаны.

Обратите внимание, что в трех предпоследних случаях связь между несколькими подпрограммами в модуле обусловлена внешними причинами. А в последнем – отсутствует вообще. Это соответствующим образом проецируется на технологические характеристики модулей. Как правило, при хорошо продуманной декомпозиции модули верхних уровней иерархии имеют функциональную или последовательную связность функций и данных. Для модулей обслуживания данных характерна информационная связность функций. Данные таких модулей могут быть связаны по-разному. Так, модули, содержащие описание классов при объектно-ориентированном подходе, характеризуются информационной связностью методов и функциональной связностью данных. Получение в процессе декомпозиции модулей с другими видами связности, скорее всего, означает недостаточно продуманное проектирование. Исключением являются лишь библиотеки ресурсов.

2.2.2. Сцепление модулей

Сцепление является мерой взаимозависимости модулей, которая определяет, насколько хорошо модули отделены друг от друга. Модули независимы, если каждый из них не содержит никакой информации о другом. Чем больше информации о других модулях хранит модуль, тем крепче он с ними сцеплен.

Различают пять типов сцепления модулей:

- по данным;
- по образцу;
- по управлению;
- по общей области данных;
- по содержимому.

Сцепление ***по данным*** предполагает, что модули обмениваются данными, представленными скалярными значениями. При небольшом количестве передаваемых параметров этот тип обеспечивает наилучшие технологические характеристики программного обеспечения.

Сцепление ***по образцу*** предполагает, что модули обмениваются данными, объединенными в структуры. Этот тип также обеспечивает неплохие характеристики, но они хуже, чем у предыдущего типа, так как конкретные передаваемые данные «спрятаны» в структуры, и потому уменьшается «прозрачность» связи

между модулями. Кроме того, при изменении структуры передаваемых данных необходимо модифицировать все использующие ее модули.

При *сцеплении* ***по управлению*** один модуль посылает другому некоторый информационный объект (флаг), предназначенный для управления внутренней логикой модуля. Таким способом часто выполняют настройку режимов работы программного обеспечения. Подобные настройки также снижают наглядность взаимодействия модулей и потому обеспечивают еще худшие характеристики технологичности разрабатываемого программного обеспечения по сравнению с предыдущими типами связей.

Сцепление ***по общей области данных*** предполагает, что модули работают с общей областью данных. Этот тип сцепления считается недопустимым, поскольку:

- программы, использующие данный тип сцепления, очень сложны для понимания при сопровождении программного обеспечения;
- ошибка одного модуля, приводящая к изменению общих данных, может проявиться при выполнении другого модуля, что существенно усложняет локализацию ошибок;
- при ссылке к данным в общей области модули используют конкретные имена, что уменьшает гибкость разрабатываемого программного обеспечения.

Следует иметь в виду, что «подпрограммы с памятью», действия которых зависят от истории вызовов, используют сцепление по общей области, что делает их работу в общем случае непредсказуемой. Именно этот вариант используют статические переменные C и C++.

В случае *сцепления* ***по содержимому*** один модуль содержит обращения к внутренним компонентам другого (передает управление внутрь, читает и/или изменяет внутренние данные или сами коды), что полностью противоречит блочно-иерархическому подходу. Отдельный модуль в этом случае уже не является блоком («черным ящиком»): его содержимое должно учитываться в процессе разработки другого модуля. Современные универсальные языки процедурного программирования, например Pascal, данного типа сцепления в явном виде не поддерживают, но для языков низкого уровня, например Ассемблера, такой вид сцепления остается возможным.

Как правило, модули сцепляются между собой несколькими способами. Учитывая это, качество программного обеспечения принято определять по типу сцепления с худшими характеристиками. Так, если использовано сцепление по данным и сцепление по управлению, то определяющим считают сцепление по управлению.

В некоторых случаях сцепление модулей можно уменьшить, удаляя необязательные связи и структурируя необходимые связи. Примером может служить объектно-ориентированное программирование, в котором вместо большого количества параметров метод неявно получает адрес области (структуры), в которой расположены поля объекта, и явно – дополнительные параметры. В результате модули оказываются сцепленными по образцу.

2.3. Методы нисходящего проектирования

Основное назначение нисходящего проектирования – служить средством декомпозиции сложной задачи на ряд независимых, меньшей сложности.

Суть метода нисходящего проектирования заключается в следующем.

На начальном шаге в соответствии с общими функциональными требованиями к программному средству разрабатывается его укрупненная структура без детальной проработки его отдельных частей. Затем выделяются функциональные требования более низкого уровня и в соответствии с ними разрабатываются отдельные компоненты программного средства, не детализированные на предыдущем шаге. Эти действия являются рекурсивными, то есть каждый из компонентов детализируется до тех пор, пока его составные части не будут окончательно уточнены. В последнем случае принимается решение о прекращении дальнейшего проектирования.

На каждом шаге нисходящего проектирования делается оценка правильности вносимых уточнений в контексте правильности функционирования разрабатываемого программного средства в целом.

Компоненты нижнего уровня ПС называются *программными модулями*. Для модулей характерны достаточная простота и прозрачность, позволяющие выполнять их непосредственное программирование.

Таким образом, на каждом шаге разработки уточняется реализация фрагмента алгоритма, то есть решается более простая задача.

Основными классическими стратегиями, на которых основана реализация метода нисходящего проектирования, являются:

- пошаговое уточнение; данная стратегия разработана Э. Дейкстрой;
- анализ сообщений; данная стратегия базируется на работах группы авторов (Йордана, Константайна, Мейерса).

Эти стратегии отличаются способами определения начальных спецификаций требований, методами разбиения задачи на части и правилами записи (нотациями), положенными в основу проектирования ПС.

2.3.1. Пошаговое уточнение

Пошаговое уточнение является одной из классических стратегий, реализующих метод нисходящего проектирования ПС. На очередном этапе декомпозиции детализируются программные компоненты очередного более низкого уровня. При этом результаты каждого этапа являются уточнением результатов предыдущего этапа лишь с небольшими изменениями.

Существуют различные способы реализации пошагового уточнения, наиболее распространенными из которых являются следующие:

- проектирование программного средства с помощью псевдокода и управляющих конструкций структурного программирования;
- использование комментариев для описания обработки данных.

Пошаговое уточнение требует, чтобы взаимное расположение строк текста программы обеспечивало ее читабельность. Служебные слова, которыми начинается и заканчивается та или иная управляющая конструкция, записываются на одной вертикали; все вложенные в данную конструкцию псевдокоды (или комментарии, или операторы программы) и управляющие конструкции записываются с отступом вправо.

Достоинства метода пошагового уточнения:

- основное внимание при его использовании обращается на проектирование корректной структуры программы, а не на ее детализацию;
- так как каждый последующий этап является уточнением предыдущего лишь с небольшими изменениями, то легко может быть выполнена проверка корректности процесса разработки на всех этапах.

Недостаток метода пошагового уточнения:

на поздних этапах проектирования программного средства может обнаружиться необходимость в структурных изменениях, требующих пересмотра более ранних решений.

2.3.2. Проектирование программных средств с помощью псевдокода и управляющих конструкций структурного программирования

Одним из классических способов реализации пошагового уточнения является проектирование ПС с помощью псевдокода и управляющих конструкций структурного программирования.

При использовании данного способа разбиение программы на модули осуществляется эвристическим способом.

На каждом этапе проектирования осуществляется *выбор необходимых управляющих конструкций*, но *операции с данными по возможности не*

уточняются (это откладывается на возможно более поздние сроки). Таким образом, фактически проектируется управляющая структура программы.

На этапе, когда принимается решение о прекращении дальнейшего уточнения, оставшиеся неопределенными функции становятся вызываемыми модулями или подпрограммами, а проектируемый модуль – управляющим модулем.

Пример

Рассмотрим пример проектирования программы с использованием псевдокода и управляющих конструкций структурного программирования. Пусть программа обрабатывает файл дат. Необходимо отсортировать правильные даты, отделив их от неправильных, перенести летние и зимние даты в выходной файл, вывести неправильные даты.

В данном примере в качестве псевдокода используются предложения, состоящие из русских слов, соединенных между собой символом подчеркивания.

Первый этап пошагового уточнения. Задается заголовок программы, соответствующий ее назначению, например:

```
Program Обработка_дат.
```

Второй этап пошагового уточнения. Определяются основные структурные компоненты программы в соответствии с ее основными функциями, например:

```
Program Обработка_дат;
   Отделить правильные_даты_от_неправильных {*}
   Сортировать_правильные_даты
   Выделить_зимние_и_летние_даты
   Обработать_неправильные_даты
End.
```

Фрагмент программы, детализированный на втором этапе, располагается правее детализированного на первом этапе.

Третий этап пошагового уточнения. Дальнейшая детализация программы (детализация фрагмента {*}).

На данном этапе возможно появление необходимости в использовании управляющих конструкций структурного программирования.

Например:

```
Program Обработка дат;
   While не конец входного файла Do
```

```
    Begin
      Прочитать_дату
      Проанализировать_правильность_даты
    End
    Сортировать_правильные_даты
    Выделить_зимние_и_летние_даты
    Обработать_неправильные_даты
  End.
```

Жирным шрифтом здесь выделены служебные слова цикла с предусловием **While**, представляющего собой одну из управляющих конструкций структурного программирования.

На некоторых этапах уточнения можно приостановить определение некоторых функций, выделяя их в вызываемые модули в том случае, если они функционально независимы от основной обработки (например, в примере это могут быть функции «Проанализировать правильность даты» и «Сортировать правильные даты»).

Процесс детализации продолжается, пока не будет принято решение о прекращении дальнейшей детализации. В этом случае все действия, записанные в последней программе в виде предложений, необходимо оформить в виде подпрограмм. Такие подпрограммы могут быть реализованы, например, с использованием принципа структурного программирования.

2.3.3. Анализ сообщений

Анализ сообщений является второй из рассматриваемых классических стратегий, реализующих метод нисходящего проектирования. Анализ сообщений используется в первую очередь для структуризации ПС обработки информации и основывается на анализе потоков данных, обрабатываемых программным средством.

Пример

Рассмотрим тот же пример проектирования программы обработки файла дат, который был рассмотрен выше. Необходимо отделить правильные даты от неправильных, отсортировать правильные, перенести летние и зимние даты в выходной файл, вывести неправильные даты.

Рис. 2.1 иллюстрирует возможный вариант укрупненного представления потоков информации и обрабатывающих их процессов для данной программы, представленный с помощью диаграмм потоков данных (Data Flow Diagram, DFD) в нотации *Йордана – Де Марко*.

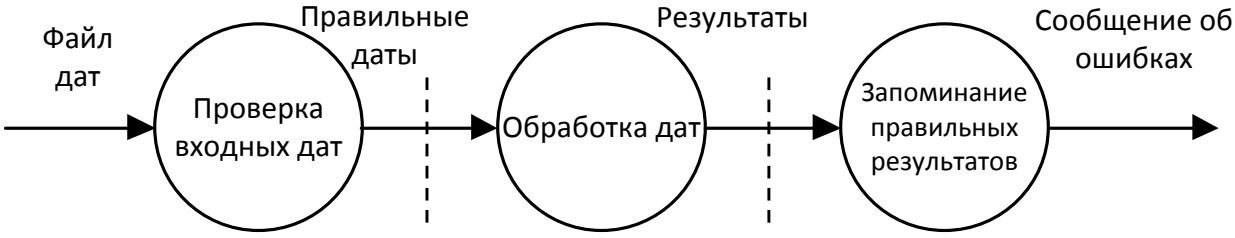

Рис. 2.1. Укрупненная диаграмма потоков данных для программы обработки файла дат

В соответствии со стратегией анализа сообщений первоначальный поток данных разбивается на три потока: первый содержит непреобразованные входные данные, второй – потоки преобразования, третий – только выходную информацию. Границы, разделяющие эти потоки, на рисунке показаны штриховыми линиями. Они делят диаграмму на три части.

Процесс «Обработка дат» в общем случае может включать различные виды преобразования данных (например, кодирование, декодирование, вычисления и др.). Результаты данного процесса представляют собой выходные данные, хотя они могут быть еще неотформатированными, неотредактированными, а, возможно, и неверными.

Три части программы, соответствующие трем потокам данных, принято называть соответственно *истоком, преобразователем* и *стоком*.

Преобразователь – это основная часть программы, ***исток*** выполняет функцию управления входным потоком данных, а ***сток*** – функцию управления выходным потоком данных.

На рис. 2.2 представлен общий вид DFD-диаграммы разбиения любой программы на исток→преобразователь→сток. Линии на диаграмме показывают потоки передачи данных между процессами.

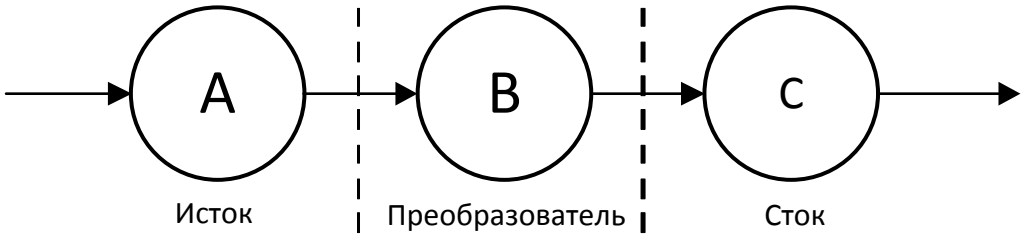

Рис. 2.2. DFD-диаграмма разбиения программы на исток→преобразователь→сток

Процесс декомпозиции программы заключается в рекурсивном использовании метода разбиения на исток→преобразователь→сток на отдельных ветвях ее древовидной структуры. На нижнем уровне в результате формируется совокупность программных модулей – наименьших единиц проектирования программы. Каждый из модулей может быть реализован в

зависимости от назначения, сложности и размера как независимый модуль, внутренняя подпрограмма или некоторая часть основной программы.

Следует отметить, что не все компоненты программы обязательно должны быть разбиты на три компонента более низкого уровня. Результат декомпозиции компонента-стока должен обязательно содержать сток, компонента-преобразователя – преобразователь, компонента-истока – исток. Вызывающий компонент – это главный сток для компонента-истока и главный исток для компонента-стока.

Если структуры данных имеют большие размеры, управление ими осуществляется на уровне детализации данных для компонентов истока и стока. Количество уровней декомпозиции определяется сложностью задачи и необходимой степенью детализации.

Каждый компонент программы при информационном обмене использует определенную часть данных. Однако в иерархической структуре программы информационные связи между компонентами не отражены. Поэтому описание иерархической структуры должно содержать *таблицу взаимодействия компонентов*, показывающую передачу данных между ними. В этой таблице должны быть определены все способы информационного обмена, задаваемые как при помощи формальных параметров, так и с помощью глобальных переменных.

2.4. Методы восходящего проектирования

При использовании восходящего проектирования в первую очередь выделяются функции нижнего уровня, которые должно выполнять программное средство. Эти функции реализуются с помощью программных модулей самых нижних уровней. Затем на основе этих модулей проектируются программные компоненты более высокого уровня. Данные компоненты реализуют функции более высокого уровня. Процесс продолжается, пока не будет завершена разработка всего программного средства.

В чистом виде метод восходящего проектирования используется крайне редко. Основным его **недостатком** является то, что программисты начинают разработку программного средства с несущественных, вспомогательных деталей. Это затрудняет проектирование программного средства в целом.

Метод восходящего проектирования целесообразно применять в следующих случаях:

- существуют разработанные модули, которые могут быть использованы для выполнения некоторых функций разрабатываемой программы;

- заранее известно, что некоторые простые или стандартные модули потребуются нескольким различным частям программы (например, подпрограмма анализа ошибок, ввода-вывода и т.п.).

Обычно используется сочетание методов нисходящего и восходящего проектирования. Такое сочетание возможно различными способами.

Первый способ сочетания. Выделяются ключевые (наиболее важные) модули промежуточных уровней разрабатываемой программы. Затем проектирование ведется нисходящим и восходящим методами одновременно.

Второй способ сочетания. Проектируются модули нижнего уровня (те, которые необходимо спроектировать заранее). Затем программа проектируется одновременно нисходящим и восходящим методами.

При таком способе проектирования наиболее важной задачей является согласование интерфейса между верхними и нижними уровнями программы, выполняемое в последнюю очередь. Это является существенным недостатком данного способа сочетания. Разработчики должны обладать достаточно высокой квалификацией, чтобы не оказалось, что верхняя и нижняя части программы несовместимы между собой.

2.5. Методы расширения ядра

При использовании данных методов в первую очередь создается ядро (основная часть) программы. Затем данное ядро постепенно расширяется, пока не будет полностью сформирована управляющая структура разрабатываемой программы.

Существует два подхода к реализации методов расширения ядра.

Первый подход основан на методах проектирования структур данных, используемых при иерархическом проектировании модулей.

Второй подход основан на определении областей хранения данных с последующим анализом связанных с ними функций. Данный подход использует метод определения спецификаций модуля.

Вопросы для самопроверки

1. Дайте определение модульной программы.
2. Перечислите признаки модульности программ.
3. Назовите основные достоинства и недостатки модульного проектирования.
4. Дайте классификацию классических методов структурного проектирования модульных программных средств.
5. Поясните сущность методов нисходящего проектирования.
6. Поясните сущность и назовите способы реализации стратегии пошагового уточнения.
7. Поясните сущность методов восходящего проектирования.
8. Перечислите и охарактеризуйте способы сочетания методов нисходящего и восходящего проектирования.
9. Что такое связность модуля?
10. Назовите и охарактеризуйте типы связности модулей.
11. Что такое сцепление модулей?
12. Назовите и охарактеризуйте типы и степени сцепления модулей.

3. ПРОЕКТИРОВАНИЕ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ

Прежде чем говорить об основных этапах создания программы, необходимо напомнить о принципах, которыми должен руководствоваться каждый программист. Очень часто программисты упрямо считают, что их основной целью является изобретение новых изощренных алгоритмов, а не выполнение полезной работы, и напоминают этим печально известную точку зрения «магазин – для продавца, а не для покупателя». Надо помнить о том, что программист, в конечном счете, всегда работает для пользователя программы и является членом коллектива, который должен обеспечить создание надежной программы в установленный срок.

3.1. Цели и этапы проектирования

Структурный подход к программированию, как уже упоминалось, охватывает все стадии разработки проекта: спецификацию, проектирование, собственно программирование и тестирование. Задачи, которые при этом ставятся, это уменьшение числа возможных ошибок путем применения только допустимых структур, возможно, более раннее обнаружение ошибок и упрощение процесса их исправления. Ключевыми идеями структурного подхода являются нисходящая разработка, структурное программирование и нисходящее тестирование.

Приведенные ниже этапы создания программ рассчитаны на достаточно большие проекты, разрабатываемые коллективом программистов. Для программы небольшого объема каждый этап упрощается, но содержание и последовательность этапов не изменяются.

I этап. Постановка задачи. Создание любой программы начинается с постановки задачи. Изначально задача ставится в терминах предметной области, и необходимо перевести ее в термины, более близкие к программированию. Поскольку программист редко досконально разбирается в предметной области, а заказчик – в программировании (простой пример: требуется написать бухгалтерскую программу), постановка задачи может стать весьма непростым итерационным процессом. Кроме того, при постановке задачи заказчик зачастую не может четко и полно сформулировать свои требования и критерии. В качестве иллюстрации приведем карикатуру «Качели» (рис. 3.1), которая появилась в 1973 году в информационном бюллетене вычислительного центра Лондонского университета и сразу стала широко известной, поскольку очень точно отражала процесс создания программы.

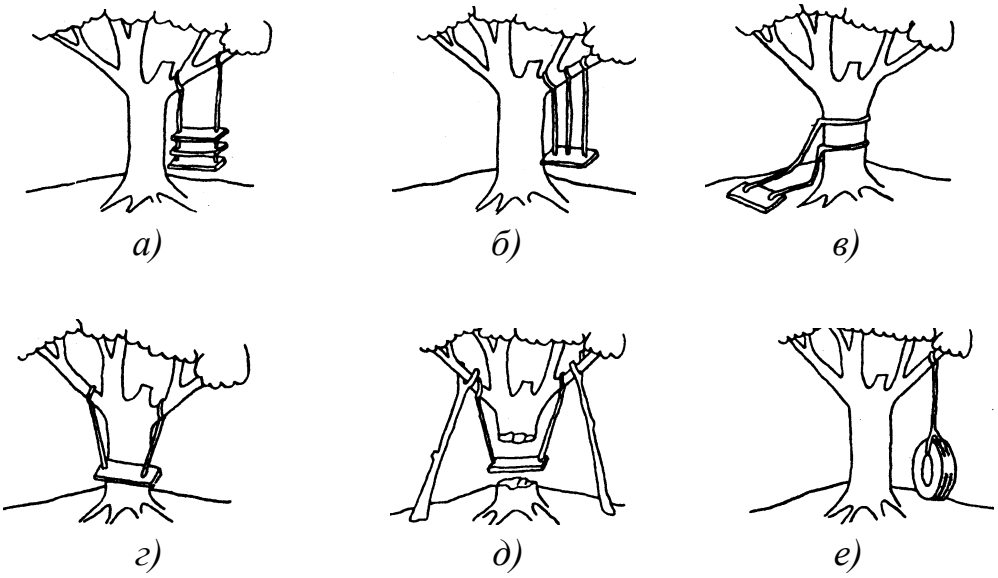

Рис. 3.1. «Качели»:
а) как было предложено организатором разработки; б) как было описано в техническом задании; в) как было спроектировано ведущим специалистом; г) как было реализовано программистами; д) как было внедрено; е) что хотел пользователь

Постановка задачи завершается созданием технического задания, а затем внешней спецификации программы, включающей в себя:

- описание исходных данных и результатов (типов, форматов, точности, способа передачи, ограничений);
- описание задачи, реализуемой программой;
- способ обращения к программе;
- описание возможных аварийных ситуаций и ошибок пользователя.

Таким образом, программа рассматривается как черный ящик, для которого определена функция и входные и выходные данные.

II этап. Разработка внутренних структур данных. Большинство алгоритмов зависит от того, каким образом организованы данные, поэтому интуитивно ясно, что начинать проектирование программы надо не с алгоритмов, а с разработки структур, необходимых для представления входных, выходных и промежуточных данных. При этом принимаются во внимание многие факторы, например ограничения на размер данных, необходимая точность, требования к быстродействию программы. Структуры данных могут быть статическими или динамическими.

