К.Ф.КЕРИМОВ, Ш.К.КАМАЛОВ,
Ш.Ш.МУХСИНОВ, Ф.С.АГЗАМОВ

ВВЕДЕНИЕ В ПРОГРАММНЫЙ ИНЖИНИРИНГ

МИНИСТЕРСТВО РАЗВИТИЯ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ
И КОММУНИКАЦИЙ РЕСПУБЛИКИ УЗБЕКИСТАН

ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ
ТЕХНОЛОГИЙ ИМЕНИ МУХАММАДА АЛ-ХОРЕЗМИ

К.Ф.КЕРИМОВ, Ш.К.КАМАЛОВ,
Ш.Ш.МУХСИНОВ, Ф.С.АГЗАМОВ

ВВЕДЕНИЕ В ПРОГРАММНЫЙ ИНЖИНИРИНГ

Рекомендовано
Министерством высшего и среднего специального образования
Республики Узбекистан в качестве учебного пособия
для студентов высших учебных заведений.

Ташкент – 2019

УДК: 004.41
ББК: 32.973.-01

К.Ф.Керимов, Ш.К.Камалов, Ш.Ш.Мухсинов, Ф.С.Агзамов "Введение в программный инжиниринг". Учебное пособие /ТУИТ. 220 с. Ташкент, 2019

ISBN 978–9943–5642–8–2

Данное пособие представляет собой информацию о программной инженерии в виде законченного пособия. В каждой главе представлены вопросы, относящиеся к созданию программного обеспечения. В пособии систематически изложены методы программирования, их теория и практика с учетом ядра знаний SWEBOK (SoftwareEngineeringofBodyKnowledge) и положений стандартов программной инженерии. Представлены методы прикладного и теоретического проектирования, методы доказательства, верификация и тестирование, а также методы интеграции и преобразования программ и данных. Определены основы инженерной дисциплины разработки – управление проектом, риском и качеством. Описана инженерия приложений и предметной области на основе повторного использования компонентов, определены подходы и методы их аннотации для накопления, выбора и оценки применимости в новых программных проектах.

Данное пособие предназначен для студентов факультета программной инженерии, разработчиков и докторантов в области программирования, которые могут ознакомиться с систематизированными знаниями по современным методам анализа, проектирования, интеграции и тестирования, а также по методам инженерии программирования – управление проектом, рисками и качеством проектируемых систем.

УДК: 004.41
ББК: 32.973.-01

Рецензенты:

Латипова Н.Х. – кандидат технических наук, доцент кафедры «Системное и прикладное программирование» Ташкентского университета информационных технологий;

Худайбердиев М.Х. – кандидат технических наук, Старший научный сотрудник «Центр разработки программных продуктов и аппаратно-программных комплексов».

ISBN 978–9943–5642–8–2

ВВЕДЕНИЕ

Постановление Президента Республики Узбекистан Ш. Мирзиёев 15 марта 2017 г., № ПП-2834 «О мерах по дальнейшему совершенствованию деятельности Ташкентского университета информационных технологий» открывает Великий ученый Востока Мухаммад ал-Хоразмий, живший и творивший в VIII-IX веках, является автором научных трудов по математике, геометрии, астрономии, истории, географии и другим наукам. В настоящее время все вычислительные операции, выполняемые в сфере информационно-коммуникационных технологий на основе высоких технологий, базируются на научных открытиях Мухаммада ал-Хоразмий.

“Факультет «Программный инжиниринг» создан на основе Постановления Президента Республики Узбекистан №ПП-1942 «О мерах по дальнейшему совершенствованию системы подготовки кадров в области информационно-коммуникационных технологий» от 26 марта 2013 года. Факультет готовит специалистов бакалавриата и магистратуры по специальности управлений и разработке программного обеспечения компьютерных и мобильных устройств, по сохранению, обработке и передаче данных, по разработке программ для компьютерных и телекоммуникационных систем и сетей, а также программ по управлению базами данных.

Программная инженерия (программный инжиниринг) обычно ассоциируется с разработкой больших и сложных программ коллективами разработчиков. Становление и развитие этой области деятельности было вызвано рядом проблем, связанных с высокой стоимостью программного обеспечения, сложностью его создания, необходимостью управления и прогнозирования процессов разработки.

Сам термин – software engineering (программная инженерия) - впервые был озвучен в октябре 1968 года на конференции подкомитета НАТО по науке и технике (г. Гармиш, Германия). Присутствовало 50 профессиональных разработчиков ПО из 11 стран. Рассматривались проблемы проектирования, разработки, распространения и поддержки программ. Там впервые и прозвучал термин «программная инженерия» как название дисциплина, которую надо создавать и которой надо руководствоваться в решении

перечисленных проблем. Вскоре после этого в Лондоне состоялась встреча 22-х руководителей проектов по разработке ПО. На встрече анализировались проблемы и перспективы развития ПО. Отмечалась возрастающее воздействие ПО на жизнь людей. Впервые серьезно заговорили о надвигающемся кризисе ПО. Применяющиеся принципы и методы разработки ПО требовали постоянного усовершенствования. Именно на этой встрече была предложена концепция жизненного цикла ПО (SLC – Software Lifetime Cycle) как последовательности шагов-стадий, которые необходимо выполнить в процессе создания и эксплуатации ПО. Вокруг этой концепции было много споров. В 1970 г. У.У. Ройс (W.W. Royce) произвел 5 идентификацию нескольких стадий в типичном цикле и было высказано предположение, что контроль выполнения стадий приведет к повышению качества ПО и сокращению стоимости разработки.