III этап. Проектирование (определение общей структуры и взаимодействия модулей). На этом этапе применяется технология нисходящего проектирования программы, основная идея которого теоретически проста: разбиение задачи на подзадачи меньшей сложности, которые можно рассматривать раз-

дельно. При этом используется метод пошаговой детализации. Можно представить себе этот процесс так, что сначала программа пишется на языке некоторой гипотетической машины, которая способна понимать самые обобщенные действия, а затем каждое из них описывается на более низком уровне абстракции, и т.д. Очень важной на этом этапе является спецификация интерфейсов, то есть способов взаимодействия подзадач.

Для каждой подзадачи составляется внешняя спецификация, аналогичная приведенной выше. На этом же этапе решаются вопросы разбиения программы на модули. Главный критерий декомпозиции – минимизация их взаимодействия. Одна задача может реализовываться с помощью нескольких модулей, и, наоборот, в одном модуле может решаться несколько задач. На более низкий уровень проектирования переходят только после окончания проектирования верхнего уровня. Алгоритмы записывают в обобщенной форме, например словесной, в виде обобщенных блок-схем или другими способами. Если возникают трудности с записью алгоритма, то он, скорее всего, плохо продуман.

На этапе проектирования следует учитывать возможность будущих модификаций программы и стремиться проектировать программу таким образом, чтобы вносить изменения было возможно проще. Поскольку неизвестно, какие изменения придется выполнить, это пожелание напоминает создание «общей теории всего»; на практике надо ограничиться разумными компромиссами. Программист, исходя из своего опыта и здравого смысла, решает, какие именно свойства программы может потребоваться изменить или усовершенствовать в будущем.

Процесс проектирования является итерационным, поскольку в программах значительного размера невозможно продумать и учесть все детали с первого раза.

IV этап. Структурное программирование. Процесс программирования также организуется по принципу «сверху вниз»: вначале кодируются модули самого верхнего уровня и составляются тестовые примеры для их отладки, при этом на месте еще не написанных модулей следующего уровня ставятся «заглушки» – временные программы. «Заглушки» в простейшем случае просто выдают сообщение о том, что им передано управление, а затем возвращают его в вызывающий модуль.

В других случаях «заглушка» может выдавать значения, заданные заранее или вычисленные по упрощенному алгоритму.

Таким образом, сначала создается логический скелет программы, который затем обрастает плотью кода.

Казалось бы, более логично применять к процессу программирования восходящую технологию – написать и отладить сначала модули нижнего уровня, а

затем объединять их в более крупные фрагменты, но этот подход имеет ряд недостатков.

Во-первых, в процессе кодирования верхнего уровня могут быть вскрыты те или иные трудности проектирования более низких уровней программы (просто потому, что при написании программы ее логика продумывается более тщательно, чем при проектировании). Если подобная ошибка обнаруживается в последнюю очередь, то требуются дополнительные затраты на переделку уже готовых модулей нижнего уровня.

Во-вторых, для отладки каждого модуля, а затем более крупных фрагментов программы требуется каждый раз составлять свои тестовые примеры, и программист часто вынужден имитировать то окружение, в котором должен работать модуль.

Нисходящая же технология программирования обеспечивает естественный порядок создания тестов – возможность нисходящей отладки, которая рассмотрена далее.

При программировании следует отделять интерфейс (функции, модуля, класса) от его реализации и ограничивать доступ к ненужной информации. Небрежное (даже в мелочах) программирование может привести к огромным затратам на поиск ошибок на этапе отладки.

Этапы проектирования и программирования совмещены во времени: в идеале сначала проектируется и кодируется верхний уровень, затем следующий и т.д. Такая стратегия применяется потому, что в процессе кодирования может возникнуть необходимость внести изменения, отражающиеся на модулях нижнего уровня.

V этап. Нисходящее тестирование. Этот этап записан последним, отнюдь не означая, что тестирование не должно проводиться на предыдущих этапах. Проектирование и программирование должны обязательно сопровождаться написанием набора тестов – проверочных исходных данных и соответствующих им наборов эталонных реакций.

Необходимо различать процессы тестирования и отладки программы.

Тестирование – это процесс, посредством которого проверяется правильность программы. Тестирование носит позитивный характер, его цель – показать, что программа работает правильно и удовлетворяет всем проектным спецификациям.

Отладка – это процесс исправления ошибок в программе, при этом цель исправить все ошибки не ставится. Исправляют ошибки, обнаруженные при тестировании. При планировании следует учитывать, что процесс обнаружения ошибок подчиняется закону насыщения, то есть большинство ошибок обнару-

живается на ранних стадиях тестирования, и чем меньше в программе осталось ошибок, тем дольше искать каждую из них.

Для исчерпывающего тестирования программы необходимо проверить каждую из ветвей алгоритма. Общее число ветвей определяется комбинацией всех альтернатив на каждом этапе. Это – конечное число, но оно может быть очень большим, поэтому программа разбивается на фрагменты, после исчерпывающего тестирования которых они рассматриваются как элементарные узлы ветвей более длинных. Кроме данных, обеспечивающих выполнение операторов в требуемой последовательности, тесты должны содержать проверку граничных условий (например, переход по условию x>10 должен проверяться для значений, которые больше, меньше и равны 10). Отдельно проверяется реакция программы на ошибочные исходные данные.

Идея нисходящего тестирования предполагает, что к тестированию программы приступают еще до того, как завершено ее проектирование. Это позволяет раньше опробовать основные межмодульные интерфейсы, а также убедиться в том, что программа в основном удовлетворяет требованиям пользователя. Только после того как логическое ядро испытано настолько, что появляется уверенность в правильности реализации основных интерфейсов, приступают к кодированию и тестированию следующего уровня программы.

Естественно, полное тестирование программы, пока она представлена в виде скелета, невозможно, однако добавление каждого следующего уровня позволяет постепенно расширять область тестирования.

Этап комплексной отладки на уровне системы при нисходящем проектировании занимает меньше времени, чем при восходящем, и приносит меньше сюрпризов, поскольку вероятность появления серьезных ошибок, затрагивающих большую часть системы, гораздо ниже. Кроме того, для каждого подключаемого к системе модуля уже создано его окружение, и выходные данные отлаженных модулей можно использовать как входные для тестирования других, что облегчает процесс тестирования. Это не значит, что модуль надо подключать к системе совсем «сырым» – бывает удобным провести часть тестирования автономно, поскольку сгенерировать на входе системы все варианты, необходимые для тестирования отдельного модуля, трудно.

При отладке активно используются средства конкретной оболочки программирования.

3.2. Анализ предметной области и выработка требований к программному средству

3.2.1. Анализ предметной области

Для того чтобы разработать программную систему, приносящую реальные выгоды определенным пользователям, необходимо сначала выяснить, какие же задачи она должна решать для этих людей и какими свойствами обладать.

Требования к ПО определяют, какие свойства и характеристики оно должно иметь для удовлетворения потребностей пользователей и других заинтересованных лиц. Однако сформулировать требования к сложной системе не так легко. В большинстве случаев будущие пользователи могут перечислить набор свойств, который они хотели бы видеть, но никто не даст гарантий, что это – исчерпывающий список. Кроме того, часто сама формулировка этих свойств будет непонятна большинству программистов – могут прозвучать фразы типа **«должно использоваться и частотное, и временное уплотнение каналов»,** или **«передача клиента должна быть мягкой»,** или **«для обычных швов отмечайте бригаду, а для доверительных – конкретных сварщиков»**, и это еще не самые тяжелые для понимания примеры.

Чтобы ПО оказалось действительно полезным, важно, чтобы оно удовлетворяло реальные потребности людей и организаций, которые часто отличаются от непосредственно выражаемых пользователями желаний. Для выявления этих потребностей, а также для выяснения смысла высказанных требований приходится проводить достаточно большую дополнительную работу, которая называется *анализом предметной области* или *бизнес-моделированием,* если речь идет о потребностях коммерческой организации. В результате этой деятельности разработчики должны научиться понимать язык, на котором говорят пользователи и заказчики, выявить цели их деятельности, определить набор задач, решаемых ими. В дополнение стоит выяснить, какие вообще задачи нужно уметь решать для достижения этих целей, выяснить свойства результатов, которые хотелось бы получить, а также определить набор сущностей, с которыми приходится иметь дело при решении этих задач. Кроме того, анализ предметной области позволяет выявить места возможных улучшений и оценить последствия принимаемых решений о реализации тех или иных функций.

После этого можно определять область ответственности будущей программной системы, – какие именно из выявленных задач будут ею решаться, при решении каких задач она может оказать существенную помощь, и чем именно. А, определив задачи ПО в рамках общей системы задач и деятельностей пользователей, можно уже более точно сформулировать требования к нему.

Анализом предметной области занимаются *системные аналитики* или *бизнес-аналитики*, которые передают полученные ими знания другим членам проектной команды, сформулировав их на языке, более понятном разработчикам. Для передачи этих знаний обычно служит некоторый набор моделей, в виде графических схем и текстовых документов.

Наиболее удобной формой представления информации при анализе предметной области являются графические диаграммы различного рода. Они позволяют достаточно быстро зафиксировать полученные знания (составить рисунок из прямоугольников и связывающих их стрелок обычно можно гораздо быстрее, чем записать соответствующий объем информации), быстро восстановить их в памяти (на рисунке за один взгляд видно гораздо больше, чем в тексте) и успешно объясняться с заказчиками и другими заинтересованными лицами.

Часто для описания поведения сложных систем и деятельности крупных организаций используются *диаграммы потоков данных (data flow diagrams)*. Эти диаграммы содержат четыре вида графических элементов: *процессы*, представляющие собой любые трансформации данных в рамках описываемой системы, *хранилища данных*, *внешние* по отношению к системе *сущности* и *потоки данных* между элементами трех предыдущих видов.

Используются несколько систем обозначений для перечисленных элементов, наиболее известны нотация *Йордана – Де Марко* и нотация *Гэйна – Сарсона*, обе предложенные в 1979 году.

Методы объектно-ориентированного анализа предназначены для обеспечения более удобной передачи информации между моделями анализируемых систем и моделями разрабатываемого ПО. В качестве графических моделей в этих методах вместо диаграмм потоков данных используются диаграммы вариантов использования, а вместо диаграмм сущностей и связей – диаграммы классов. Однако диаграммы вариантов использования несут несколько меньше информации по сравнению с соответствующими диаграммами потоков данных – на них процессы и хранилища в соответствии с принципом объединения данных и методов их обработки объединяются в варианты использования, и остаются только связи между вариантами использования и действующими лицами (аналогом внешних сущностей). Для представления остальной информации каждый вариант использования может дополняться набором разнообразных диаграмм UML – диаграммами деятельностей, диаграммами сценариев, и пр.

3.2.2. Выделение и анализ требований

После получения общего представления о деятельности и целях организаций, в которых будет работать будущая программная система, о ее предметной области, можно определить более четко, какие именно задачи система будет ре-

шать. Кроме того, важно понимать, какие из задач стоят наиболее остро и обязательно должны быть поддержаны уже в первой версии, а какие могут быть отложены до следующих версий или вообще вынесены за рамки области ответственности системы. Эта информация выявляется при анализе потребностей возможных пользователей и заказчиков.

Потребности определяются на основе наиболее актуальных проблем и задач, которые пользователи и заказчики видят перед собой. При этом требуется аккуратное выявление значимых проблем, определение того, насколько хорошо они решаются при текущем положении дел, и расстановка приоритетов при рассмотрении недостаточно хорошо решаемых, поскольку чаще всего решить сразу все проблемы невозможно.

Формулировка потребностей может быть разбита на следующие этапы.

1. Выделение 1–3 основных проблемы.

2. Определение причины возникновения проблем, оценка степени их влияния и выделение наиболее существенных из проблем, влекущих появление остальных.

3. Определение ограничения на возможные решения.

Формулировка потребностей не должна накладывать лишних ограничений на возможные решения, удовлетворяющие им. Нужно попытаться сформулировать, что именно является проблемой, а не предлагать сразу возможные решения.

Например, формулировки вроде **«система должна использовать СУБД Oracle для хранения данных», «нужно, чтобы при вводе неверных данных раздавался звуковой сигнал»** не очень хорошо описывают потребности (за исключением особых случаев, например, если СУБД Oracle уже используется для хранения других данных, которые должны быть интегрированы с рассматриваемыми, при этом ее использование становится внешним ограничением). Соответствующие потребности лучше описать так: **«нужно организовать надежное и удобное для интеграции с другими системами хранение данных», «необходимо предотвращать попадание некорректных данных в хранилище».**

При выявлении потребностей пользователей анализируются модели деятельности пользователей и организаций, в которых они работают, для выявления проблемных мест. Также используются такие приемы, как анкетирование, демонстрация возможных сеансов работы будущей системы, интерактивные опросы, где пользователям предоставляется возможность самим предложить варианты внешнего вида системы и ее работы или поменять предложенные кем-то другим, демонстрация прототипа системы и др.

После выделения основных потребностей нужно решить вопрос о разграничении области ответственности будущей системы, то есть определить, какие из потребностей надо пытаться удовлетворить в ее рамках, а какие – нет.

На основе выделенных потребностей пользователей, отнесенных к области ответственности системы, формулируются возможные *функции* будущей системы, представляющие собой услуги, предоставляемые системой и удовлетворяющие потребности одной или нескольких групп пользователей (или других заинтересованных лиц). Идеи для определения таких функций можно брать из имеющегося опыта разработчиков (наиболее часто используемый источник) или из результатов мозговых штурмов и других форм выработки идей.

Формулировка функций должна быть достаточно короткой, ясной для пользователей, без лишних деталей. Например:

1. Все данные о сделках и клиентах будут сохраняться в базе данных.
2. Статус выполнения заказа клиент сможет узнать через Интернет.
3. Система будет поддерживать до 10 000 одновременно работающих пользователей.
4. Расписание проведения ремонтных работ будет строиться автоматически.

Предлагая те или иные функции нужно уметь аккуратно оценивать их влияние на структуру и деятельность организаций, в рамках которых будет использоваться ПО. Это можно сделать, имея полученные при анализе предметной области модели их текущей деятельности.

Имея набор функций (достаточно хорошо поддерживающих решение наиболее существенных задач) которые придется осуществлять разрабатываемой системе, можно составлять требования к ней, представляющие собой детализацию работы этих функций. При этом надо учитывать, что часто ПО является частью программно-аппаратной системы, требования к которой надо преобразовать в требования к ее программной и аппаратной составляющим.

Каждое требование раскрывает детали поведения системы при выполнении ею некоторой функции в некоторых обстоятельствах. При этом часть требований происходит из потребностей и пожеланий заинтересованных лиц и решений, удовлетворяющих эти потребности и пожелания, а часть – из внешних ограничений, накладываемых на систему (например, основными законами той предметной области, в рамках которой системе придется работать, государственным законодательством, корпоративной политикой и пр.).

Еще до перехода от функций к требованиям полезно расставить приоритеты и оценить трудоемкость их реализации и рискованность. Это позволит отказаться от реализации наименее важных и наиболее трудоемких, не соответствующих бюджету проекта функций еще до их детальной проработки, а также

выявить возможные проблемные места проекта – наиболее трудоемкие и неясные из вошедших в него функций.

Наиболее широко распространенными техниками фиксации требований в настоящий момент являются структурированные текстовые документы и диаграммы *вариантов использования*.

Вариантом использования *(use case)* называют некоторый сценарий действий системы, который обеспечивает ощутимый и значимый для ее пользователей результат. На практике в виде одного варианта использования оформляется сценарий действий системы, который будет, скорее всего, неоднократно возникать во время ее использования и имеет достаточно четко определенные условия начала выполнения и завершения.

Примеры вариантов использования:

1. Покупатель в Интернет-магазине выбирает товар. Для этого он может выбрать категорию товара, фирму-изготовителя или группу таких фирм и отфильтровать оставшиеся товары по цене, габаритам и цвету. Определившись, он выбирает товар, кликая на соответствующем значке мышкой.

2. Оператор системы контроля качества газопровода ищет участки газопровода с повышенным риском возникновения аварии. Для этого он выбирает группу ранее случившихся аварий, фильтруя их по дате, нанесенному ущербу, типу аварии и запускает процедуру анализа характеристик соответствующих участков газопровода на совпадение, по крайней мере, двух характеристик (учитываются изготовитель труб и их партия, история хранения труб на складах, землепроходческая бригада, бригада сварщиков, показатели нескольких последних проведенных инспекций, показатели химической активности грунтов, наличие близлежащих предприятий, влияющих на химические и электрические характеристики грунтов). После этого на карте выделяются участки, характеристики которых также попадают под найденный «шаблон аварии».

В языке UML вариант использования изображается в виде **овала, помеченного именем представляемого варианта использования.**

Варианты использования могут быть связаны с участвующими в них *действующими лицами (actors)*, представляющими различные роли пользователей системы или внешние системы, взаимодействующие с ней.

Варианты использования могут быть связаны друг с другом тремя видами связей: *обобщением (generalization)*, *расширением (extend relationship)* и *включением (include relationship)*. Действующие лица также могут быть связаны друг с другом с помощью связей *обобщения (generalization)*.

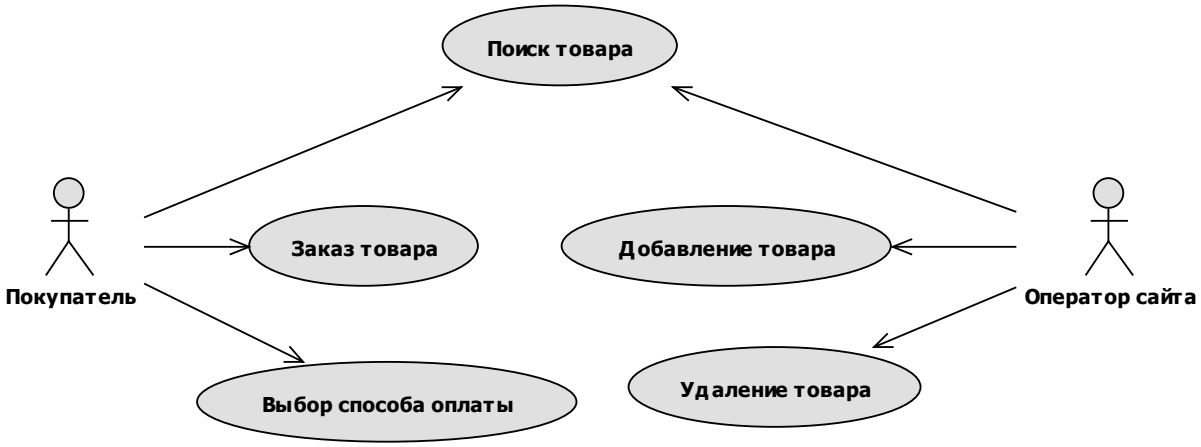

Рис. 3.2. Диаграмма вариантов использования (полученных на первых стадиях анализа) для простого Интернет-магазина

Если несколько вариантов использования имеют много общего в структуре выполняемых в их рамках сценариев и достигаемых целей, можно выделить *обобщающий* их вариант использования, содержащий общие части описываемого ими поведения. При этом в общем случае сценарий работы обобщаемого варианта состоит из нескольких кусков – последовательности действий, выполняемых в рамках сценария работы общего варианта использования, перемежаются с последовательностями, специфическими для частного. Например, если система регистрации заказов в магазине позволяет оформить заказ (данные о котором в дальнейшем будут присутствовать в системе) как при помощи сайта магазина, так и по телефону, то варианты использования **«Заказ товара через сайт»** и **«Заказ товара по телефону»** могут быть обобщены в варианте **«Заказ товара»**.

Вариант использования A *расширяет (extends)* другой вариант использования B, если в ходе сценария работы A при определенных условиях надо включить полный сценарий работы B. Например, оператор сайта магазина может удалить товар, введя его идентификатор; а если идентификатор ему не известен, а известна лишь марка товара и производитель, он должен сначала найти такой товар и определить идентификатор в его описании, а затем уже удалить товар. Соответственно, вариант использования **«Удаление товара»** будет расширять вариант использования **«Поиск товара».**

Вариант использования A *включает (includes, или использует, uses)* вариант использования B, если A всегда в некоторый момент включает полностью сценарий работы B. Например, при оформлении заказа покупатель всегда должен определить способ его оплаты. Значит, вариант использования **«Заказ товара»** включает вариант **«Определение способа оплаты».**

Обобщение между действующими лицами вводится, если задачи, решаемые одним действующим лицом с помощью данной системы, являются подмножеством задач, решаемых другим действующим лицом. Например, обычный оператор сайта может иметь права только на внесение дополнений и изменений в данные, но не иметь прав на приостановку работы сайта и изменение структуры, которые имеет администратор сайта. В то же время администратор может делать все, что может обычный оператор сайта. Соответственно, администратор сайта является специальным частным случаем оператора. Доработанная диаграмма вариантов использования системы с учетом отношений между актерами или вариантами использования представлена на рис. 3.3.

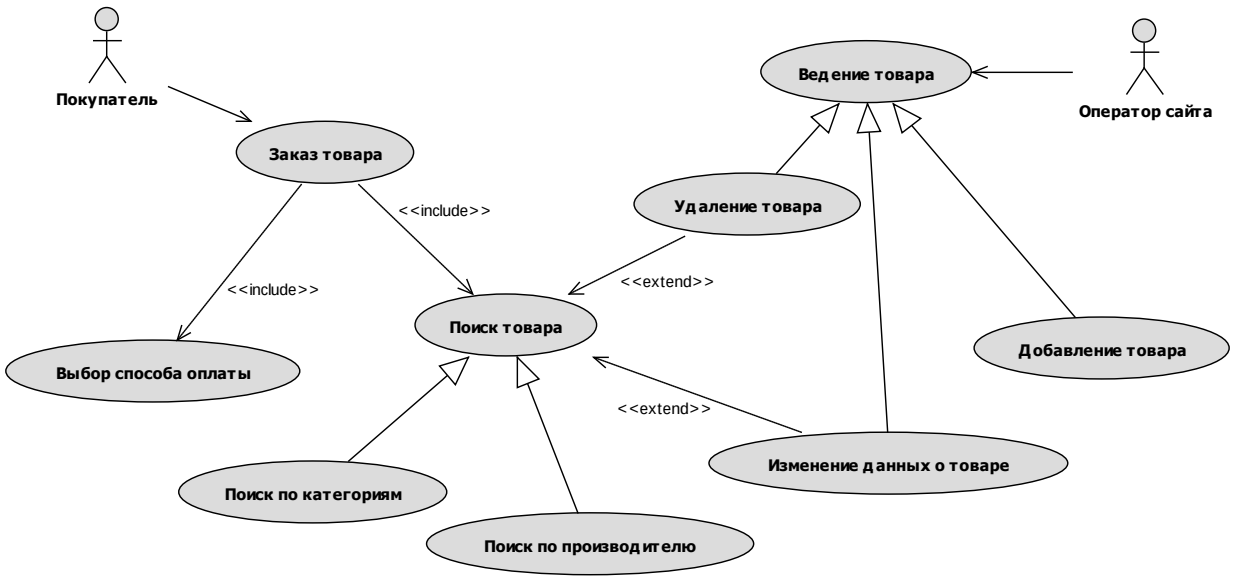

Рис. 3.3. Доработанная диаграмма вариантов использования для простого Интернет-магазина

Хорошо описанный вариант использования имеет такие атрибуты, как:

1. *Имя*, ясно говорящее о назначении варианта использования.