Программная инженерия – это область компьютерной науки и технологии, которая занимается построением программных систем, настолько больших и сложных, что для этого требуется участие слаженных *команд разработчиков различных специальностей и квалификаций*. Обычно такие системы существуют и применяются долгие годы, развиваясь от версии к версии, претерпевая на своем жизненном пути множество изменений, улучшение существующих функций, добавление новых или удаление устаревших возможностей, адаптацию для работы в новой среде, устранение дефектов и ошибок. *Суть методологии программной инженерии* состоит в применении систематизированного, научного и предсказуемого процесса проектирования, производства и сопровождения программных комплексов реального времени.

Массовое создание сложных программных средств промышленными методами и большими коллективами специалистов вызвало необходимость их четкой организации, планирования работ по требуемым ресурсам, этапам и срокам реализации. Для решения этих задач в программной инженерии формируется новая область знания и научная дисциплина – *экономика жизненного цикла программных средств*, как часть экономики промышленности и вычислительной техники в общей экономике государств и предприятий. Объективно положение осложнено трудностью измерения экономических

характеристик таких объектов. Широкий спектр количественных и качественных показателей, которые с различных сторон характеризуют содержание этих объектов, и невысокая достоверность оценки их значений, определяют значительную дисперсию при попытках описать и измерить экономические свойства создаваемых или используемых крупных комплексов программ. Вследствие роста сфер применения и ответственности функций, выполняемых программами, резко возросла необходимость ***гарантирования высокого качества программных продуктов***, регламентирования и корректного формирования требований к характеристикам реальных комплексов программ и их достоверного определения. В результате специалисты в области теории и методов, определяющих качество продукции, вынуждены осваивать область развития и применения нового, специфического продукта – программных средств и систем в целом, и их качество при использовании. Сложность анализируемых объектов – комплексов программ и психологическая самоуверенность ряда программистов в собственной «непогрешимости», часто приводят к тому, что реальные характеристики качества функционирования программных продуктов остаются неизвестными не только для заказчиков и пользователей, но также для самих разработчиков. Отсутствие четкого декларирования в документах понятий и требуемых значений характеристик качества программных комплексов вызывает конфликты между заказчиками-пользователями и разработчиками-поставщиками из-за разной трактовки одних и тех же характеристик.

Цель данного пособия - представить программную инженерию в виде целостного изложения, концентрируясь на концепции процесса, различных методологиях разработки ПО, отдельных видах деятельности процесса - разработке архитектуры, конфигурационном управлении, работе с требованиями, тестировании. В стороне умышленно оставлены вопросы, собственно, программирования, поскольку в рамках общего курса их невозможно эффективно рассмотреть.

В пособии систематически изложены методы программирования, их теория и практика с учетом ядра знаний SWEBOK (SoftWare Engineering of Body Knowledge) и положений стандартов программной инженерии. Представлены методы

прикладного и теоретического проектирования, методы доказательства, верификация и тестирование, а также методы интеграции и преобразования программ и данных. Определены основы инженерной дисциплины разработки – управление проектом, риском и качеством. Описана инженерия приложений и предметной области на основе повторного использования компонентов, определены подходы и методы их аннотации для накопления, выбора и оценки применимости в новых программных проектах.

Для студентов факультета программной инженерии, разработчиков и докторантов в области программирования, желающих ознакомиться с систематизированными знаниями по современным методам анализа, проектирования, интеграции и тестирования, а также по методам инженерии программирования – управление проектом, рисками и качеством проектируемых систем.

ГЛАВА 1. ПРОЦЕСС РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1. Процесс создания программного обеспечения

Как мы работаем, какова последовательность наших шагов, каковы нормы и правила в поведении и работе, каков регламент отношений между членами команды, как проект взаимодействует с внешним миром и т.д.? Все это вместе мы склонны называть процессом. Его осознание, выстраивание и улучшение - основа любой эффективной групповой деятельности. Поэтому не случайно, что процесс оказался одним из основных понятий программной инженерии.

Центральным объектом изучения программной инженерии является **процесс** создания ПО – множество различных видов деятельности, методов, методик и шагов, используемых для разработки и эволюции ПО и связанных с ним продуктов (проектных планов, документации, программного кода, тестов, пользовательской документации и пр.).