2. *Описание*. Несколько предложений, описывающих этот вариант использования.

3. *Частота*. Насколько часто данный вариант использования возникает.

4. *Предусловия*. Все условия запуска варианта использования.

5. *Постусловия*. Все условия, которые должны быть выполнены после успешного выполнения варианта использования.

6. *Основной сценарий работы*, который используется в большинстве случаев.

7. *Альтернативные сценарии*, возникающие иногда. Для каждого альтернативного сценария указываются условия его запуска.

Кроме того, варианты использования могут дополняться диаграммами других видов – прежде всего, сценарными диаграммами и диаграммами активно-

стей, описывающими последовательности действий участвующих компонентов, диаграммами состояний и переходов компонентов и диаграммами классов этих компонентов, и др.

3.3. Основы языка визуального проектирования объектно-ориентированных систем

Для создания моделей анализа и проектирования объектно-ориентированных программных систем используют языки визуального моделирования. Появившись сравнительно недавно, эти языки уже имеют представительную историю развития.

В настоящее время различают три поколения языков визуального моделирования. И если первое поколение образовали 10 языков, то численность второго поколения уже превысила 50 языков. Среди наиболее популярных языков второго поколения можно выделить: язык Буча (*G. Booch*), язык Рамбо (*J. Rumbaugh*), язык Джекобсона (*I. Jacobson*), язык Коада – Йордона (*Coad – Yourdon*), язык Шлеера – Меллора (*Shlaer – Mellor*) и т.д. Каждый язык вводил свои выразительные средства, ориентировался на собственный синтаксис и семантику, иными словами, претендовал на роль единственного и неповторимого языка. В результате разработчики (и пользователи этих языков) перестали понимать друг друга. Возникла острая необходимость унификации языков.

Идея унификации привела к появлению языков третьего поколения. В качестве стандартного языка третьего поколения был принят Unified Modeling Language (**UML**), создававшийся в 1994–1997 годах (основные разработчики – три «amigos» Г. Буч, Дж. Рамбо, И. Джекобсон). В настоящее время актуальны две версии UML 1.4 и 2.0.

3.3.1. Унифицированный язык моделирования

UML – стандартный язык для написания моделей анализа, проектирования и реализации объектно-ориентированных программных систем. UML может использоваться для визуализации, спецификации, конструирования и документирования результатов программных проектов. UML – это не визуальный язык программирования, но его модели прямо транслируются в текст на языках программирования (Java, C++, Visual Basic, Ada 95, Object Pascal) и даже в таблицы для реляционной БД.

Словарь UML образуют три разновидности строительных блоков: предметы, отношения, диаграммы.

Предметы – это абстракции, которые являются основными элементами в модели, отношения связывают эти предметы, диаграммы группируют коллекции предметов.

3.3.2. Предметы в UML

В языке UML имеются четыре разновидности предметов:

- структурные предметы;
- предметы поведения;
- группирующие предметы;
- поясняющие предметы.

Указанные предметы являются базовыми объектно-ориентированными строительными блоками. Они используются для построения моделей сложных программных систем.

Структурные предметы являются существительными в UML-моделях. Они представляют статические части модели – понятийные или физические элементы. Существует восемь разновидностей структурных предметов: класс, интерфейс, кооперация, актер, вариант использования, активный класс, компонент, узел.

Класс – это описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл). Класс реализует один или несколько интерфейсов. Как показано на рис. 3.4, графически класс отображается в виде прямоугольника, обычно включающего секции с именем, свойствами (атрибутами) и операциями.

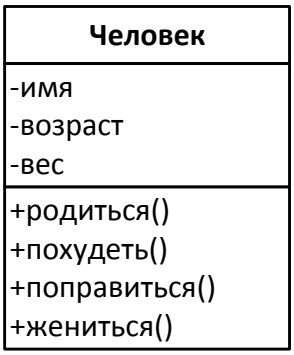

Рис. 3.4. Класс

Интерфейс – это набор операций, которые определяют услуги класса или компонента. Интерфейс описывает поведение элемента, видимое извне, может представлять полные услуги класса или компонента или часть таких услуг. Интерфейс определяет набор спецификаций операций (их сигнатуры), а не набор их реализаций. Графически интерфейс изображается в виде прямоугольника с именем и дополнительным ключевым словом «interface», как показано на рис. 3.5. Имя интерфейса обычно начинается с буквы *I*. Интерфейс редко показывают самостоятельно. Обычно его присоединяют к классу или компоненту, который реализует интерфейс.

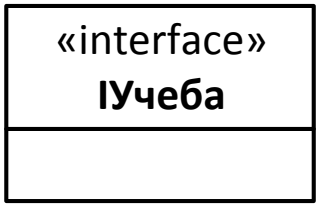

Рис. 3.5. Интерфейс

Кооперация (сотрудничество) определяет взаимодействие и является совокупностью ролей и других элементов, которые работают вместе для обеспечения коллективного поведения более сложного, чем простая сумма всех элементов. Таким образом, кооперации имеют как структурное, так и поведенческое измерения. Конкретный класс может участвовать в нескольких кооперациях. Эти кооперации представляют реализацию паттернов (шаблонов), которые формируют систему. Как показано на рис. 3.6, графически кооперация изображается в виде пунктирного эллипса, в который вписывается ее имя.

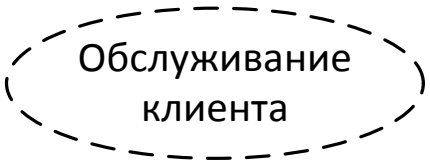

Рис. 3.6. Кооперация

Актер – это набор согласованных ролей, которые могут играть пользователи при взаимодействии с системой (ее элементами Use Case). Каждая роль требует от системы определенного поведения. Как показано на рис. 3.7, актер изображается как проволочный человечек с именем.

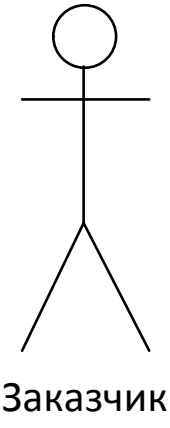

Рис. 3.7. Актер

Вариант использования *(элемент Use Case)* – это описание последовательности действий (или нескольких последовательностей), выполняемых системой в интересах отдельного актера и производящих видимый для актера результат.

В модели элемент Use Case применяется для структурирования предметов поведения. Вариант использования реализуется кооперацией. Как показано на рис. 3.8, элемент Use Case изображается в виде эллипса, в который вписывается его имя.

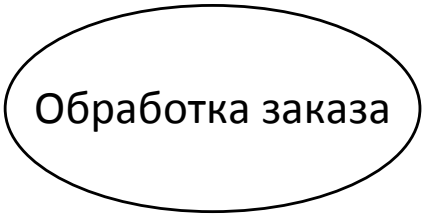

Рис. 3.8. Вариант использования системы

Активный класс – это класс, чьи объекты имеют один или несколько процессов (или потоков) и поэтому могут инициировать управляющую деятельность.

Активный класс похож на обычный класс, за исключением того, что его объекты действуют одновременно с объектами других классов. Как показано на рис. 3.7, активный класс изображается как утолщенный прямоугольник, обычно включающий имя, свойства (атрибуты) и операции.

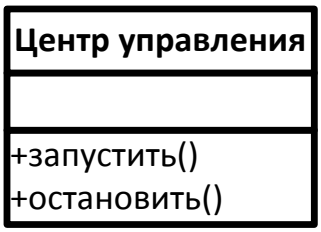

Рис. 3.9. Активный класс

Компонент – это физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов.

В систему включаются как компоненты, являющиеся результатами процесса разработки (файлы исходного кода), так и различные разновидности используемых компонентов (COM$^+$-компоненты, Java Beans и др.).

Обычно компонент – это физическая упаковка различных логических элементов (классов, интерфейсов и сотрудничеств).

Как показано на рис. 3.10, компонент изображается в виде прямоугольника с вкладками, обычно включающего имя.

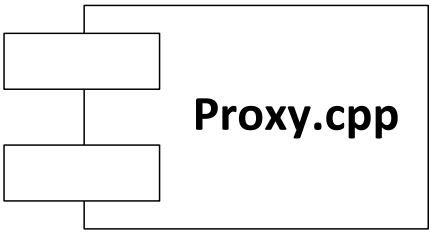

Рис. 3.10. Компонент

Узел – это физический элемент, который существует в период работы системы и представляет ресурс, обычно имеющий память и возможности обработки. В узле размещается набор компонентов, который может перемещаться от узла к узлу. Как показано на рис. 3.11, узел изображается в виде куба с именем.

Рис. 3.11. Узел

Предметы поведения – это динамические части UML-моделей. Они являются глаголами моделей, представлением поведения во времени и пространстве. Существуют две основные разновидности предметов поведения: взаимодействие и конечный автомат.

Взаимодействие – это поведение, заключающее в себе набор сообщений, которыми обменивается набор объектов в конкретном контексте для достижения определенной цели. Взаимодействие может определять динамику как совокупности объектов, так и отдельной операции.

Элементами взаимодействия являются сообщения, последовательность действий (поведение, вызываемое сообщением) и связи (соединения между объектами). Как показано на рис. 3.12, сообщение изображается в виде направленной линии с именем ее операции.

Рис. 3.12. Сообщение

Конечный автомат – это поведение, которое определяет последовательность состояний объекта или взаимодействия, выполняемые в ходе его существования в ответ на события (и с учетом обязанностей по этим событиям). С помощью конечного автомата может определяться поведение индивидуального класса или кооперации классов. Элементами конечного автомата являются состояния, переходы (от состояния к состоянию), события (предметы, вызывающие переходы) и действия (реакции на переход). Как показано на рис. 3.13, состояние изображается как закругленный прямоугольник, обычно включающий его имя и его подсостояния (если они есть).

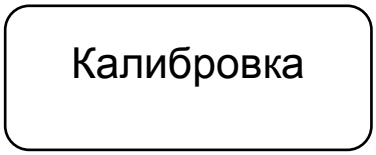

Рис. 3.13. Состояние

Указанные два элемента – взаимодействие и конечный автомат – являются базисными предметами поведения, которые могут включаться в UML-модели. Семантически эти элементы ассоциируются с различными структурными элементами (прежде всего с классами, сотрудничествами и объектами).

Группирующие предметы – это организационные части UML-моделей – ящики, по которым может быть разложена модель. Предусмотрена одна разновидность группирующего предмета – пакет.

Пакет – это общий механизм для распределения элементов по группам. В пакет могут помещаться структурные предметы, предметы поведения и даже другие группировки предметов. В отличие от компонента (который существует в период выполнения), пакет – чисто концептуальное понятие. Это означает, что пакет существует только в период разработки. Как показано на рис. 3.14, пакет изображается в виде папки с закладкой, на которой обозначено его имя и, иногда, его содержание.

Рис. 3.14. Пакет

Поясняющие предметы – это разъясняющие части UML-моделей. Они представлены в виде замечаний, которые можно применить для описания, объяснения и комментирования любого элемента модели. В UML предусмотрена одна разновидность поясняющего предмета – примечание.

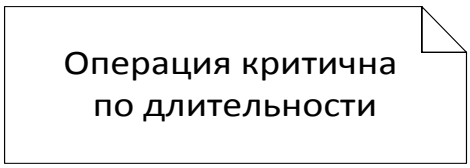

Рис. 3.15. Примечание

Примечание – это символ для отображения ограничений и замечаний, присоединяемых к элементу или совокупности элементов. Как показано на рис. 3.15, примечание изображается в виде прямоугольника с загнутым углом, в который вписывается текстовый или графический комментарий.

3.3.3. Отношения в UML

В языке UML имеются четыре разновидности отношений: зависимость, ассоциация, обобщение, реализация.

Указанные отношения являются базовыми строительными блоками для описания взаимодействия между объектами. Они используются при построении моделей программных систем.

Зависимость – это семантическое отношение между двумя предметами, в котором изменение в одном предмете (независимом предмете) может влиять на семантику другого предмета (зависимого предмета). Как показано на рис. 3.16, зависимость изображается в виде пунктирной линии, возможно, направленной на независимый предмет и иногда имеющей метку.

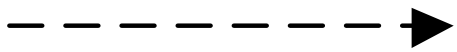

Рис. 3.16. Зависимость

Ассоциация – это структурное отношение, которое описывает набор связей, являющихся соединением между объектами. *Агрегация* – это специальная разновидность ассоциации, представляющая структурное отношение между целым и его частями. Как показано на рис. 3.17, ассоциация изображается в виде сплошной линии, возможно, направленной, иногда имеющей метку и часто включающей такие атрибуты, как мощность и имена ролей.

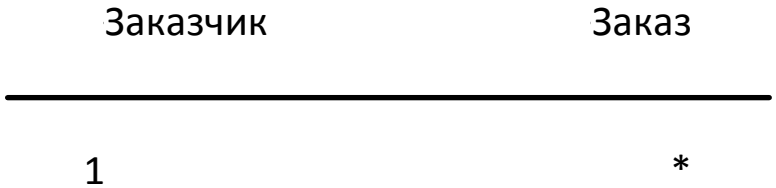

Рис. 3.17. Ассоциация

Обобщение – это отношение специализации/обобщения, в котором объекты специализированного элемента (потомка, ребенка) могут заменять объекты обобщенного элемента (предка, родителя). Иначе говоря, потомок разделяет структуру и поведение родителя. Как показано на рис. 3.18, обобщение изображается в виде сплошной стрелки с полым наконечником, указывающим на родителя.

Рис. 3.18. Обобщение

Реализация – это семантическое отношение между классификаторами, где один классификатор определяет контракт, который другой классификатор обязуется выполнять (к классификаторам относят классы, интерфейсы, компоненты, элементы Use Case, кооперации). Отношения реализации применяют в двух случаях: между интерфейсами и классами (или компонентами), реализующими их; между элементами Use Case и кооперациями, которые реализуют их. Как показано на рис. 3.19, реализация изображается как нечто среднее между обобщением и зависимостью.

Рис. 3.19. Реализация

Рассмотренные выше элементы UML являются основными типами отношений в UML-моделях. Существуют также их варианты, например *уточнение* (refinement), *трассировка* (trace), а также включение и расширение для зависимостей.

3.3.4. Диаграммы в UML

Диаграмма – это графическое представление множества элементов. Наиболее часто она изображается как связный граф из вершин (предметов) и дуг (отношений). Диаграммы рисуются для визуализации системы с разных точек зрения, затем они отображаются в систему. Обычно диаграмма дает неполное представление элементов, которые составляют систему. Хотя один и тот же элемент может появляться во всех диаграммах, на практике он появляется только в некоторых диаграммах. Теоретически диаграмма может содержать

любую комбинацию предметов и отношений, на практике ограничиваются малым количеством комбинаций, которые соответствуют пяти представлениям архитектуры программной системы (**ПС**). По этой причине UML включает девять видов диаграмм:

- диаграммы классов;
- диаграммы объектов;
- диаграммы вариантов использования (Use Case диаграммы);
- диаграммы последовательности;
- диаграммы сотрудничества (кооперации);
- диаграммы схем состояний;
- диаграммы деятельности;
- компонентные диаграммы;
- диаграммы размещения (развертывания).

Диаграмма классов показывает набор классов, интерфейсов, сотрудничеств и их отношений. При моделировании объектно-ориентированных систем диаграммы классов используются наиболее часто. Они обеспечивают статическое проектное представление системы. Диаграммы классов, включающие активные классы, обеспечивают статическое представление процессов системы.

Диаграмма объектов показывает набор объектов и их отношения. Диаграмма объектов представляет статический «моментальный снимок» экземпляров предметов, которые находятся в диаграммах классов. Как и диаграммы классов, эти диаграммы обеспечивают статическое проектное представление или статическое представление процессов системы (но с точки зрения реальных или фототипичных случаев).

Диаграмма вариантов использования (Use Case диаграмма) показывает набор элементов вариантов использования системы, актеров и их отношений. Эти диаграммы особенно важны при организации и моделировании поведения системы, задании требований заказчика к системе.

Диаграммы последовательности и диаграммы сотрудничества – это разновидности диаграмм взаимодействия.

Диаграмма взаимодействия показывает взаимодействие, включающее набор объектов и их отношений, а также пересылаемые между объектами сообщения. Диаграммы взаимодействия обеспечивают динамическое представление системы.

Диаграмма последовательности – это диаграмма взаимодействия, которая выделяет упорядочение сообщений по времени.

Диаграмма сотрудничества (диаграмма кооперации) – это диаграмма взаимодействия, которая выделяет структурную организацию объектов, посылающих и принимающих сообщения. Диаграммы последовательности и диаграммы

сотрудничества изоморфны. Это означает, что одну диаграмму можно трансформировать в другую диаграмму.

Диаграмма схем состояний показывает конечный автомат, представляет состояния, переходы, события и действия. Диаграммы схем состояний обеспечивают динамическое представление системы. Они особенно важны при моделировании поведения интерфейса, класса или сотрудничества. Эти диаграммы выделяют такое поведение объекта, которое управляется событиями, что особенно полезно при моделировании реактивных систем.

Диаграмма деятельности – это специальная разновидность диаграммы схем состояний, которая показывает поток от действия к действию внутри системы. Диаграммы деятельности обеспечивают динамическое представление системы. Они особенно важны при моделировании функциональности системы и выделяют поток управления между объектами.

Компонентная диаграмма показывает организацию набора компонентов и зависимость между ними. Компонентные диаграммы обеспечивают статическое представление реализации системы. Они связаны с диаграммами классов в том смысле, что в компонент обычно отображается один или несколько классов, интерфейсов или кооперации.

Диаграмма размещения (диаграмма развертывания) показывает конфигурацию обрабатывающих узлов периода выполнения, а также компоненты, живущие в них. Диаграммы размещения обеспечивают статическое представление размещения системы. Они связаны с компонентными диаграммами в том смысле, что узел обычно включает один или несколько компонентов.

3.4. CASE-технологии проектирования объектно-ориентированных программ

В современных условиях создание сложных программных приложений невозможно без использования систем автоматизированного конструирования программного обеспечения (CASE-систем).

CASE (англ. Computer-Aided Software Engineering) – это набор инструментов и методов программной инженерии для проектирования программного обеспечения, который помогает обеспечить высокое качество программ, отсутствие ошибок и простоту в обслуживании программных продуктов. CASE-системы существенно сокращают сроки и затраты разработки, оказывая помощь инженеру в проведении рутинных операций, облегчая его работу на самых разных этапах жизненного цикла разработки. Существует множество объектно-ориентированных CASE-систем. Кроме цены продукта, важными характеристиками средств автоматизации проектирования и разработки программного обеспечения являются поддерживаемые платформы, возможность интеграции в

другие инструментальные системы, а также поддержка кодогенерации на различных языках программирования. Наиболее развитые CASE-системы поддерживают технологию обратного инжиниринга, т.е. построение UML-моделей по исходным кодам программ.

3.4.1. Rational Rose

Rational Rose является одним из наиболее популярных средств визуального моделирования объектно-ориентированных информационных систем. Работа продукта основана на универсальном языке моделирования UML (Universal Modeling Language). Благодаря уникальному языку моделирования Rational Rose способен решать задачи проектирования информационных систем: от анализа бизнес-процессов до кодогенерации на определенном языке программирования. Программное средство Rational Rose позволяет разрабатывать как высокоуровневые, так и низкоуровневые модели, осуществляя тем самым как абстрактное проектирование, так и логическое.

Rational Rose имеет весь необходимый набор визуальных средств проектирования. Rose помогает решить проблемы с кодогенерацией на определенном языке программирования. Rational Rose осуществляет такие подходы, как прямое и обратное проектирование, а также Round Trip Engineering. Такой арсенал позволит не только проектировать новую систему, но и доработать старую, произведя процесс обратного проектирования.

Для того чтобы наиболее полно покрыть весь сегмент рынка средств проектирования и разработки, компания «Rational» выпускает несколько версий своего продукта. Каждый из них может решать как строго определенный круг задач, так и весь спектр проблем проектирования и разработки.

Rational Rose Modeler – это версия программного продукта, которая позволяет аналитикам и проектировщикам проводить анализ бизнес-процессов и выстраивать систему. Эта редакция подразумевает только моделирование без кодогенерации. Продукт будет интересен проектировщикам систем и аналитикам.

Rational Rose Professional – это профессиональная редакция продукта. Она содержит в своем наборе весь спектр изобразительных средств, в зависимости от выбранного языка программирования осуществляет прямое и обратное проектирование. Rose Professional заказывается только в определенной конфигурации (например, Rose Professional C++ или Rose Professional C++ DataModeler). Rational Rose Professional не создает 100 % исполняемого кода. На выходе разработчик получает шаблон информационной системы на определенном языке программирования, который впоследствии нужно запрограммировать. Продукт нацелен как на аналитиков, так и на разработчиков.

Rational Rose RealTime – это версия продукта для создания 100 % исполняемого кода в реальном масштабе времени. RealTime позволяет проводить прямое и обратное проектирование на языках C или C++. На выходе модель автоматически компилируется и собирается в исполняемый файл. Продукт направлен на разработчиков.

Rational Rose Enterprise – это абсолютно полная версия продукта. В ней поддерживаются все вышеперечисленные функции за исключением возможности 100%-ной кодогенерации. Версия продукта покрывает весь спектр задач по проектированию, анализу и кодогенерации. Продукт направлен на всех участников проекта конструирования программного обеспечения.

Rational Rose DataModeler – это модуль программного продукта, предоставляющий возможность проектирования баз данных. Функции DataModeler входят в состав Rose Enterprise или Professional.

В зависимости от поставки в Rational Rose может быть расширен или сужен набор диаграмм.

Rational Rose поддерживает прямое и обратное проектирование на языках ADA, Java, C, C++, Basic, поддерживает технологии COM, DDL, XML, позволяет генерировать схемы Oracle и SQL.

Rational Rose имеет открытый API, позволяющий создавать собственными силами модули для конкретных языков программирования.

На рынке существует достаточное число модулей для популярных языков программирования и систем, таких, как Delphi, ErWin, Jbuilder, VisualCafe, Jdeveloper, VisualAge SmallTalk и др. Одна из ведущих компаний в области создания дополнительных модулей – Ensemble Systems.

3.4.2. StarUML

StarUML – это программный инструмент моделирования, который поддерживает стандарт UML. StarUML ориентирован на UML версии 1.4 и поддерживает 11 различных типов диаграмм, принятых в нотации UML 2.0. Он активно поддерживает подход MDA (модельно-управляемую архитектуру), реализуя концепцию профилей UML. Среда разработки StarUML превосходно настраивается в соответствии с требованиями пользователя и имеет высокую степень расширяемости, особенно в области своих функциональных возможностей. Использование StarUML, одного из ведущих программных инструментов моделирования, гарантирует достижение максимальной производительности и качества программных проектов.

StarUML – это пакет с открытым программным кодом, написанный на Delphi и работающий под управлением ОС семейства Windows. Функционал пакета можно расширить за счет использования плагинов, так что каждый желающий

может создать свой собственный модуль для StarUML на любом COM-совместимом языке (C++, Delphi, C#, ...). На сайте проекта доступны для загрузки несколько модулей, добавляющих поддержку ER-диаграмм (Entity-Relation Diagram), некоторых профайлов UML, например SPEM (Software Process Engineering Metamodel), WAE (Web Application Extension), интеграцию с MS Word и др.

Интерфейс пакета StarUML не может похвастаться красивыми разноцветными «пластмассовыми» элементами управления, но очень удобен и интуитивно понятен. Больше всего StarUML напоминает интегрированную среду разработки программного обеспечения Microsoft Visual Studio. StarUML с успехом может заменить такие коммерческие программы, как Rational Rose, Together или TAU G2. Пакет способен выполнять кодогенерацию на языках C++, C#, Java. А если использовать шаблоны, имеющиеся на сайте StarUML, то можно добавить поддержку PHP и некоторых других языков.

В качестве достоинств указанного программного продукта можно выделить:

- генерацию кода в языки: C#, Java, C++;
- поддержку работы с фреймворками;
- удобный графический редактор;
- полное соответствие стандарту UML 2.0.
- возможность расширения функционала (про это написано отдельное руководство разработчика);
- экспорт документации в форматы: DOC, PPT, TXT, XLS;
- поддержку паттернов (шаблонов) проектирования;
- возможность импорта проектов Rational Rose.

3.4.3. Sybase PowerDesigner

PowerDesigner – это полнофункциональный инструмент для моделирования бизнес-приложений, включающий в себя средства моделирования бизнес-процессов, сочетающий возможности моделирования UML-объектов с возможностями традиционного проектирования баз данных и анализа и предоставляющий централизованный репозиторий объектов масштаба предприятия. Репозиторий объектов масштаба предприятия обеспечивает:

- возможность одновременной работы над одной моделью многих аналитиков и проектировщиков;
- хранение, управление и создание версий моделей PowerDesigner и других документов;
- поиск объектов в модели и их повторное использование в других моделях;
- эффективное управление взаимосвязями между моделями.

Основные функциональные возможности и особенности программного продукта следущие:

- моделирование бизнес-процессов на основе диаграмм потоков управления;
- технологии моделирования данных (концептуальная, логическая и физическая модели), основанные на индустриальном стандарте «сущность/связь» (entity/relationship), включая технологии моделирования хранилищ данных (схему «звезда», схему «снежинка», многомерное моделирование, привязку к конкретному источнику данных).
- стандартные диаграммы UML: вариантов использования, деятельности, последовательности, диаграммы классов и компонентов;
- генерация на основе диаграмм классов исходных текстов для Java, C++, PowerBuilder и Visual Basic;
- генерация операторов DDL (Data Definition Language) более чем для 30 реляционных СУБД;
- поддержка EJB 2.0;
- отображение «сущность/связь»;
- определение сложных пользовательских типов данных, включая Java-классы и хранимые Java-процедуры, содержащиеся в БД;
- обратное проектирование схемы базы данных в концептуальную и физическую модели;
- обратное проектирование существующей бизнес-логики в диаграммы классов (Java и PowerBuilder);
- прямое и обратное проектирование XML-приложений в диаграммы классов. Поддержка XML-DTD, XML-схемы и XML-данных;
- интеграция с популярными средствами разработки на Java и с ведущими, сертифицированными под J2EE/EJB 2.0 серверами приложений;
- современный, графический, настраиваемый пользовательский интерфейс, содержащий: общую оболочку, обозреватель объектов, область редактирования диаграмм, область состояния;
- улучшенное управление моделями, включая синхронизацию объектов, моделей и баз данных;
- расширенный, независимый от модели генератор отчетов, позволяющий получить документ, включающий в себя информацию по нескольким моделям.

В зависимости от требований к проектированию и разработке можно выбрать только необходимые модули PowerDesigner, т.е. приобрести только тот комплект функциональных возможностей, который необходим.

PhysicalArchitect (PDM). Данный модуль осуществляет физическое проектирование и генерацию базы данных, включая моделирование хранилищ данных. Этот модуль, входящий в состав минимальной конфигурации, обеспечивает инструментальные средства, необходимые для создания физических моделей баз данных, как OLTP, так и OLAP, генерации SQL-кода и реинжиниринга существующих баз данных из гетерогенных источников.

DataArchitect (PDM, CDM). Модуль осуществляет двухуровневое, итерационное проектирование баз данных и генерацию операторов описания базы данных (**DLL**), поддерживает интегрированное физическое и концептуальное моделирование данных (включая моделирование хранилищ данных), позволяя проектировать и генерировать базы данных более чем для 30 реляционных СУБД и настольных платформ.

ObjectArchitect (PDM, CDM, OOM). Модуль осуществляет объектно-ориентированный анализ и проектирование в комбинации с двухуровневым, итерационным проектированием баз данных и языком описания базы данных (**DLL**). Теперь, с расширенной поддержкой UML, интегрированной с возможностями моделирования данных от DataArchitect, этот модуль делает максимально эффективной работу проектировщиков баз данных и приложений.

Developer (PDM, OOM). Данный модуль осуществляет объектное моделирование и физическое проектирование баз данных. Это – идеальный инструмент для разработчика. Он позволяет выполнять физическое моделирование данных совместно с UML-моделированием, включающее возможности генерации сложного объектного кода и реинжиниринг.

Studio (BPM, PDM, CDM, OOM). Поставляемый в редакциях Personal и Enterprise модуль Studio объединяет в себе технологии моделирования бизнес-процессов с возможностями расширенного UML-моделирования и моделирования данных. Комбинируя все необходимые методы моделирования в одной среде, модули этого типа дают возможность бизнес- и IT-менеджерам с их командами разработчиков совместно строить бизнес-системы, эффективно поддерживающие работу предприятия.

Viewer. Модуль обеспечивает всем IT-специалистам организации единое представление информации о моделировании. Этот модуль, работающий только в режиме чтения (read-only), обеспечивает графическое представление информации обо всех смоделированных компонентах системы и включает генератор отчетов с расширенной функциональностью.

Обновление любого модуля до версии Enterprise открывает возможность ведения центрального архива для моделей и других файлов, что значительно облегчает контроль доступа, совместное использование, сотрудничество, консолидацию, проверку версий и управление в рамках работы с моделями.

Вопросы для самопроверки

1. Назовите цели и основные этапы проектирования программных средств.
2. Охарактеризуйте процесс анализа предметной области.
3. Какие задачи решают системные аналитики?
4. Каковы основные этапы формулировки потребностей возможных пользователей и заказчиков программного средства?
5. Какие виды связей могут устанавливаться между вариантами использования?
6. Перечислите основные атрибуты варианта использования.
7. Перечислите разновидности строительных блоков, составляющих словарь UML.
8. Перечислите и охарактеризуйте разновидности предметов в UML.
9. Перечислите и охарактеризуйте разновидности отношений в UML.
10. Перечислите и охарактеризуйте разновидности диаграмм в UML.

4. ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНЫХ СРЕДСТВ

4.1. Основные понятия тестирования и отладки программных средств

Отладка ПС – это деятельность по обнаружению и исправлению ошибок в ПС с использованием процессов выполнения его программ.

Тестирование ПС – процесс выполнения его программ на некотором наборе данных, для которого заранее известен результат применения или известны правила поведения этих программ. Этот набор данных называется *тестовым* или просто *тестом*.

Отладка = Тестирование + Поиск ошибок + Редактирование.

Задачи отладки программных средств.

1. Подготовить такой набор тестов и применить к ним ПС, чтобы обнаружить в нем по возможности большее число ошибок.

2. Определить момент окончания отладки ПС (или отдельной его компоненты).

Для оптимизации набора тестов, необходимо заранее планировать этот набор и использовать рациональную стратегию планирования тестов.

Проектирование тестов можно начинать сразу же после завершения этапа внешнего описания ПС. Возможны разные подходы к выработке стратегии проектирования тестов, которые можно условно графически разместить (рис. 4.1) между следующими двумя крайними подходами.

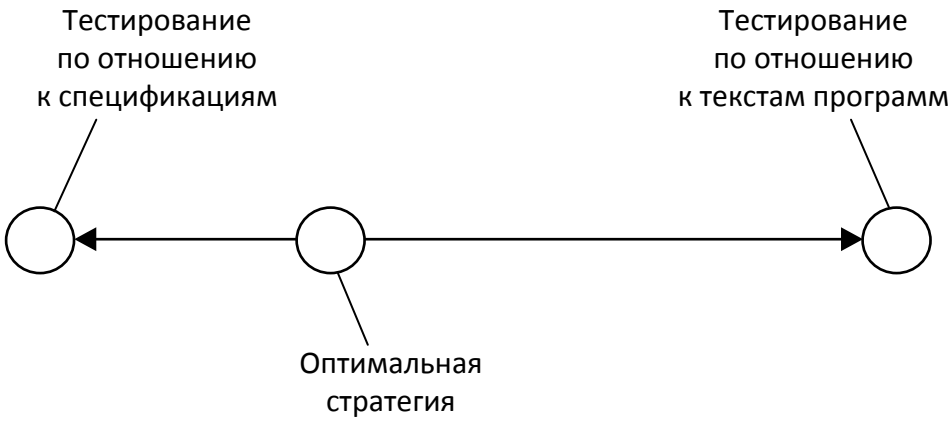

Рис. 4.1. Спектр подходов к проектированию тестов.

Оптимальная стратегия проектирования тестов расположена внутри интервала между этими крайними подходами, но ближе к левому краю. При этом, в первом случае эта стратегия базируется на принципах:

- на каждую используемую функцию или возможность – хотя бы один тест;

- на каждую область и на каждую границу изменения какой-либо входной величины – хотя бы один тест;
- на каждую особую (исключительную) ситуацию, указанную в спецификациях, – хотя бы один тест.

Во втором случае эта стратегия базируется на принципе: каждая команда каждой программы ПС должна проработать хотя бы на одном тесте. Различают два основных вида отладки (включая тестирование):

Автономная отладка – последовательное раздельное тестирование различных частей программ с поиском и исправлением в них фиксируемых при тестировании ошибок.

Комплексная отладка – тестирование ПС в целом с поиском и исправлением фиксируемых при тестировании ошибок во всех документах.

4.2. Отладка программного средства

4.2.1. Принципы и виды отладки программного средства

В целом, отладку можно разделить на два вида: синтаксическую и семантическую.

Отладка синтаксиса обычно не вызывает трудностей и требует только аккуратности. Результаты синтаксически правильной программы сравниваются с тестовыми, их несовпадение является признаком наличия семантической ошибки. Это сравнение надо выполнять очень скрупулезно: даже лишний пробел между двумя значениями может дать ключ к поиску ошибки.

В отличие от синтаксической ошибки, *семантическая ошибка не формализуема,* и поэтому она составляет основу отладки.

Отладка готовой части (ветви) программного обеспечения может быть начата ранее, чем будет написан весь программный продукт, в результате чего сокращается время разработки ПО, то есть экономится календарное время его разработки. Временно отсутствующие блоки либо оставляются пустыми, либо заменяются печатью сообщения о том, что программное обеспечение не рассчитано на такие данные, или пишутся временные программные заглушки.

Отметим феномен – по мере роста числа обнаруженных и исправленных ошибок в ПС растет также относительная вероятность существования в нем необнаруженных ошибок.

Рассмотрим основные принципы отладки ПС:

1. Считайте тестирование ключевой задачей разработки ПС, поручайте его самым квалифицированным и одаренным программистам; нежелательно тестировать свою собственную программу.

2. Хорош тот тест, для которого высока вероятность обнаружения ошибки, а не тот, который демонстрирует правильную работу программы.

3. Готовьте тесты, как для правильных, так и для неправильных данных.

4. Документируйте пропуск тестов через компьютер; детально изучайте результаты каждого теста; избегайте тестов, пропуск которых нельзя повторить.

5. Каждый модуль подключайте к программе только один раз; никогда не изменяйте программу, чтобы облегчить ее тестирование.

6. Пропускайте заново все тесты, связанные с проверкой работы какой-либо программы ПС или ее взаимодействия с другими программами, если в нее были внесены изменения (например, в результате устранения ошибки).

4.2.2. Автономная отладка программного средства

При автономной отладке тестируется всегда некоторая программа (тестируемая программа), построенная специально для тестирования отлаживаемого модуля. В процессе автономной отладки ПС производится наращивание тестируемой программы отлаженными модулями (интеграция программы).

При восходящем тестировании окружение содержит только один отладочный модуль, головной в тестируемой программе – ведущий (или драйвер). Ведущий отладочный модуль подготавливает информационную среду для тестирования отлаживаемого модуля, осуществляет обращение к отлаживаемому модулю и выдает необходимые сообщения.

Достоинства восходящего тестирования:

- простота подготовки тестов;
- возможность полной реализации плана тестирования модуля.

Недостатки восходящего тестирования:

- тестовые данные готовятся, как правило, не в той форме, которая рассчитана на пользователя;
- большой объем отладочного программирования;
- необходимость специального тестирования сопряжения модулей.

При нисходящем тестировании окружение в качестве отладочных содержит отладочные имитаторы (заглушки) некоторых еще не отлаженных модулей. Некоторые из этих имитаторов при отладке одного модуля могут изменяться для разных тестов.

На практике в окружении отлаживаемого модуля могут содержаться отладочные модули обоих типов, если используется смешанная стратегия тестирования.

Достоинства нисходящего тестирования:

- большинство тестов готовится в форме, рассчитанной на пользователя;

- во многих случаях относительно небольшой объем отладочного программирования;
- отпадает необходимость тестирования сопряжения модулей.

Недостатком нисходящего тестирования является то, что тестовое состояние информационной среды перед обращением к отлаживаемому модулю готовится косвенно – оно является результатом применения уже отлаженных модулей к тестовым данным или данным, выдаваемым имитаторами.

Прежде всего, необходимо организовать отладку программы таким образом, чтобы как можно раньше были отлажены модули, осуществляющие ввод данных. Пока не отлажены модули, осуществляющие ввод данных, тестовые данные поставляются некоторыми имитаторами: они либо включаются в имитатор как его часть, либо вводятся этим имитатором.

При нисходящем тестировании некоторые состояния информационной среды, при которых требуется тестировать отлаживаемый модуль, могут не возникать при выполнении отлаживаемой программы ни при каких входных данных. Чаще же пользуются модифицированным вариантом нисходящего тестирования, при котором отлаживаемые модули перед их интеграцией предварительно тестируются отдельно. Однако представляется более целесообразной другая модификация нисходящего тестирования: после завершения нисходящего тестирования отлаживаемого модуля для достижимых тестовых состояний информационной среды следует его отдельно протестировать для остальных требуемых состояний информационной среды.

Часто применяют также комбинацию восходящего и нисходящего тестирования, которую называют *методом сэндвича*. Суть этого метода заключается в одновременном осуществлении как восходящего, так и нисходящего тестирования, пока оба этих процесса тестирования не встретятся на каком-либо модуле где-то в середине структуры отлаживаемой программы.

Весьма важным при автономной отладке является тестирование сопряжения модулей.

При нисходящем тестировании тестирование сопряжения осуществляется попутно каждым пропускаемым тестом, что считают достоинством нисходящего тестирования.

При восходящем тестировании обращение к отлаживаемому модулю производится не из модулей отлаживаемой программы, а из ведущего отладочного модуля.

Автономное тестирование модуля целесообразно осуществлять в четыре последовательно выполняемых шага.

Шаг 1. На основании спецификации отлаживаемого модуля подготовьте тесты для каждой возможности и каждой ситуации, для каждой границы областей

допустимых значений всех входных данных, для каждой области изменения данных, для каждой области недопустимых значений всех входных данных и каждого недопустимого условия.

Шаг 2. Проверьте текст модуля, чтобы убедиться, что каждое направление любого разветвления будет пройдено хотя бы на одном тесте. Добавьте недостающие тесты.

Шаг 3. Проверьте текст модуля, чтобы убедиться, что для каждого цикла существуют тесты, обеспечивающие, по крайней мере, три следующие ситуации:

- тело цикла не выполняется ни разу,
- тело цикла выполняется один раз,
- тело цикла выполняется максимальное число раз.

Добавьте недостающие тесты.

Шаг 4. Проверьте текст модуля, чтобы убедиться, что существуют тесты, проверяющие чувствительность к отдельным особым значениям входных данных. Добавьте недостающие тесты.

4.2.3. Комплексная отладка программного средства

Тестирование при *комплексной отладке* – это применение ПС к конкретным данным, которые могут возникнуть у пользователя, но, возможно, в моделируемой (а не в реальной) среде. В рамках комплексной отладки используются основные типы тестирования, в числе которых:

- *тестирование архитектуры ПС.* Целью тестирования является поиск несоответствия между описанием архитектуры и совокупностью программ ПС. К моменту начала тестирования архитектуры ПС должна быть уже закончена автономная отладка каждой подсистемы;
- *тестирование внешних функций.* Целью тестирования является поиск расхождений между функциональной спецификацией и совокупностью программ ПС. Несмотря на то, что все эти программы автономно уже отлажены, указанные расхождения могут быть;
- *тестирование качества ПС.* Целью тестирования является поиск нарушений требований качества, сформулированных в спецификации качества ПС. Завершенность ПС проверяется уже при тестировании внешних функций;
- *тестирование документации по применению ПС.* Целью тестирования является поиск несогласованности документации по применению и совокупностью программ ПС, а также выявление неудобств, возникающих при применении ПС. Этот этап непосредственно предшествует подключению пользователя к завершению разработки ПС (тестированию определения требований к ПС и аттестации ПС);

- *тестирование определения требований к ПС.* Целью тестирования является выяснение, в какой мере ПС не соответствует предъявленному определению требований к нему. Особенность этого вида тестирования заключается в том, что его осуществляет организация-покупатель или организация-пользователь. Обычно производится с помощью контрольных задач – типовых, для которых известен результат решения;
- *тестирования в объектно-ориентированных системах.* Тестирование является достаточно независимым процессом, применимым к программному обеспечению, разработанному с помощью любого метода проектирования. Тем не менее, объектно-ориентированный подход привносит свои особенности.

Традиционно тестирование делится на тестирование элементов, интеграционное тестирование и системное тестирование. На уровне элементов тестирование объектно-ориентированных программ отличается по следующим показателям:

- определение единиц тестирования;
- тестирование наследования;
- тестирование полиморфизма.

Естественной единицей тестирования является *класс*. Разбиение его на более мелкие элементы (методы) нецелесообразно, поскольку отдельно от классов они не существуют. Иногда за единицу тестирования принимается *тесно связанная группа классов.*

Тестирование наследования состоит в тестировании методов, унаследованных классом от своего базового класса. Если базовый класс уже прошел тестирование, нужно ли повторять его для унаследованных методов? Вопреки достаточно распространенным надеждам программистов перетестирование необходимо – основная причина заключается в том, что методы выполняются в новом контексте. Применимость тестовых сценариев базового класса для тестирования производного класса зависит от того, является ли наследование классификацией. Если нет, то даже для унаследованных методов необходимо разрабатывать новые тестовые сценарии.

Тестирование полиморфизма сходно с тестированием наследования в том, что в тестовых сценариях необходимо предусмотреть все варианты связывания, то есть все варианты конкретной реализации полиморфизма.

Интеграционное тестирование представляет собой тестирование того процесса, при котором отдельные элементы программы работают вместе. Свойства объектно-ориентированных языков исключают целый ряд возможных ошибок, прежде всего на основе строгого определения внешних интерфейсов классов и

объектов. Однако указаннное не означает, что интеграционное тестирование становится легче. Планирование интеграционного тестирования немного отличается от традиционного подхода.

При традиционном тестировании имеется такая характеристика, как степень покрытия кода. При тестировании по принципу открытого, или белого ящика (то есть в случае, когда тестирующему структура кода известна) необходимо обеспечить прохождение управления по всем ответвлениям программы. Иными словами, требуется выполнить как можно больший процент инструкций, имеющихся в программе. Наряду с этой характеристикой планирование тестов для объектно-ориентированной программы должно включать «покрытие состояний».

То, что объект соединяет в себе состояние и поведение, обуславливает необходимость проверки всех возможных переходов из состояния в состояние. В планировании таких тестов должна помочь модель состояний класса, построенная на этапе анализа и проектирования.

Системное тестирование проверяет всю программную систему целиком и строится (в большинстве случаев) по принципу «черного ящика». Тестирующий знает только внешние характеристики системы, но не представляет себе, как она работает.

Построение требований к системе в форме вариантов использования обеспечивает естественный и простой способ планирования системного тестирования. Фактически система вариантов использования становится основой плана тестов на этапе системного тестирования.

4.3. Тестирование программного кода

Тестирование программного кода – это процесс выполнения программного кода, направленный на выявление существующих в нем дефектов. Под *дефектом* здесь понимается участок программного кода, выполнение которого при определенных условиях приводит к неожиданному поведению системы (то есть поведению, не соответствующему требованиям). Неожиданное поведение системы может приводить к сбоям в ее работе и отказам, в этом случае говорят о *существенных дефектах программного кода*. Некоторые дефекты вызывают незначительные проблемы, не нарушающие процесс функционирования системы, но несколько затрудняющие работу с ней. В этом случае говорят о *средних или малозначительных дефектах*.

Задача тестирования при таком подходе – это определение условий, при которых проявляются дефекты системы и протоколирование этих условий. В задачи тестирования обычно не входит выявление конкретных дефектных

участков программного кода и никогда не входит исправление дефектов – это задача отладки, которая выполняется по результатам тестирования системы.

Цель применения процедуры тестирования программного кода – минимизация количества дефектов в конечном продукте, в особенности существенных. Тестирование само по себе не может гарантировать полного отсутствия дефектов в программном коде системы. Однако, в сочетании с процессами верификации и валидации, направленными на устранение противоречивости и неполноты проектной документации (в частности – требований на систему), грамотно организованное тестирование дает гарантию того, что система удовлетворяет требованиям и ведет себя в соответствии с ними во всех предусмотренных ситуациях.

При разработке систем повышенной надежности, например, авиационных, гарантии надежности достигаются при помощи четкой организации процесса тестирования, определения его связи с остальными процессами жизненного цикла, введения количественных характеристик, позволяющих оценивать успешность тестирования. При этом, чем выше требования к надежности системы (ее уровень критичности), тем более жесткие требования предъявляются.

Таким образом, в первую очередь мы, используя подход «хорошо организованный процесс дает качественный результат», рассматриваем не конкретные результаты тестирования конкретной системы, а общую организацию самого процесса. Такой подход является общим для многих международных и отраслевых стандартов качества. При таком подходе качество разрабатываемой системы является следствием организованного процесса разработки и тестирования, а не самостоятельным неуправляемым результатом.

Поскольку современные программные системы имеют весьма значительные размеры, при тестировании их программного кода используется *метод* ***функциональной декомпозиции*** – то есть система разбивается на отдельные модули (классы, пространства имен и т.п.), имеющие функциональность и интерфейсы, определенные требованиями. После этого каждый модуль тестируется по отдельности – выполняется модульное тестирование. Затем выполняется сборка отдельных модулей в более крупные конфигурации – выполняется интеграционное тестирование, и, наконец, тестируется система в целом – выполняется системное тестирование.

С точки зрения программного кода, в модульном, интеграционном и системном тестировании содержится много общего, поэтому в данном пособии основное внимание будет уделено модульному тестированию.

В ходе модульного тестирования каждый модуль тестируется как на соответствие требованиям, так и на отсутствие проблемных участков программного кода, могущих вызвать отказы и сбои в работе системы. Как правило, модули не работают вне системы – они принимают данные от других модулей, перерабатывают их и передают дальше. Для того чтобы изолировать модуль от системы и исключить влияние потенциальных ошибок системы, с одной стороны, а с другой стороны – обеспечить модуль всеми необходимыми данными, используется тестовое окружение.

Задача тестового окружения – создать среду выполнения для модуля, эмулировать все внешние интерфейсы, к которым он обращается. Об особенностях организации тестового окружения пойдет речь в данной теме.

Типичная процедура тестирования состоит в подготовке и выполнении тестовых примеров (также называемых *просто тестами*). Каждый тестовый пример проверяет одну «ситуацию» в поведении модуля и состоит из списка значений, передаваемых на вход модуля, описания запуска и выполнения переработки данных – тестового сценария, и списка значений, которые ожидаются на выходе модуля в случае его корректного поведения. Тестовые сценарии составляются таким образом, чтобы исключить обращения к внутренним данным модуля, все взаимодействие должно происходить только через его внешние интерфейсы.

Выполнение тестового примера поддерживается тестовым окружением, которое включает в себя программную реализацию тестового сценария. Выполнение начинается с передачи модулю входных данных и запуска сценария. Реальные выходные данные, полученные от модуля в результате выполнения сценария сохраняются и сравниваются с ожидаемыми. В случае их совпадения тест считается пройденным, в противном случае – не пройденным. Каждый не пройденный тест указывает либо на дефект в тестируемом модуле, либо в тестовом окружении, либо в описании теста.

Совокупность описаний тестовых примеров составляет ***тест-план*** – основной документ, определяющий процедуру тестирования программного модуля. Тест-план задает не только сами тестовые примеры, но и порядок их следования, который также может быть важен.

При тестировании часто бывает необходимо учитывать не только требования к системе, но и структуру программного кода тестируемого модуля. В этом случае тесты составляются таким образом, чтобы детектировать типичные ошибки программистов, вызванные неверной интерпретацией требований. Применяются проверки граничных условий, проверки классов эквивалентности. Отсутствие в системе возможностей, не заданных требованиями, гарантируют различные оценки покрытия программного кода

тестами, то есть оценку того, какой процент тех или иных языковых конструкций в результате выполнения всех тестовых примеров будет осуществлен.

4.3.1. Тестирование методом черного ящика

Основная идея в тестировании *системы как черного ящика* состоит в том, что все материалы, которые доступны тестировщику – это всего только требования к системе, описывающие ее поведение, и сама система, работать с которой он сможет, лишь подавая некоторые внешние воздействия на ее входы и наблюдая на выходах некоторый результат. Все внутренние особенности реализации системы от тестировщика скрыты. Таким образом, система представляет собой «черный ящик», правильность поведения которого по отношению к требованиям и предстоит проверить.

С точки зрения программного кода, черный ящик может представлять собой набор классов (или модулей) с известными внешними интерфейсами, но недоступными исходными текстами.

Основная задача тестировщика для данного метода тестирования состоит в последовательной проверке соответствия поведения системы требованиям. Кроме того, тестировщик должен проверить работу системы в критических ситуациях – что происходит в случае подачи неверных входных значений. В идеальной ситуации все варианты критических ситуаций должны быть описаны в требованиях на систему и тестировщику остается только придумывать конкретные проверки этих требований. Однако в реальности в результате тестирования обычно выявляется два типа проблем системы:

1. **Несоответствие поведения системы требованиям.**
2. **Неадекватное поведение системы в ситуациях, не предусмотренных требованиями.**

Отчеты об обоих типах проблем документируются и передаются разработчикам. При этом проблемы первого типа обычно вызывают изменение программного кода, гораздо реже – изменение требований. Изменение требований в данном случае может потребоваться ввиду их противоречивости (несколько разных требований описывают разные модели поведения системы в одной и той же ситуации) или некорректности (требования не соответствуют действительности).

Проблемы второго типа однозначно требуют изменения требований ввиду их неполноты – в требованиях явно пропущена ситуация, приводящая к неадекватному поведению системы. При этом под неадекватным поведением может пониматься как полный крах системы, так и вообще любое поведение, не описанное в требованиях.

Тестирование черного ящика называют также тестированием по требованиям, так как это единственный источник информации для построения тест-плана.

4.3.2. Прозрачный (белый) ящик

При тестировании системы как стеклянного ящика, тестировщик имеет доступ не только к требованиям на систему, ее входам и выходам, но и к ее внутренней структуре – видит ее программный код.

Доступность программного кода расширяет возможности тестировщика тем, что он может видеть соответствие требований участкам программного кода и видеть тем самым – на весь ли программный код существуют требования. Программный код, для которого отсутствуют требования, называют кодом, непокрытым требованиями. Такой код является потенциальным источником неадекватного поведения системы. Кроме того, прозрачность системы позволяет углубить анализ ее участков, вызывающих проблемы – часто одна проблема нейтрализует другую и они никогда не возникают одновременно.

4.3.3. Тестирование моделей

Тестирование моделей находится несколько в стороне от классических методов тестирования программного обеспечения. Прежде всего, это связано с тем, что объект тестирования – не сама система, а ее модель, спроектированная формальными средствами. Если оставить в стороне вопросы проверки корректности и применимости самой модели (считается, что ее корректность и соответствие исходной системе может быть доказана формальными средствами), то тестировщик получает в свое распоряжение достаточно мощный инструмент анализа общей целостности системы. Работая с моделью можно создать такие ситуации, которые невозможно создать в тестовой лаборатории для реальной системы. Работая с моделью программного кода системы можно анализировать его свойства и такие параметры системы, как оптимальность алгоритмов или ее устойчивость.

Однако тестирование моделей не получило широкого распространения именно ввиду трудностей, возникающих при разработке формального описания поведения системы. Одно из немногих исключений – системы связи, алгоритмический и математический аппарат которых достаточно хорошо проработан.

4.3.4. Анализ программного кода (инспекции)

Во многих ситуациях тестирование поведения системы в целом невозможно – отдельные участки программного кода могут никогда не выполняться, при этом они будут покрыты требованиями. Примером таких участков кода могут служить обработчики исключительных ситуаций. Если, например, два модуля передают друг другу числовые значения, и функции проверки корректности значений работают в обоих модулях, то функция проверки модуля-приемника никогда не будет активизирована, т.к. все ошибочные значения будут отсечены еще в передатчике.

В этом случае выполняется ручной анализ программного кода на корректность, называемый также просмотрами или инспекциями кода. Если в результате инспекции выявляются проблемные участки, то информация об этом передается разработчикам для исправления.

4.4. Тестовое окружение

Основной объем тестирования практически любой сложной системы обычно выполняется в автоматическом режиме.

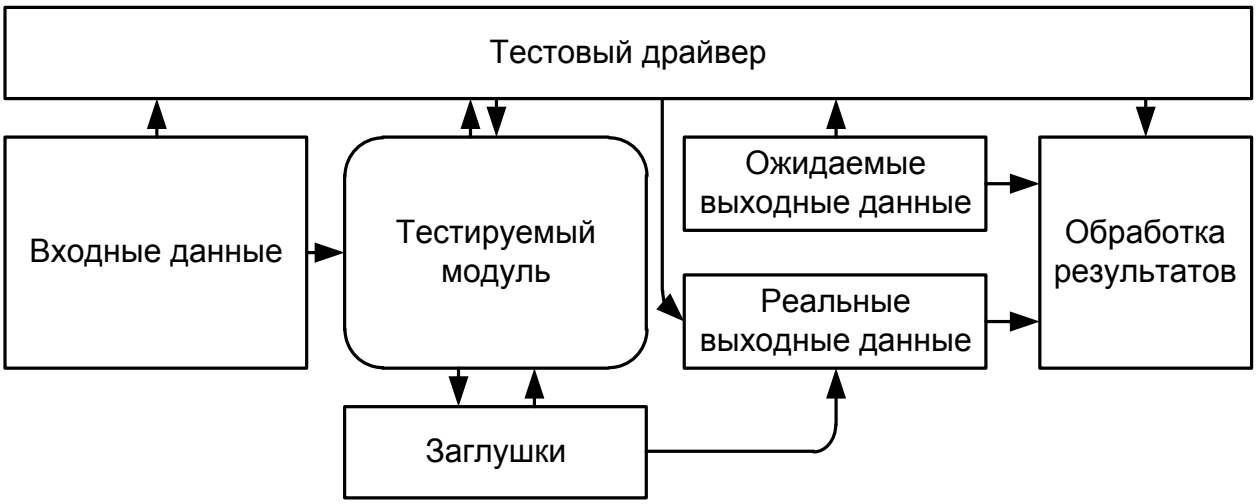

Рис. 4.1. Обобщенная схема среды тестирования

Кроме того, тестируемая система обычно разбивается на отдельные модули, каждый из которых тестируется вначале отдельно от других, затем в комплексе. Это означает, что для выполнения тестирования необходимо создать некоторую среду, которая обеспечит запуск и выполнение тестируемого модуля, передаст ему входные данные, соберет реальные выходные данные, полученные в результате работы системы на заданных входных данных. После этого среда должна сравнить реальные выходные данные с ожидаемыми и на основании

данного сравнения сделать вывод о соответствии поведения модуля заданному (см. рис. 4.1).

Тестовое окружение также может использоваться для отчуждения отдельных модулей системы от всей системы. Разделение модулей системы на ранних этапах тестирования позволяет более точно локализовать проблемы, возникающие в их программном коде. Для поддержки работы модуля в отрыве от системы тестовое окружение должно моделировать поведение всех модулей, к функциям или данным которых обращается тестируемый модуль.

Поскольку тестовое окружение само является программой (причем, часто не на том языке программирования, на котором написана система), оно само должно быть протестировано. Целью тестирования тестового окружения является доказательство того, что тестовое окружение никаким образом не искажает выполнение тестируемого модуля и адекватно моделирует поведение системы.

4.4.1. Драйверы и заглушки

Тестовое окружение для программного кода на структурных языках программирования состоит из двух компонентов – драйвера, который обеспечивает запуск и выполнение тестируемого модуля и заглушек, которые моделируют функции, вызываемые из данного модуля. Разработка тестового драйвера представляет собой отдельную задачу тестирования, сам драйвер должен быть протестирован, дабы исключить неверное тестирование. Драйвер и заглушки могут иметь различные уровни сложности, причем требуемый уровень сложности выбирается в зависимости от сложности тестируемого модуля и уровня тестирования.

Так, ***драйвером*** может обеспечиваться выполнение следующих функций:

1. Вызов тестируемого модуля.
2. Передача в тестируемый модуль входных значений и прием результатов.
3. Вывод выходных значений.
4. Протоколирование процесса тестирования и ключевых точек программы.

Заглушки могут:

1. Не производить никаких действий (такие заглушки нужны для корректной сборки тестируемого модуля.
2. Выводить сообщения о том, что заглушка была вызвана.
3. Выводить сообщения со значениями параметров, переданных в функцию.
4. Возвращать значение, заранее заданное во входных параметрах теста.
5. Выводить значение, заранее заданное во входных параметрах теста.
6. Принимать от тестируемого ПО значения и передавать их в драйвер.

Для тестирования программного кода, написанного на процедурном языке программирования, используются драйверы, представляющие собой программу с точкой входа (например, функцией main()), функциями запуска тестируемого модуля и функциями сбора результатов. Обычно драйвер имеет как минимум одну функцию – точку входа, которой передается управление при его вызове.

Функции-заглушки могут помещаться в тот же файл исходного кода, что и основной текст драйвера. Имена и параметры заглушек должны совпадать с именами и параметрами «заглушаемых» функций реальной системы. Это требование важно не столько с точки зрения корректной сборки системы (при сборке тестового драйвера и тестируемого ПО может использоваться приведение типов), сколько для того, чтобы максимально точно моделировать поведение реальной системы по передаче данных. Так, например, если в реальной системе присутствует функция вычисления квадратного корня

```
double sqrt(double value);
```

то с точки зрения сборки системы вместо типа double может использоваться и float, но снижение точности может вызвать непредсказуемые результаты в тестируемом модуле.

4.4.2. Тестовые классы

Тестовое окружение для объектно-ориентированного ПО выполняет те же самые функции, что и для структурных программ (на процедурных языках). Однако оно имеет некоторые особенности, связанные с применением наследования и инкапсуляции.

Если при тестировании структурных программ минимальным тестируемым объектом является функция, то в объектно-ориентированным ПО минимальным объектом является класс. При применении принципа инкапсуляции, все внутренние данные класса и некоторая часть его методов недоступна извне. В этом случае тестировщик лишен возможности обращаться в своих тестах к данным класса и произвольным образом вызывать методы – единственное, что ему доступно – вызывать методы внешнего интерфейса класса.

Существует несколько подходов к тестированию классов, каждый из них накладывает свои ограничения на структуру драйвера и заглушек:

1. Драйвер создает один или больше объектов тестируемого класса, все обращения к объектам происходят только с использованием их внешнего интерфейса. Текст драйвера в этом случае представляет собой так называемый тестирующий класс, который содержит по одному методу для каждого тестового примера. Процесс тестирования заключается в последовательном вызове этих методов. Вместо заглушек в состав

тестового окружения входит программный код реальной системы, соответственно отсутствует изоляция тестируемого класса. Однако именно такой подход к тестированию принят сейчас в большинстве методологий и сред разработки. Его классическое название – ***unit testing*** (тестирование модулей).

2. Аналогично предыдущему подходу, но для всех классов, которые использует тестируемый класс, создаются заглушки.

3. Программный код тестируемого класса модифицируется таким образом, чтобы открыть доступ ко всем его свойствам и методам. Строение тестового окружения в этом случае полностью аналогично окружению для тестирования структурных программ.

4. Используются специальные средства доступа к закрытым данным и методам класса на уровне объектного или исполняемого кода – скрипты отладчика или accessors в Visual Studio.

Основное **достоинство** двух первых методов – при их использовании класс работает точно таким же образом, как в реальной системе. Однако в этом случае нельзя гарантировать того, что в процессе тестирования будет выполнен весь программный код класса и не останется непротестированных методов.

Основной **недостаток** 3-го метода – после изменения исходных текстов тестируемого модуля нельзя дать гарантии того, что класс будет вести себя таким же образом, как и исходный. В частности это связано с тем, что изменение защиты данных класса влияет на наследование данных и методов другими классами.

Тестирование наследования – отдельная сложная задача в объектно-ориентированных системах. После того, как протестирован базовый класс, необходимо тестировать классы-потомки. Однако, для базового класса нельзя создавать заглушки, так как в этом случае можно пропустить возможные проблемы полиморфизма. Если класс-потомок использует методы базового класса для обработки собственных данных, необходимо убедиться в том, что эти методы работают.

Таким образом, иерархия классов может тестироваться сверху вниз, начиная от базового класса. Тестовое окружение при этом может меняться для каждой тестируемой конфигурации классов.

Вопросы для самопроверки

1. Что такое отладка программных средств?
2. Дайте определение тестированию программного средства.
3. Какие виды отладки вы знаете?
4. Перечислите основные принципы отладки программных средств.
5. Назовите основные достоинства и недостатки восходящего тестирования.
6. Назовите основные достоинства и недостатки нисходящего тестирования.
7. Что такое тестирование программного кода?
8. Охарактеризуйте процесс тестирования программного кода методом черного ящика.
9. В чем особенность тестирования программного кода методом прозрачного (белого) ящика.
10. Что такое драйвер и каковы его основные функции?
11. Что такое программная заглушка и каковы ее основные функции?
12. В чем основные особенности создания тестовых классов для тестирования объектно-ориентированных программ?

5. СОВМЕСТНАЯ РАЗРАБОТКА ПРОГРАММНЫХ СРЕДСТВ

5.1. Управление разработкой сложных программных систем

5.1.1. Обзор методик совместной разработки программного обеспечения

Как правило, сложные программные системы представляют собой многомодульные программы, построенные по иерархическому принципу, обладающие сложными связями между модулями. Разработка таких программ предполагает совместную работу коллектива разработчиков, причем эффективность разработки напрямую зависит от качества управления этим коллективом.

На сегодняшний день существует и активно применяется целый ряд методик коллективной разработки ПС. К ним относятся парное программирование, формальные инспекции, неформальные технические обзоры, чтение документации, а также другие методики, подразумевающие разделение ответственности за те или иные результаты работы между несколькими программистами.

Все методики совместного конструирования основаны на идее, что разработчикам трудно заметить дефекты в своей работе, и что каждый человек имеет свои недостатки, поэтому качество работы неизбежно повысится, если ее проверит кто-то другой. Исследования, проведенные в Институте разработки ПО (Software Engineering Institute), показали, что в среднем разработчики допускают от 1 до 3 дефектов в час при проектировании и от 5 до 8 дефектов в час при кодировании. Ясно, что устранение этих дефектов – обязательное условие эффективного конструирования программного обеспечения.

Совместное конструирование дополняет другие методики контроля качества. Главной целью совместного конструирования является повышение качества ПО. Само по себе тестирование ПО имеет довольно невысокую эффективность:

средний уровень определения дефектов равен примерно

- 30 % при блочном тестировании;
- 35 % при интеграционном тестировании;
- 35 % при ограниченном бета-тестировании.

В то же время средняя эффективность инспекций проектов и кода равна соответственно 55 % и 60 %. Дополнительное преимущество совместного конструирования состоит в том, что оно сокращает время разработки, что в свою очередь снижает расходы.

В IBM обнаружили, что каждый час инспекции предотвращал около 100 часов аналогичной работы (тестирования и исправления дефектов).

Самые разные исследования показали, что методики совместной разработки не только обеспечивают более высокую эффективность нахождения ошибок,

чем тестирование, но и позволяют находить типы ошибок, на которые тестирование указать не может.

Кроме того, программисты, знающие, что их работа будет подвергнута обзору, выполняют ее более добросовестно. Таким образом, даже при высокой эффективности тестирования в программу контроля качества следует включить обзоры или другие методики совместной разработки.

5.1.2. Парное программирование

При парном программировании один программист печатает код на клавиатуре, а второй следит за тем, чтобы в программу не вкрались ошибки, и думает о правильности кода в стратегическом масштабе. Первоначально парное программирование приобрело популярность благодаря экстремальному программированию, но теперь парное программирование используется более широко.

Для эффективного использования методики парного программирования необходимо следовать нескольким советам:

1. ***Поддерживайте парное программирование стандартами кодирования.*** Парное программирование не будет эффективным, если члены пары будут тратить время на споры о стиле кодирования.

2. ***Не позволяйте парному программированию превратиться в наблюдение.*** Член пары, не занимающийся непосредственно написанием кода, должен быть активным участником программирования. Он должен анализировать код, думать о том, что реализовать в следующую очередь, оценивать проект программы и планировать тестирование кода.

3. ***Не используйте парное программирование для реализации простых фрагментов.*** Члены одной группы, использовавшие парное программирование для написания наиболее сложного кода, обнаружили, что выгоднее посвятить 15 минут детальному проектированию на доске и затем программировать поодиночке.

4. ***Регулярно меняйте состав пар и назначаемые парам задачи.*** Как и при других методиках совместной разработки, при парном программировании выгода объясняется тем, что каждый из программистов изучает разные части системы.

5. ***Объединяйте в пару людей, предпочитающих одинаковый темп работы.*** Если один партнер работает слишком быстро, парное программирование начинает терять смысл. Более быстрый член пары должен снизить темп, или пару следует разбить и сформировать в другом составе.

6. ***Убедитесь, что оба члена пары видят экран.*** Эффективность парного программирования могут снижать даже такие, казалось бы, банальные вопросы, как неправильное расположение монитора и слишком мелкий шрифт.

7. ***Не объединяйте в пару людей, которые не нравятся друг другу.*** Эффективность работы в паре зависит от соответствия характеров двух программистов. Бессмысленно объединять в пару людей, которые плохо ладят друг с другом.

8. ***Не составляйте пару из людей, которые ранее не программировали в паре.*** Парное программирование приносит максимальную выгоду, если хотя бы один из партнеров имеет опыт работы в паре.

9. ***Назначьте лидера группы.*** Если все члены вашей группы хотят выполнить все программирование в парах, возложите на кого-то ответственность за распределение задач, контроль результатов и связь с людьми, не участвующими в проекте.

Парное программирование имеет целый ряд **достоинств:**

- в сравнении с одиночным программированием оно позволяет программистам успешнее противостоять стрессу. Члены пар поощряют друг друга поддерживать высокое качество кода даже в напряженных условиях, подталкивающих к быстрому написанию «грязного» кода;
- парное программирование повышает качество кода. Удобочитаемость и понятность кода всех программистов повышаются до уровня кода лучшего программиста группы;
- парное программирование ускоряет разработку системы. Как правило, пары пишут код быстрее, допуская при этом меньше ошибок. Соответственно в конце проекта группе приходится тратить меньше времени на исправление дефектов;
- парное программирование обеспечивает остальные общие преимущества совместного конструирования, такие как распространение корпоративной культуры, обучение неопытных программистов и содействие совместному владению результатами работы.

5.1.3. Формальные инспекции

Инспекцией называют специфический вид обзора, обеспечивающий очень высокую эффективность обнаружения дефектов и требующий меньших затрат, чем тестирование. Хотя любой обзор предполагает изучение проектов или кода, инспекция отличается от обыкновенного обзора несколькими важными аспектами:

- используемые при инспекциях контрольные списки концентрируют внимание инспекторов на областях, с которыми ранее были связаны проблемы;
- главной целью инспекции является обнаружение, а не исправление дефектов;

- люди, выполняющие инспекцию, готовятся к инспекционному собранию заблаговременно и прибывают на него со списком обнаруженных ими проблем;
- всем участникам инспекции назначаются конкретные роли;
- координатор инспекции *не является* автором продукта, подвергающегося инспекции;
- данные, полученные при каждой инспекции, используются для улучшения будущих инспекций.

Отдельные инспекции обычно приводят к обнаружению около 60 % дефектов, что превышает эффективность других методик, за исключением методики прототипирования и крупномасштабного бета-тестирования.

5.2. Базовые основы управления версиями программных средств

Совместная разработка программных средств предполагает, что каждый разработчик отдельно формирует некоторый результат своей работы, который необходимо корректно интегрировать в систему, а в случае возникновения ошибок – обеспечить возврат к предыдущей (корректной) версии программы. Если не управлять этим процессом, то неизбежно возникновение проблем. Например, чаще всего контроль версий программ начинающими программистами осуществляется путем копирования файлов программы в новый каталог с добавлением текущей даты к названию каталога.

Это может привести к негативным последствиям:

- случайному изменению не тех фалов, или изменению файлов не в том каталоге;
- ошибочному копированию файлов, и в результате затиранию нужных файлов;
- отсутствию информации об изменениях в программе от версии к версии.

5.2.1. Базовые термины контроля версий

Репозиторий это хранилище, в котором хранятся исходные коды программы, а также история их изменений.

Клиент – компьютер, который содержит свою локальную копию исходных кодов, с которой работает разработчик.

Изменения – набор изменений, именованный набор правок, сделанных в локальной копии для какой-либо цели.

Ревизия – очередная версия документа, новые изменения создают новую ревизию репозитория.

Метка – пометка начала отсчета изменений в дереве, групирует несколько файлов в пригодный для использования блок. Чаще всего используется для обозначения конечной версии файлов для сборки.

Три вида операций, выполняемых в системе управления версиями, могут приводить к необходимости объединения изменений:

1. ***Обновление рабочей копии (update)*** – изменения, сделанные в основной версии, сливаются с локальными, происходит синхронизация рабочей копии до некоторого заданного состояния хранилища (в том числе и к более старому состоянию, чем текущее).

2. ***Фиксация изменений (commit)*** – локальные изменения сливаются с изменениями, уже зафиксированными в основной версии.

3. ***Слияние ветвей (merge, integration)*** – объединение независимых изменений в единую версию документа – изменения, сделанные в одной ветви разработки, сливаются с изменениями, сделанными в другой.

Основные виды изменений:

- модификация содержимого файла;
- создание нового файла или каталога;
- удаление нового файла или каталога;
- переименование ранее существовавшего файла или каталога в проекте.

Механизм автоматического слияния изменений основывается на принципе:

> если два изменения относятся к разным и не связанным между собой файлам и/или каталогам, они всегда могут быть объединены автоматически. В этом случае изменения, сделанные в каждой версии проекта, копируются в объединяемую версию.

Конфликт *(conflict)* – ситуация, при которой слияние нескольких версий приводит к пересечению между собой сделанных в них изменений.

Конфликтующими обычно являются:

- удаление и изменение одного и того же файла или каталога;
- удаление и переименование одного и того же файла или каталога (в случае, если система поддерживает операцию переименования);
- создание в разных версиях файла с одним и тем же именем и разным содержимым;
- изменения в пределах одного текстового файла, сделанные в разных версиях, если они пересекаются;

- изменения в пределах одного файла, если он не является текстовым, всегда являются конфликтующими и не могут быть объединены автоматически.

Как правило, для разрешения конфликта требуется участие разработчика. При этом разработчику предлагается три варианта конфликтующих файлов:

- базовый;
- локальный;
- серверный.

Механизмом разрешения конфликтов является *механизм блокировки*. Он позволяет одному из разработчиков захватить в монопольное использование файл или группу файлов для внесения в них изменений. На то время, пока файл заблокирован, он остается доступным всем остальным разработчикам только на чтение, и любая попытка внести в него изменения отвергается сервером.

Пример необходимости использования блокировок – работа с бинарными файлами, для которых нет инструментальных средств слияния изменений либо такое слияние принципиально невозможно (как, например, для файлов изображений).

Патч (patch) – это файл, описывающий различие между файлами.

Дельта-компрессия – это способ хранения документов, при котором сохраняются только изменения между последовательными версиями, что позволяет уменьшить объем хранимых данных.

Ветвь (branch) – направление разработки, не зависимое от других. Представляет собой копию части хранилища, в которую можно вносить свои изменения, не влияющие на другие ветви.

Обычно механизм ветвей используется:

- при групповом режиме работы;
- для уменьшения риска потери изменений при локальных авариях;
- для обеспечения возможности анализа и возврата к предыдущим вариантам кода.

Ветвь представляет собой копию части (как правило, одного каталога) хранилища, в которую можно вносить свои изменения, не влияющие на другие ветви. Документы в разных ветвях имеют одинаковую историю до точки ветвления и разные – после нее. Изменения из одной ветви можно переносить в другую. Если изменения порождают новый вариант проекта, который далее развивается отдельно от основного, то ветвь может остаться самостоятельной.

К главным ветвям проекта относятся:

- *ствол (trunk, master)* – основная ветвь разработки проекта;
- *рабочая ветвь (develop).*

Схема процесса разработки программы с фиксацией изменений в главных ветках представлена на рис. 5.1. Ветвь *master* создается при инициализации репозитория для будущей разработки программного средства. В любой момент времени в ней должен находиться только стабильный код, готовый для безошибочной сборки. Данная ветвь считается главной, интеграционной.

Когда код в ветке *develop* становится пригодным для релиза (выпуска очередной версии программы), его интегрируют в ветвь *master* и помечают тегом версии.

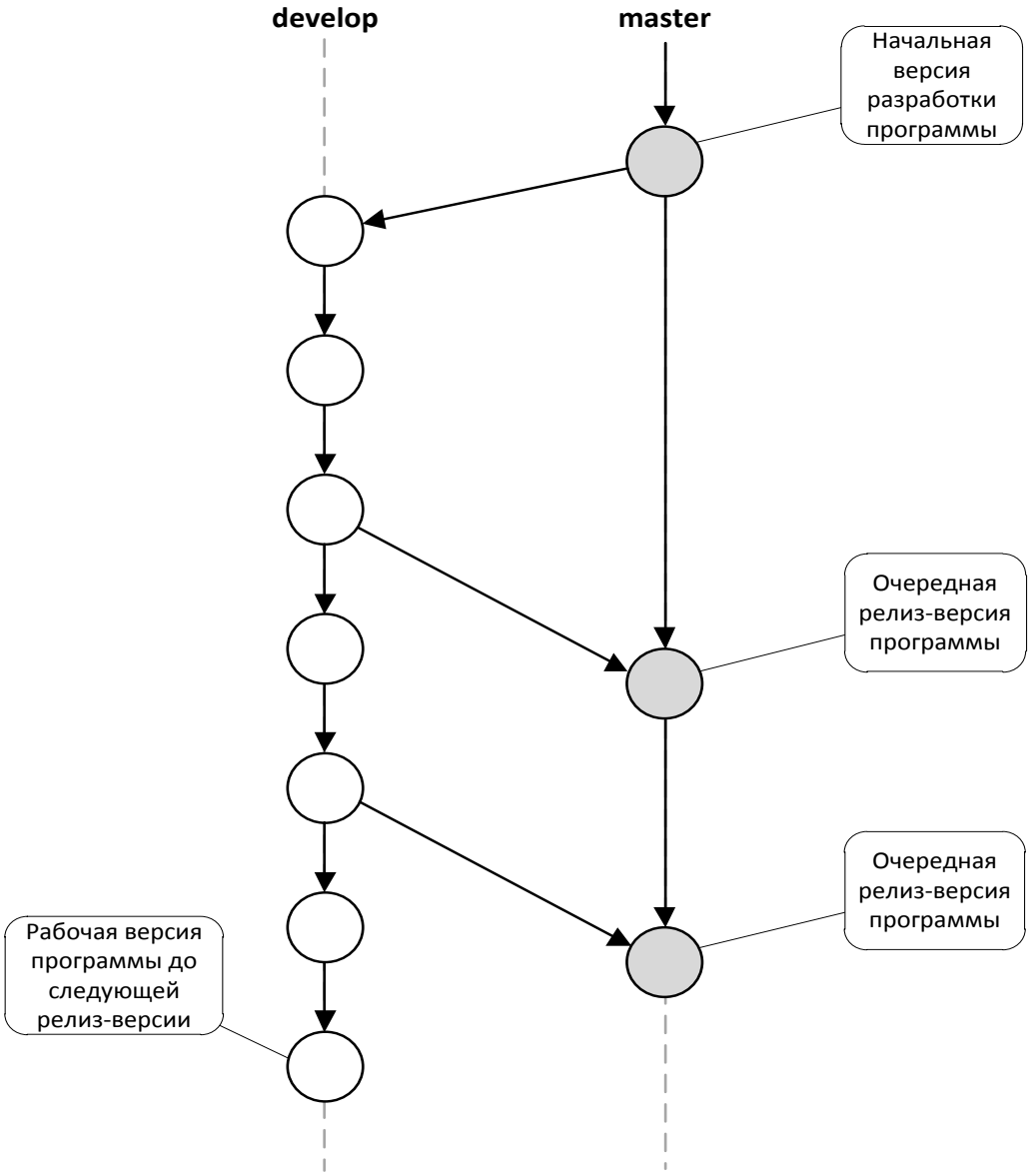

Рис. 5.1. Схема процесса разработки программы с фиксацией изменений в главных ветках

Кроме основных ветвей, существуют дополнительные:

- ветки функциональностей (Feature branches),
- ветки релизов (Release branches),
- ветки исправлений (Hotfix branches).

Основные особенности **веток функциональностей**:

- могут порождаться от *develop*;
- должны интегрироваться в *develop*;
- используются для разработки новых функций, которые должны появиться в текущем или следующем релизе;
- существуют столько, сколько функция (feature) разрабатывается.
- как правило, существуют в репозитории разработчиков, а не в центральном.

К основным особенностям **веток релизов** относятся следующие:

- могут порождаться от *develop*;
- должны интегрироваться в *develop*, *master*;
- название: *release*-*;
- используются для подготовки версий к релизу.

Характеристики **веток исправлений**:

- могут порождаться от *master*;
- должны интегрироваться в *develop*, *master*.
- название: *hotfix*-*;
- похожи на релизные ветки, отличие в том, что они не запланированные.
- создаются в момент, когда необходимо срочно исправить ошибку в готовом к выпуску коде;
- команда может работать над *develop* версией, а в это время в ветке исправлений кто-то пытается быстро устранить найденную ошибку.

5.2.2. Базовые принципы разработки программных средств с использованием систем контроля версий

Последовательность работы над проектом с использованием систем контроля версий следующая.

Начало работы с проектом заключается в извлечении из хранилища рабочей копии проекта или той его части, с которой предстоит работать. Для извлечения рабочей копии проекта используются команды *checkout* или *clone*.

Ежедневный цикл работы с проектом представлен на рис. 5.2.

Порядок использования системы управления версиями определяется техническими регламентами и правилами, принятыми в конкретной организации, разрабатывающей проект.

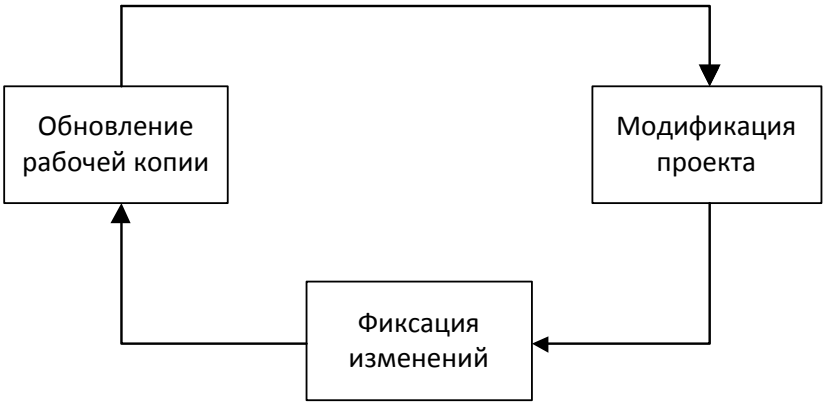

Рис. 5.2. Ежедневный цикл работы с использованием системы контроля версий

Общие принципы правильного использования систем контроля версий

1. Любые рабочие, тестовые или демонстрационные версии проекта собираются только из репозитория системы.

2. Текущая версия главной ветви всегда корректна.

3. Не допускается фиксация в главной ветви неполных или не прошедших хотя бы предварительное тестирование изменений.

4. В любой момент сборка проекта, проведенная из текущей версии, должна быть успешной.

5. Любое значимое изменение должно оформляться как отдельная ветвь. Промежуточные результаты работы разработчика фиксируются в этой ветви. После завершения работы над изменением ветвь объединяется со стволом. Исключения допускаются только для мелких изменений, работа над которыми ведется одним разработчиком в течение не более чем одного рабочего дня.

6. Версии проекта помечаются тегами. После этого выделенная и помеченная тегом версия не должна изменяться.

5.2.3. Принципы версионности программных средств

Каждая версия программы, готовая к выпуску, помечается тегом, который может записываться:

- целым числом (**Windows 8**);
- дробным числом (**Sybase Powerdesigner 15.3**);
- последовательностью чисел (**JDK 1.0.3, Skype 6.6.0.106**);
- годом (**Windows 2000**);
- текстом (**Embarcadero RAD Studio XE**).

Пример

Adobe Photoshop CS6 – торговое название, v13.1.2 – внутренняя версия.

Рассмотрим основные правила нумерации версий программного обеспечения. Номер версии записывается в виде последовательности **A.B.C.D[r]**, где:

- A – главный номер версии (major version number);
- B – вспомогательный номер версии (minor version number);
- C – номер сборки, номер логической итерации по работе над функционалом версии A.B (build number);
- D – номер ревизии, сквозной номер, назначаемый автоматически программным обеспечением хранения версий (SVN);
- [r] – условное обозначение релиза (промышленного издания);

Промышленное издание – это стабильная версия программы, которая готова к тиражированию. ПО соответствует всем требованиям качества и готово для массового распространения.

5.3. Системы контроля версий

5.3.1. Основные понятия

Система контроля (управления) версиями (от англ. **Version Control System** (**VCS**) или **Revision Control System**) – это программное обеспечение для облегчения работы с изменяющейся информацией.

Основные функции системы контроля версий

- хранение нескольких версий одного и того же документа (история версий);
- хранение истории разработки;
- при необходимости возвращение к более ранним версиям документа (отмена изменений);
- определение, кто и когда сделал изменение (поиск «виновного»);
- совмещение изменений сделанных разными разработчиками (синхронизация работы команды);
- реализация альтернативных/экспериментальных вариантов проекта.

Области применения систем контроля версий

1. Разработка программного обеспечения – хранение исходных кодов разрабатываемой программы.

2. Области, в которых ведется работа с большим количеством непрерывно изменяющихся электронных документов:

- в инструментах конфигурационного управления (**Software Configuration Management Tools**);
- в системах автоматизированного проектирования (**САПР**), обычно в составе систем управления данными об изделии (**Product Data Management** (**PDM**)).

В качестве примера можно привести интернет-ресурс «Википедия», история изменений в котором для всех материалов статей ведется на основе методов, аналогичных тем, которые применяются в системах управления версиями.

5.3.2. Классификация систем контроля версий

По степени централизации системы контроля версий условно делятся на следующие типы:

- локальные (например, **RSC**) (рис. 5.3);
- централизованные (например, **CVS, Subversion** и **Perforce**);
- децентрализованные (например, **Git, Mercurial, Bazaar, Darcs**).

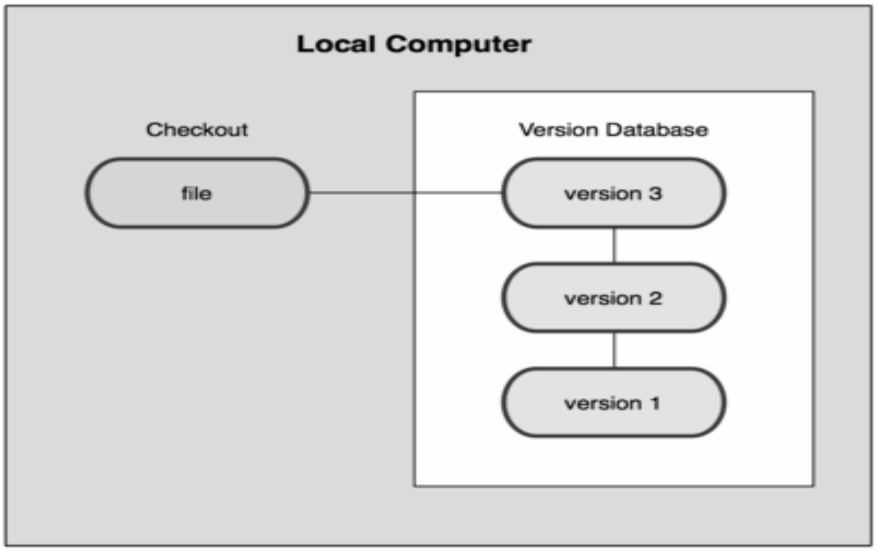

Рис. 5.3. Схема работы локальной системы контроля версий

Другим основанием для классификации систем контроля версий является возможность блокирования файлов во время работы с ними. В соответствии с данным основанием системы контроля версий делятся на:

- ***блокирующие*** – позволяют наложить запрет на изменение файла, пока один из разработчиков работает над ним;
- ***не блокирующие*** – один файл может одновременно изменяться несколькими разработчиками.

По отношению к форматам хранимых данных системы контроля версий делятся на следующие виды:

- предназначенные для работы с текстовыми данными (очень важна поддержка слияния изменений);
- работающие с бинарными данными (важна возможность блокировки).

Наиболее важным основанием для выбора той или иной системы контроля версий при разработке программного обеспечения является первое из рассмотренных, поэтому коснемся его детальнее.

Централизованные системы контроля версий. Особенностью таких систем является то, что они позволяют организовать процесс сотрудничества между разработчиками. Все файлы, находящиеся под версионным контролем, хранятся на центральном сервере. Совокупность клиентов взаимодействуют с сервером, работая с локальной копией необходимого файла (рис. 5.4).

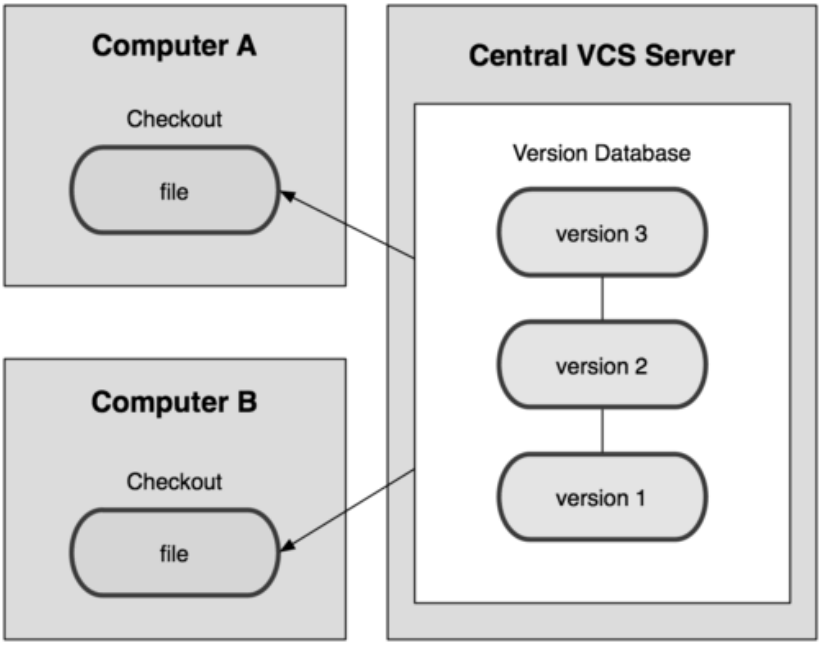

Рис. 5.4. Схема работы централизованной системы контроля версий

Основные **достоинства** централизованной СКВ:

- все знают, кто и чем занимается в проекте;
- администраторы осуществляют контроль над тем, кто и что может делать.

Главным **недостатком** централизованной СКВ является то, что централизованный сервер является уязвимым местом всей системы.

Таким образом, если:

- сервер не работает, то разработчики не могут взаимодействовать, и никто не может сохранить новой версии своей работы;
- отсутствует подключение к сети у разработчика, в это время он также не может взаимодействовать с сервером;

- повреждается диск с центральной базой данных, и нет резервной копии, следовательно, теряется практически все – вся история проекта, разве что за исключением нескольких рабочих версий, сохранившихся на рабочих машинах пользователей.

Распределенные системы контроля версий. В таких системах клиенты не просто выгружают последние версии файлов, как в централизованных, а полностью копируют весь репозиторий (рис. 5.5).

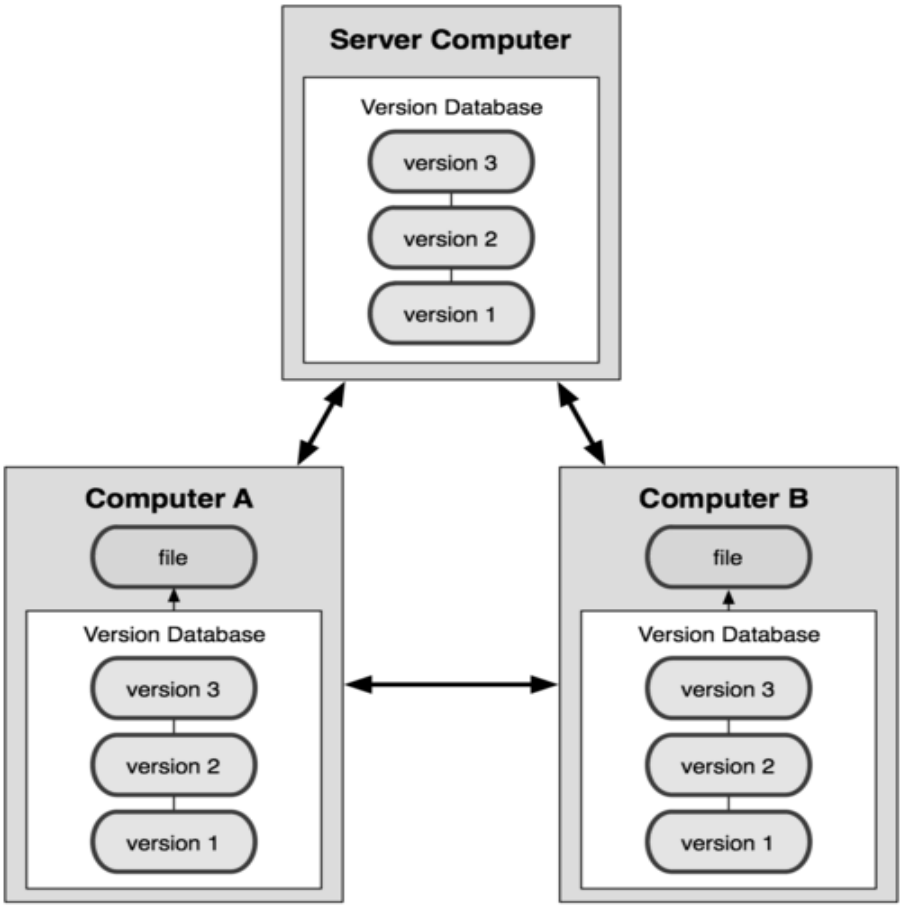

Рис. 5.5. Схема работы распределенной системы контроля версий

К основным **достоинствам** распределенной СКВ относятся следующие:

1. Так как каждый раз, когда клиент забирает свежую версию файлов, он создает себе полную копию всех данных, то в случае сбоев на сервере, через который шла работа, ***любой клиентский репозиторий может быть скопирован*** (возвращен на сервер для восстановления базы данных).

2. ***Возможность работы с несколькими удаленными репозиториями*** – можно одновременно работать по-разному с разными группами людей в рамках одного проекта. Так, в одном проекте можно одновременно вести несколько типов рабочих процессов, что невозможно в централизованных системах.

5.3.3. Обзор систем контроля версий

RCS (Revision Control System) – система управления пересмотрами версий. На данный момент RCS активно вытесняется более мощной системой контроля версий CVS, но все еще достаточно популярна, и является частью проекта GNU.

RCS позволяет работать только с отдельными файлами, создавая для каждого историю изменений. Для текстовых файлов сохраняются не все версии файла, а только последняя версия и все изменения, внесенные в нее. RCS также может отслеживать изменения в бинарных файлах, но при этом каждое изменение хранится в виде отдельной версии файла.

Когда изменения в файл вносит один из пользователей, для всех остальных этот файл остается заблокированным. Они не могут запросить его из репозитория для редактирования, пока первый пользователь не закончит работу и не зафиксирует изменения.

Рассмотрим основные преимущества и недостатки системы контроля версий RCS.

Преимущества:

- RCS проста в использовании и удобна для ознакомления с принципами работы систем контроля версий;
- хорошо подходит для резервного копирования отдельных файлов, не требующих частого изменения группой пользователей;
- широко распространена и предустановлена в большинстве свободно распространяемых операционных систем.

Недостатки:

- отслеживает изменения только отдельных файлов, что не позволяет ее использовать для управления версиями больших проектов;
- не позволяет одновременно вносить изменения в один и тот же файл несколькими пользователями;
- низкая функциональность, по сравнению с современными системами контроля версий.

CVS – система управления параллельными версиями. Система управления параллельными версиями (**Concurrent Versions System**) – логическое развитие системы управления пересмотрами версий (**RCS**), использующая ее стандарты и алгоритмы по управлению версиями, но значительно более функциональная, и позволяющая работать не только с отдельными файлами, но и с целыми проектами.

CVS основана на технологии *клиент-сервер*, взаимодействующих по сети. Клиент и сервер также могут располагаться на одной машине, если над проек-

том работает только один человек, или требуется вести локальный контроль версий.

Работа CVS организована следующим образом. Последняя версия и все сделанные изменения хранятся в репозитории сервера. Клиенты, подключаясь к серверу, проверяют отличия локальной версии от последней версии, сохраненной в репозитории, и, при обнаружении отличий, загружают их в свой локальный проект. Исходя из необходимости, решают конфликты и вносят требуемые изменения в разрабатываемый продукт. После этого все изменения загружаются в репозиторий сервера. CVS, при необходимости, позволяет «откатываться» на нужную версию разрабатываемого проекта и вести управление несколькими проектами одновременно.

Приведем основные достоинства и недостатки системы управления параллельными версиями.

Достоинства:

- несколько клиентов могут одновременно работать над одним и тем же проектом;
- система позволяет управлять не одним файлом, а целыми проектами;
- система обладает большим количеством удобных графических интерфейсов, способных удовлетворить практически любой, даже самый требовательный вкус;
- система широко распространена и поставляется по умолчанию с большинством операционных систем Linux;
- при загрузке тестовых файлов из репозитория передаются только изменения, а не весь файл целиком.

Недостатки:

- при перемещении или переименовании файла или директории теряются все изменения, привязанные к этому файлу или директории;
- сложности при ведении нескольких параллельных веток одного и того же проекта;
- ограниченная поддержка шрифтов;
- для каждого изменения бинарного файла сохраняется вся версия файла, а не только внесенное изменение;
- с клиента на сервер измененный файл всегда передается полностью;
- выполняемые операции требуют частого обращения к репозиторию, поэтому обладают значительной ресурсоемкостью.

Система управления версиями Subversion. Это централизованная система управления версиями, созданная в 2000 году и основанная на технологии клиент-сервер. Она обладает всеми достоинствами CVS и решает основные ее про-

блемы (переименование и перемещение файлов и каталогов, работа с двоичными файлами и т.д.). Часто ее называют по имени клиентской части – **SVN.**

Принцип работы с Subversion очень походит на работу с CVS. Клиенты копируют изменения из репозитория и объединяют их с локальным проектом пользователя. Если возникают конфликты локальных изменений и изменений, сохраненных в репозитории, то такие ситуации разрешаются вручную. Затем в локальный проект вносятся изменения, и полученный результат сохраняется в репозитории.

При работе с файлами, не позволяющими объединять изменения, может использоваться следующая последовательность действий:

1. Файл скачивается из репозитория и блокируется (запрещается его скачивание из репозитория).
2. Вносятся необходимые изменения.
3. Загружается файл в репозиторий и разблокируется (разрешается его скачивание из репозитория другим клиентам).

Достоинства:

- система команд, схожая с CVS;
- поддерживается большинство возможностей CVS;
- разнообразные графические интерфейсы и удобная работа из консоли;
- отслеживается история изменения файлов и каталогов даже после их переименования и перемещения;
- высокая эффективность работы, как с текстовыми, так и с бинарными файлами;
- встроенная поддержка во многие интегрированные средства разработки, такие как **KDevelop, NetBeans, MS Visual Studio** и многие другие;
- возможность создания зеркальных копий репозитория;
- два типа репозитория – база данных или набор обычных файлов;
- возможность доступа к репозиторию через *Apache* с использованием протокола *WebDAV*;
- наличие удобного механизма создания меток и ветвей проектов;
- наличие возвожности с каждым файлом и директорией связать определенный набор свойств, облегчающий взаимодействие с системой контроля версии;
- широкое распространение позволяет быстро решить большинство возникающих проблем, обратившись к данным, накопленным Интернет-сообществом.

Недостатки:

- полная копия репозитория хранится на локальном компьютере в скрытых файлах, что требует достаточно большого объема памяти;

- существуют проблемы с переименованием файлов, если переименованный локально файл одним клиентом был в это же время изменен другим клиентом и загружен в репозиторий;
- слабо поддерживаются операции слияния веток проекта;
- сложности с полным удалением информации о файлах, попавших в репозиторий, так как в нем всегда остается информация о предыдущих изменениях файла, и непредусмотрено никаких штатных средств для полного удаления данных о файле из репозитория.

Система управления версиями Git. С февраля 2002 года для разработки ядра Linux большинством программистов стала использоваться система контроля версий BitKeeper. Довольно долгое время с ней не возникало проблем, но в 2005 году *Ларри Маквой* (разработчик BitKeeper'а) отозвал бесплатную версию программы.

Разрабатывать проект масштаба Linux без мощной и надежной системы контроля версий невозможно. Одним из кандидатов и наиболее подходящим проектом оказалась система контроля версий Monotone, но *Торвальдса Линуса* не устроила скорость ее работы. Так как особенности организации Monotone не позволяли значительно увеличить скорость обработки данных, то 3 апреля 2005 года Линус приступил к разработке собственной системы контроля версий – Git. И практически одновременно с Линусом (на три дня позже) к разработке новой системы контроля версий приступил и *Мэтт Макал* – свой проект он назвал Mercurial.

Git – это гибкая, распределенная (без единого сервера) система контроля версий, дающая массу возможностей не только разработчикам программных продуктов, но и писателям для изменения, дополнения и отслеживания изменения «рукописей» и сюжетных линий, и учителям для корректировки и развития курса лекций, и администраторам для ведения документации, и для многих других направлений, требующих управления историей изменений. У каждого разработчика, использующего Git, есть свой локальный репозиторий, позволяющий локально управлять версиями. Затем, сохраненными в локальном репозитории данными можно обмениваться с другими пользователями. Часто при работе с Git создают ***центральный репозиторий***, с которым остальные разработчики синхронизируются.

Пример организации системы с центральным репозиторием – это проект разработки ядра Linux (http://www.kernel.org). В этом случае все участники проекта ведут свои локальные разработки и беспрепятственно скачивают обновления из центрального репозитория. Когда необходимые работы отдельными участниками проекта выполнены и отлажены, они, после удостоверения вла-

дельцем центрального репозитория в корректности и актуальности проделанной работы, загружают свои изменения в центральный репозиторий.

Наличие локальных репозиториев также значительно повышает надежность хранения данных, так как, если один из репозиториев выйдет из строя, данные могут быть легко восстановлены из других репозиториев.

Работа над версиями проекта в Git может вестись в нескольких ветках, которые затем с легкостью полностью или частично объединяются, уничтожаются, откатываются или разрастаются во все новые и новые ветки проекта.

Можно долго обсуждать возможности Git'а, но для краткости и более простого восприятия приведем основные достоинства и недостатки этой системы управления версиями.

Достоинства:

- надежная система сравнения ревизий и проверки корректности данных, основанные на алгоритме хеширования SHA1 (**Secure Hash Algorithm 1**);
- гибкая система ветвления проектов и слияния веток между собой;
- наличие локального репозитория, содержащего полную информацию обо всех изменениях, позволяет вести полноценный локальный контроль версий и заливать в главный репозиторий только полностью прошедшие проверку изменения;
- высокая производительность и скорость работы;
- удобный и интуитивно понятный набор команд;
- множество графических оболочек, позволяющих быстро и качественно вести работы с Git'ом;
- возможность делать контрольные точки, в которых данные сохраняются без использования дельта-компрессии, то есть полностью. Это позволяет уменьшить скорость восстановления данных, так как за основу берется ближайшая контрольная точка, и восстановление идет от нее. Если бы контрольные точки отсутствовали, то восстановление больших проектов могло бы занимать часы;
- широкая распространенность, легкая доступность и качественная документация;
- гибкость системы позволяет удобно ее настраивать и даже создавать специализированные контроля системы или пользовательские интерфейсы на базе Git;
- универсальный сетевой доступ с использованием протоколов http, ftp, rsync, ssh и др.

Недостатки:

- ориентированность на платформу Unix. На данный момент отсутствует зрелая реализация Git, совместимая с другими операционными системами;
- возможные (но чрезвычайно низкие) совпадения хеш-кода отличных по содержанию ревизий;
- не отслеживается изменение отдельных файлов, а только всего проекта целиком, что может быть неудобно при работе с большими проектами, содержащими множество несвязных файлов;
- при начальном (первом) создании репозитория и синхронизации его с другими разработчиками, потребуется достаточно длительное время для скачивания данных, особенно, если проект большой, так как требуется скопировать на локальный компьютер весь репозиторий.

Система управления версиями Mercurial. Первоначально она была создана для эффективного управления большими проектами под управлением операционной системы Linux, а поэтому была ориентирована на быструю и надежную работу с большими репозиториями. На данный момент Mercurial адаптирован для работы под Windows, Mac OS X и большинство Unix систем.

Большая часть системы контроля версий написана на языке Python, и только отдельные участки программы, требующие наибольшего быстродействия, написаны на языке Си.

Идентификация ревизий происходит на основе алгоритма хеширования SHA1 (Secure Hash Algorithm 1), однако, также предусмотрена возможность присвоения ревизиям индивидуальных номеров.

Так же, как и в Git, поддерживается возможность создания веток проекта с последующим их слиянием.

Для взаимодействия между клиентами используются протоколы HTTP, HTTPS или SSH.

Набор команд – простой и интуитивно понятный, во многом схожий с командами Subversion. Так же имеется ряд графических оболочек и доступ к репозиторию через веб-интерфейс. Немаловажным является и наличие утилит, позволяющих импортировать репозитории многих других систем контроля версий.

Рассмотрим основные достоинства и недостатки Mercurial.

Достоинства:

- быстрая обработка данных;
- кросплатформенность;
- возможность работы с несколькими ветками проекта;
- простота использования;

- возможность конвертирования репозиториев других систем контроля версий, таких как **CVS, Subversion, Git, Darcs, GNU Arch, Bazaar** и др.

Недостатки:

- возможные (но чрезвычайно низкие) совпадения хеш-кода отличных по содержанию ревизий.
- ориентированность на работу в консоли.

Система управления версиями Bazaar. Это распределенная, свободно распространяемая система контроля версий, разрабатываемая при поддержке компании Canonical Ltd, написанная на языке Python и работающая под управлением операционных систем **Linux, Mac OS X** и **Windows**. В отличие от **Git** и **Mercurial**, создаваемых для контроля версий ядра операционной системы Linux, а поэтому ориентированных на максимальное быстродействие при работе с огромным числом файлов, **Bazaar** ориентировался на удобный и дружественный интерфейс пользователя. Оптимизация скорости работы производилось уже на втором этапе, когда первые версии программы уже появились.

Как и во многих других системах контроля версий, система команд **Bazaar** очень похожа на команды CVS или Subversion и обеспечивает удобный, простой и интуитивно понятный интерфейс взаимодействия с программой.

Значительное внимание уделяется работе с ветками проектов (создание, объединение веток и т.д.), что очень важно при разработке серьезных проектов и позволяет проводить доработки и эксперименты без угрозы потери основной версии программного обеспечения. Большой плюс этой системе контроля версий дает возможность работы с репозиториями других систем контроля версий, таких как **Subversion** или **Git.** Кратко приведем наиболее значительные достоинства и недостатки этой системы контроля версий.

Достоинства:

- кросплатформенность;
- удобный и интуитивно понятный интерфейс;
- простая работа с ветками проекта;
- возможность работы с репозиториями других систем контроля версий;
- подробная документация;
- удобный графический интерфейс;
- чрезвычайная гибкость, позволяющая подстроится под нужды конкретного пользователя.

Недостатки:

- более низкая скорость работы, по сравнению с Git и Mercurial, но эта ситуация постепенно исправляется;

- для полноценного функционирования необходимо устанавливать достаточно большое количество плагинов, позволяющих полностью раскрыть все возможности системы контроля версий.

Большой выбор систем контроля версий позволяет удовлетворить любые требования и организовать работу так, как необходимо разработчику. Однако среди всего многообразия систем существуют явные лидеры. Так, если необходимо управлять огромным проектом, состоящим из десятков тысяч файлов и над которым работу ведут тысячи человек, то лучше всего выбор остановить на **Git *или* Mercurial**. Если же главное – удобный интерфейс, а разрабатываемый проект средних размеров, то в этом случае предпочтительна система **Bazaar.** Для программистов-одиночек или небольших проектов, не требующих ветвления и создания множества версий, лучше всего подойдет **Subversion**. Но, в конечном итоге, выбор – дело вкуса, так как сейчас существует множество систем контроля версий, которые дадут все необходимое. Системы контроля версий – это абсолютно необходимое программное обеспечения для каждого разработчика и не только.

Вопросы для самопроверки

1. Перечислите основные методики совместной разработки программных средств.
2. В чем суть методики парного программирования?
3. Каковы основные достоинства парного программирования?
4. Перечислите основные аспекты проведения формальных инспекций программного кода.
5. Что такое репозиторий?
6. Дайте определение системы контроля версий.
7. Перечислите основные функции систем контроля версий.
8. Приведите варианты классификации систем контроля версий по разным основаниям.
9. Назовите основные современные системы контроля версий.
10. Каковы основные достоинства системы контроля версий Git?

6. УПРАВЛЕНИЕ КАЧЕСТВОМ ПРОГРАММНОГО СРЕДСТВА

6.1. Понятие «качество программных средств»

В настоящее время разработка ПС достигла такого уровня развития, при котром уже никто не оспаривает объективной необходимости использования инженерных методов, в том числе для оценивания результатов проектирования на этапах ЖЦ, контроля достижения показателей качества и метрического их анализа, оценки риска и степени использования готовых компонентов для снижения стоимости разработки нового проекта.

Основу инженерных методов в программировании составляет повышение качества, для достижения которого сформировались методы определения требований к качеству, подходы к выбору и усовершенствованию моделей метрического анализа показателей качества, методы количественного измерения показателей качества на этапах ЖЦ.

Главная составляющая качества – ***надежность,*** которой уделяется наибольшое внимание в области надежности технических средств и тех критических систем (систем реального времени, радарных систем, систем безопасности и др.), для которых *надежность* является главной целевой *функцией* оценки их реализации. Как следствие, в проблематике надежности разработано более сотни математических моделей, являющихся функциями от ошибок, оставшихся в ПС, от ***интенсивности отказов*** или частоты появления дефектов в ПС. На их основе производится оценка надежности программной системы.

Качество *ПО* – предмет стандартизации.

Стандарт ГОСТ 2844-94 дает определение качества *ПО* как совокупности свойств (показателей качества) *ПО,* которые обеспечивают его способность удовлетворять потребности заказчика в соответствии с назначением. Этот стандарт регламентирует базовую модель качества и показатели, главным среди которых выделяет *надежность*.

Стандарт ISO/IEC 12207 определил не только основные процессы ЖЦ разработки ПС, но и организационные и дополнительные процессы, которые регламентируют инженерию планирования и управления качеством ПС.

Согласно стандарту на этапах ЖЦ должен проводиться ***контроль качества ПО:***

- проверка соответствия требований проектируемому продукту и критериев их достижения;

- верификация и аттестация (валидация) промежуточных результатов ПО на этапах ЖЦ и измерение степени удовлетворения достигаемых отдельных показателей;
- тестирование готовой ПС, сбор данных об отказах, дефектах и других ошибках, обнаруженных в системе;
- подбор моделей надежности для оценивания надежности по полученным результатам тестирования (дефекты, отказы и др.);
- оценка показателей качества, заданных в требованиях на разработку ПС.

Далее излагаются модели качества и надежности, а также способы их применения.

6.2. Модель качества программного обеспечения

Качество *ПО* – это понятие относительное, которое имеет смысл только при учете реальных условий его применения, поэтому требования, предъявляемые к качеству, ставятся в соответствии с условиями и конкретной областью их применения. Оно характеризуется тремя аспектами:

- качеством программного продукта,
- качеством процессов ЖЦ
- качеством сопровождения или внедрения (рис. 6.1).

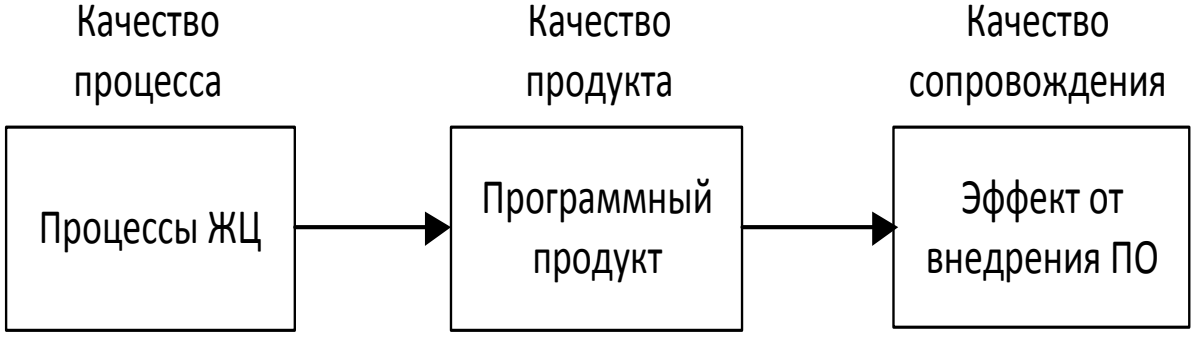

Рис. 6.1. Основные аспекты качества ПО

Аспект, связанный с ***процессами ЖЦ***, определяет степень формализации, достоверности самих процессов ЖЦ разработки ПО, а также верификацию и валидацию промежуточных результатов на этих процессах. Поиск и устранение ошибок в готовом ПО проводится методами тестирования, которые снижают количество ошибок и повышают качество этого продукта.

Качество продукта достигается процедурами контроля промежуточных продуктов на процессах ЖЦ, их проверкой на достижение необходимого качества, а также методами сопровождения продукта.

Эффект от внедрения ПС в значительной степени зависит от знаний обслуживающего персонала функций продукта и правил их выполнения.

Модель качества ПО имеет следующие четыре уровня представления.

Первый уровень

соответствует определению характеристик (показателей) качества ПО, каждая из которых отражает отдельную точку зрения пользователя на качество. В модель качества входит шесть характеристик или шесть показателей качества:

- функциональность (functionality);
- надежность (realibility);
- удобство (usability);
- эффективность (efficiency);
- сопровождаемость (maitainnability);
- переносимость (portability).

Второму уровню

соответствуют атрибуты для каждой характеристики качества, которые детализируют разные аспекты конкретной характеристики. Набор атрибутов характеристик качества используется при оценке качества.

Третий уровень

предназначен для измерения качества с помощью метрик, каждая из которых определяется как комбинация метода измерения атрибута и шкалы измерения значений атрибутов. Для оценки атрибутов качества на этапах ЖЦ (при просмотре документации, программ и результатов тестирования программ) используются метрики с заданным оценочным весом для нивелирования результатов метрического анализа совокупных атрибутов конкретного показателя и качества в целом. *Атрибут* качества определяется с помощью одной или нескольких методик оценки на этапах ЖЦ и на завершающем этапе разработки ПО.

Четвертый уровень

это оценочный элемент метрики (*вес*), который используется для оценки количественного или качественного значения отдельного атрибута показателя ПО. В зависимости от назначения, особенностей и условий сопровождения ПО выбираются наиболее важные характеристики качества и их атрибуты (рис. 6.2).

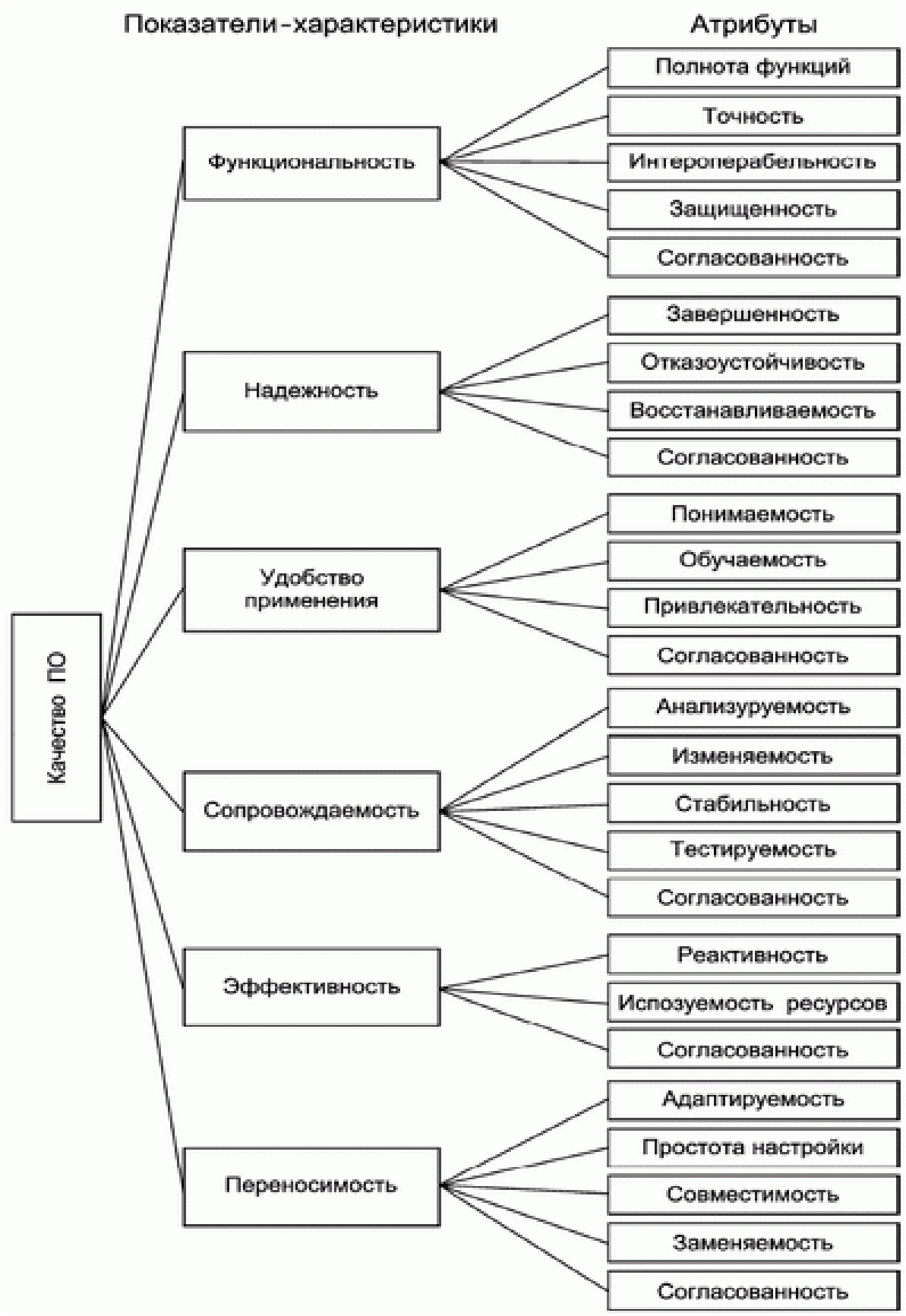

Рис. 6.2. Модель характеристик качества программного обеспечения

Выбранные атрибуты и их приоритеты отражаются в требованиях на разработку систем либо используются соответствующие приоритеты эталона класса ПО, к которому это ПО относится.

6.3. Характеристика показателей качества ПО

Краткое описание семантики характеристик модели качества приведены в табл. 6.1, а их содержательное описание – ниже.

Таблица 6.1

Краткая характеристика показателей качества ПО

Показатель	Описание свойств показателя
Функциональность	Группа свойств ПО, обуславливающая его способность выполнять определенный перечень функций, которые удовлетворяют потребностям в соответствии с назначением
Надежность	Группа свойств, обусловливающая способность ПО сохранять работоспособность и преобразовывать исходные данные в результат за установленный период времени, характер отказов которого является следствием внутренних дефектов и условий его применения
Удобство применения	Совокупность свойств ПО для предполагаемого круга пользователей и отражающих легкость его освоения и адаптации к изменяющимся условиям эксплуатации, стабильность работы и подготовки данных, понимаемость результатов, удобства внесения изменений в программную документацию и в программы
Сопровождаемость	Группа свойств, определяющая усилия, необходимые для выполнения, приспособленность к диагностике отказов и последствий внесения изменений, модификации и аттестации модифицируемого ПО
Рациональность	Группа свойств, характеризующаяся степенью соответствия используемых ресурсов среды функционирования уровню качества (надежности) функционирования ПО при заданных условиях применения
Переносимость	Группа свойств ПО, обеспечивающая его приспособленность для переноса из одной среды функционирования в другие, усилия для переноса и адаптацию ПО к новой среде функционирования

6.3.1. Функциональность

Функциональность – это совокупность свойств, определяющих способность программного обеспечения выполнять перечень функций в заданной среде и, в соответствии с требованиями, к обработке и общесистемным средствам. Под *функцией* понимается некоторая упорядоченная последовательность действий для удовлетворения потребительских свойств. Функции бывают целевые (основные) и вспомогательные.

К атрибутам функциональности относятся:

- *функциональная полнота* – это свойство компонента, которое показывает степень достаточности основных функций для решения задач в соответствии с назначением ПО;
- *правильность (точность)* – это атрибут, который показывает степень достижения правильных результатов;
- *интероперабельность* – это атрибут, который показывает возможность взаимодействия программного обеспечения со специальными системами и средами (ОС, сеть);
- *защищенность* – это атрибут, который характеризует способность ПО предотвращать несанкционированный доступ (случайный или умышленный) к программам и данным.

6.3.2. Надежность

Надежность – это совокупность атрибутов, которые определяют способность ПО преобразовывать исходные данные в результаты при условиях, зависящих от периода времени жизни (износ и его старение не учитываются). Снижение надежности ПО происходит из-за ошибок в требованиях, проектировании и выполнении.

К характеристикам надежности ПО относятся:

- *безотказность* – атрибут, который определяет способность ПО функционировать без отказов (как программы, так и оборудования);
- *устойчивость к ошибкам* – атрибут, который характеризует способность ПО выполнять функции при аномальных условиях (сбой аппаратуры, ошибки в данных и интерфейсах, нарушение в действиях оператора и др.);
- *восстанавливаемость* – атрибут, который определяет способность программы к перезапуску для повторного выполнения и восстановления данных после отказов.

К некоторым типам систем (системам реального времени, радарным системам, системам безопасности, коммуникации и др.) предъявляются требования

для обеспечения высокой надежности (недопустимость ошибок, точность, достоверность, удобство применения и др.). Таким образом, надежность ПО в значительной степени зависит от числа оставшихся и не устраненных ошибок в процессе разработки на этапах ЖЦ. В ходе эксплуатации ошибки обнаруживаются и устраняются.

Если при исправлении ошибок не вносятся новые или, по крайней мере, новых ошибок вносится меньше, чем устраняется, то в ходе эксплуатации надежность ПО непрерывно возрастает. Чем интенсивнее проводится эксплуатация, тем интенсивнее выявляются ошибки и быстрее растет надежность ПО.

К факторам, влияющим на надежность ПО, относятся:

- совокупность угроз, приводящих к неблагоприятным последствиям и к ущербу системы или среды ее функционирования;
- угроза как проявление нарушения безопасности системы;
- целостность как способность системы сохранять устойчивость работы и не иметь риска.

Обнаруженные ошибки могут быть результатом угрозы извне или отказов, они повышают риск и уменьшают некоторые свойства надежности системы.

Надежность – одна из ключевых проблем современных программных систем, и ее роль будет постоянно возрастать, поскольку постоянно повышаются требования к качеству компьютерных систем. Новое направление – инженерия программной надежности (*Software reliability* engineering) – ориентировано на количественное изучение операционного поведения компонентов системы по отношению к пользователю, ожидающему надежную работу системы, и включает:

- измерение надежности, то есть проведение ее количественной оценки с помощью предсказаний, сбора данных о поведении системы в процессе эксплуатации и современных моделей надежности;
- стратегии и метрики конструирования и выбора готовых компонентов, процесс разработки компонентной системы, а также среда функционирования, влияющая на надежность работы системы;
- применение современных методов инспектирования, верификации, валидации и тестирования при разработке систем, а также при эксплуатации.

Верификация применяется для определения соответствия готового ПО установленным спецификациям, а *валидация* – для установления соответствия системы требованиям пользователя, которые были предъявлены заказчиком.

В инженерии надежности термин ***dependability*** (пригодноспособность) обозначает способность системы иметь свойства, желательные для пользовате-

ля, который уверен в качественном выполнении функций ПС, заданных в требованиях.

Данный термин определяется дополнительным количеством атрибутов, которыми должна обладать система, а именно:

- готовность к использованию (*availability*);
- готовностью к непрерывному функционированию (*reliability*);
- безопасность для окружающей среды, то есть способность системы не вызывать катастрофических последствий в случае отказа (*safety*);
- секретность и сохранность информации (*confidential*);
- способность к сохранению системы и устойчивости к самопроизвольному ее изменению (*integrity*);
- способность к эксплуатации ПО, простота выполнения операций обслуживания, а также устранения ошибок, восстановление системы после их устранения и т.п. (*maintainability*);
- готовность и сохранность информации (*security*) и др.

Достижение надежности системы обеспечивается предотвращением отказа (*fault prevention*), его устранением (*removal fault*), а также оценкой возможности появления новых отказов и мер борьбы с ними с применением методов теории вероятности.

6.3.3. Удобство применения

Удобство применения характеризуется множеством атрибутов, которые показывают на необходимые и пригодные условия использования (диалоговое или не диалоговое) ПО заданным кругом пользователей для получения соответствующих результатов. Удобство применения определено как специфическое множество атрибутов программного продукта, характеризующих его эргономичность.

К характеристикам удобства применения относятся:

- *понимаемость* – атрибут, определяющий усилия, затрачиваемые на распознавание логических концепций и условий применения ПО;
- *изучаемость (легкость изучения)* – атрибут, определяющий усилия пользователей на определение применимости ПО путем использования операционного контроля, диагностики, а также процедур, правил и документации;
- *оперативность* – атрибут, определяющий реакцию системы при выполнении операций и операционного контроля;
- *согласованность* – атрибут, характеризующий соответствие разработки требованиям стандартов, соглашений, правил, законов и предписаний.

6.3.4. Эффективность

Эффективность – это множество атрибутов, которые определяют взаимосвязь уровней выполнения ПО, использования ресурсов (средства, аппаратура, материалы – бумага для печатающего устройства и др.) и услуг, выполняемых штатным обслуживающим персоналом и др.

К характеристикам эффективности ПО относятся:

- *реактивность* – атрибут, характеризующий время отклика, обработки и выполнения функций;
- *эффективность ресурсов* – атрибут, определяющий количество и продолжительность используемых ресурсов при выполнении функций ПО;
- *согласованность* – это атрибут, который определяет соответствие данного атрибута с заданными стандартами, правилами и предписаниями.

6.3.5. Сопровождаемость

Сопровождаемость – это множество свойств, описывающих усилия, которые надо затратить на проведение модификаций, включающих корректировку, усовершенствование и адаптацию ПО при изменении среды, требований или функциональных спецификаций.

Сопровождаемость включает такие характеристики, как:

- *анализируемость* – атрибут, определяющий необходимые усилия для диагностики отказов или идентификации частей, которые будут модифицироваться;
- *изменяемость* – атрибут, который определяет удаление ошибок в ПО или внесение изменений для их устранения, а также введение новых возможностей в ПО или в среду функционирования;
- *стабильность* – атрибут, указывающий на постоянство структуры и риск ее модификации;
- *тестируемость* – атрибут, характеризующий усилия при проведении валидации и верификации с целью обнаружения несоответствий требованиям, а также в случае необходимости проведения модификации ПО и сертификации;
- *согласованность* – атрибут, который показывает соответствие данного атрибута соглашениям, правилам и предписаниям стандарта.

6.3.6. Переносимость

Переносимость – это множество показателей, указывающих на способность ПО адаптироваться к работе в новых условиях среды выполнения. Среда может быть организационной, аппаратной и программной. Поэтому перенос ПО в но-

вую среду выполнения может быть связан с совокупностью действий, направленных на обеспечение его функционирования в среде, отличной от той, в которой оно создавалось с учетом новых программных, организационных и технических возможностей.

Переносимость включает следующие характеристики:

- ***адаптивность*** – это атрибут, определяющий усилия, затрачиваемые на адаптацию к различным средам;
- ***настраиваемость (простота инсталляции)*** – это атрибут, который определяет необходимые усилия для запуска данного ПО в специальной среде;
- ***сосуществование*** – атрибут, который определяет возможность использования специального ПО в среде действующей системы;
- ***заменяемость*** – это атрибут, который обеспечивают возможность интероперабельности при совместной работе с другими программами с необходимой инсталляцией или адаптацией ПО;
- ***согласованность*** – это атрибут, который показывает на соответствие стандартам или соглашениями по обеспечению переноса ПО.

6.4. Метрики качества программного обеспечения

В настоящее время в программной инженерии еще не сформировалась окончательно система метрик. Действуют разные подходы к определению их набора и методов измерения.

Система измерения включает метрики и модели измерений, которые используются для количественной оценки качества ПО.

При определении требований к ПО задаются соответствующие им внешние характеристики и их атрибуты (характеристики), определяющие разные стороны управления продуктом в заданной среде.

Для набора характеристик качества ПО, приведенных в требованиях, определяются соответствующие метрики, модели их оценки и диапазон значений мер для измерения отдельных атрибутов качества.

Метрики определяются по модели измерения атрибутов ПО на всех этапах ЖЦ (промежуточная, внутренняя метрика) и особенно на этапе тестирования или функционирования (внешние метрики) продукта.

Остановимся на классификации метрик ПО, правилах для проведения метрического анализа и процесса их измерения.

Существует три типа метрик:

1) метрики *программного продукта*, которые используются при измерении его характеристик – свойств;

2) метрики *процесса*, которые используются при измерении свойства процесса ЖЦ создания продукта.

3) метрики *использования*.

Метрики программного продукта включают

- **внешние метрики**, обозначающие свойства продукта, видимые пользователю, а именно:
  - ✓ метрики надежности продукта (служат для определения числа дефектов);
  - ✓ метрики функциональности (с их помощью устанавливаются наличие и правильность реализации функций в продукте);
  - ✓ метрики сопровождения (благодаря им измеряются ресурсы продукта – скорость, память, среда);
  - ✓ метрики применимости продукта (способствуют определению степени доступности для изучения и использования;
  - ✓ метрики стоимости (ими определяется стоимость созданного продукта);
- **внутренние метрики**, обозначающие свойства, видимые только команде разработчиков, а именно:
  - ✓ метрики размера, необходимые для измерения продукта с помощью его внутренних характеристик;
  - ✓ метрики сложности, необходимые для определения сложности продукта;
  - ✓ метрики стиля, которые служат для определения подходов и технологий создания отдельных компонентов продукта и его документов.

Внутренние метрики позволяют определить производительность продукта и являются релевантными по отношению к внешним метрикам.

Внешние и внутренние метрики задаются на этапе формирования требований к ПО и являются предметом планирования и управления достижением качества конечного программного продукта.

Метрики продукта часто описываются комплексом моделей для установки различных свойств, значений модели качества или прогнозирования.

Измерения проводятся, как правило, после калибровки метрик на ранних этапах проекта.

Общая мера – степень трассируемости, которая определяется числом трасс, прослеживаемых с помощью моделей сценариев типа UML и оценкой количества:

- требований;
- сценариев и действующих лиц;
- объектов, включенных в сценарий, и локализация требований к каждому сценарию;
- параметров и операций объекта и др.

При оценке общего количества некоторых величин часто используются среднестатистические метрики (среднее число операций в классе, наследников класса или операций класса и др.).

Как правило, меры в значительной степени являются субъективными и зависят от знаний экспертов, производящих количественные оценки атрибутов компонентов программного продукта.

Примером широко используемых внешних метрик программ являются *метрики Холстеда* – характеристики программ, выявляемые на основе статической структуры программы на конкретном языке программирования:

- число вхождений наиболее часто встречающихся операндов и операторов;
- длина описания программы как сумма числа вхождений всех операндов и операторов и др.

На основе этих атрибутов можно вычислить время программирования, уровень программы (структурированность и качество) и языка программирования (абстракции средств языка и ориентация на проблему) и др.

В качестве ***метрик процесса*** могут быть время разработки, число ошибок, найденных на этапе тестирования и др. Практически используются следующие метрики процесса:

- общее время разработки и отдельно время для каждой стадии;
- время модификации моделей;
- время выполнения работ на процессе;
- число найденных ошибок при инспектировании;
- стоимость проверки качества;
- стоимость процесса разработки.

Метрики использования служат для измерения степени удовлетворения потребностей пользователя при решении его задач. Они помогают оценить эксплуатационное качество, то есть не свойства самой программы, а результаты ее эксплуатации. Примерами могут служить – точность и полнота реализации задач пользователя, а также ресурсы, затраченные на эффективное решение задач

пользователя (трудозатраты, производительность и др.). *Оценка требований* пользователя проводится с помощью внешних метрик.

Вопросы для самопроверки

1. Что такое качество программного обеспечения?
2. Что является главной составляющей качества программного обеспечения?
3. Каковы основные аспекты качества программного обеспечения?
4. Перечислите основные уровни представления качества программного обеспечения.
5. Какие характеристики входят в модель качества программного обеспечения?
6. Что такое функциональность программного обеспечения?
7. Дайте определение надежности программного обеспечения.
8. Охарактеризуйте такой показатель качества программного обеспечения как сопровождаемость.
9. Перечислите основные типы метрик качества программного обеспечения.
10. Что такое *метрики Холстеда*?

ЗАКЛЮЧЕНИЕ

Современная программная инженерия (**Software Engineering**) – молодая и быстро развивающаяся область знаний и практик. Она ориентирована на комплексное решение задач, связанных с разработкой особой разновидности сложных систем – программных.

Программные системы – самое необычное и удивительное создание рук человеческих. Они не имеют материальных тел, их нельзя потрогать, ощутить ни одним из органов чувств. Они не подвергаются физическому износу, их нельзя изготовить в обычном инженерном смысле, как автомобиль на заводе. И вместе с тем разработка программных систем является одной из самых сложных задач, которые когда-либо приходилось решать инженеру.

Современное общество впадает во все большую зависимость от программных технологий, и их знание и умение применять на практике позволяют разрабатывать все больше новых и полезных программных систем.

В настоящем учебном пособии рассмотрены вопросы конструирования программных средств и систем, особенности их тестирования и отладки, а также оценивания качества программного обеспечения.

Конечно, из-за ресурсных ограничений за рамками пособия осталось множество технологий и средств разработки. В частности, не рассмотрены особенности разработки **Web-приложений**, подходы к оцениванию стоимости разработки программных систем и многое другое.

Указанная отрасль стремительно развивается: появляются и пропадают новые языки программирования, новые средства разработки и проектирования программ. Для того чтобы быть в курсе этих изменений, требуются целенаправленное и регулярное изучение современной литературы по этой тематике и совершенствование своих навыков конструирования программных систем.

СПИСОК ЛИТЕРАТУРЫ

1. Технология разработки программного обеспечения: учеб. пособие / В. В. Бахтизин, Л. А. Глухова. – Минск: БГУИР, 2010. – 267 с.

2. *Орлов С.А.* Технологии разработки программного обеспечения: учебник. – СПб.: Питер, 2002. – 464 с.

3. *Макконнелл С.* Совершенный код. Мастер-класс / пер. с англ. – М.: Издательство «Русская редакция», 2010. – 896 стр.

4. *Брукс Ф.* Мифический человеко-месяц или как создаются программные системы / пер. с англ. – СПб.: Символ-Плюс, 1999.

5. *Гайсарян С.С.* Объектно-ориентированные технологии проектирования прикладных программных систем. – Центр Информационных Технологий, http://citmgu.ru, 1998.

6. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / пер. с англ. – 2-е изд. – М.: Бином; СПб.: Невский диалект, 1998.

7. *Раскин Д.* Интерфейс: новые направления в проектировании компьютерных систем / пер. с англ. – СПб.: Символ-Плюс, 2003.

8. *Соммервилл И.* Инженерия программного обеспечения / пер. с англ. – 2-е изд. – М.: Вильямс, 2002.

9. *Фаулер М.* UML. Основы /пер. с англ. – 3-е изд. – СПб: Символ-Плюс, 2006. – 192 с.

10. *Якобсон А., Буч Г., Рамбо Д.* Унифицированный процесс разработки программного обеспечения / пер. с англ. – СПб.: Питер, 2002.

11. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес; пер. с англ. – СПб.: Питер, 2014.

12. *Лефингвел Д., Уидриг Д.* Принципы работы с требованиями к программному обеспечению. Унифицированный подход / пер. с англ. – М.: Вильямс, 2002.