Однако на сегодняшний день не существует **универсального процесса** разработки ПО – набора методик, правил и предписаний, подходящих для ПО любого вида, для любых компаний, для команд любой национальности. Каждый **текущий процесс** разработки, осуществляемый некоторой командой в рамках определенного проекта, имеет большое количество особенностей и индивидуальностей. Однако целесообразно перед началом проекта спланировать процесс работы, определив роли и обязанности в команде, рабочие продукты (промежуточные и финальные), порядок участия в их разработке членов команды и т.д. Будем называть это предварительное описание **конкретным процессом**, отличая его от плана работ, проектных спецификаций и пр. Например, в системе Microsoft Visual Team System оказывается шаблон процесса, создаваемый или адаптируемый (в случае использования стандартного) перед началом разработки. В VSTS существуют заготовки для конкретных процессов на базе CMMI, Scrum и др.

В рамках компании возможна и полезна стандартизация всех текущих процессов, которую будем называть **стандартным**

процессом. Последний, таким образом, оказывается некоторой базой данных, содержащей следующее:

- информацию, правила использования, документацию и инсталляционные пакеты средств разработки, используемых в проектах компании (систем версионного контроля, средств контроля ошибок, средств программирования – различных IDE, СУБД и т.д.);
- описание практик разработки – проектного менеджмента, правил работы с заказчиком и т.д.;
- шаблоны проектных документов – технических заданий, проектных спецификаций, планов тестирования и т.д. и пр.

Также возможна стандартизация процедуры разработки конкретного процесса как "вырезки" из стандартного. Основная идея стандартного процесса – курсирование внутри компании передового опыта, а также унификация средств разработки. Очень уж часто в компаниях различные департаменты и проекты сильно отличаются по зрелости процесса разработки, а также затруднено повторное использование передового опыта. Кроме того, случается, что компания использует несколько средств параллельных инструментов разработки, например, СУБД средства версионного контроля. Иногда это бывает оправдано (например, таковы требования заказчика), часто это необходимо – например, Java, .NET (большая компетентность оффшорной компании позволяет ей брать более широкий спектр заказов). Но очень часто это произвольный выбор самих разработчиков. В любом случае, такая множественность существенно затрудняет миграцию специалистов из проекта в проект, использование результатов одного проекта в другом и т.д. Однако при организации стандартного процесса необходимо следить, чтобы стандартный процесс не оказался всего лишь формальным, бюрократическим аппаратом. Понятие стандартного процесса введено и подробно описано в подходе CMMI.

Необходимо отметить, что наличие стандартного процесса свидетельствует о наличии "единой воли" в организации, существующей именно на уровне процесса. На уровне продаж, бухгалтерии и др. привычных для всех компаний процессов и активов единство осуществить не трудно. А вот на уровне процессов разработки очень часто каждый проект оказывается сам по себе

(особенно в оффшорных проектах) – "текучка" захватывает и изолирует проекты друг от друга очень прочно.

1.2. Совершенствование процесса

Определение. Совершенствование процесса (software process improvement) – это деятельность по изменению существующего процесса (как текущего, в рамках одного проекта, так и стандартного, для всей компании) с целью улучшения качества создаваемых продуктов и/или снижения цены и времени их разработки. Причины актуальности этой деятельности для компаний-производителей ПО заключается в следующем.

1. Происходит быстрая смена технологий разработки ПО, требуются изучение и внедрение новых средств разработки.
2. Наблюдается быстрый рост компаний и их выход на новые рынки, что требует новой организации работ.
3. Имеет место высокая конкуренция, которая требует поиска более эффективных, более экономичных способов разработки.
4. Что и каким образом можно улучшать.
5. Переход на новые средства разработки, языки программирования и т.д.
6. Улучшение отдельных управленческих и инженерных практик – тестирования, управления требованиями и пр.
7. Полная, комплексная перестройка всех процессов в проекте, департаменте, компании (в соответствии, например, с CMMI).
8. Сертификация компании (CMM/CMMI, ISO 9000 и пр.).

Мы отделили п. 3 от п. 4 потому, что на практике 4 далеко не всегда означает действительную созидательную работу по улучшению процессов разработки ПО, а часто сводится к поддержанию соответствующего документооборота, необходимого для получения сертификации. Сертификат потом используется как средство, козырь в борьбе за заказы.

Главная трудность реального совершенствования процессов в компании заключается в том, что она при этом должна работать и создавать ПО, ее нельзя "закрыть на учет".

Отсюда вытекает идея непрерывного улучшения процесса, так сказать, малыми порциями, чтобы не так болезненно. Это тем более

разумно, что новые технологии разработки, появляющиеся на рынке, а также развитие уже существующих нужно постоянно отслеживать. Эта стратегия, в частности, отражена в стандарте совершенствования процессов разработки CMMI.

Pull/Push стратегии. В контексте внедрения инноваций в производственные процессы бизнес-компаний (не обязательно компаний по созданию ПО) существуют две следующие парадигмы.

1. Organization pull – инновации нацелены на решение конкретных проблем компании.
2. Technology push – широкомасштабное внедрение инноваций из стратегических соображений. Вместо конкретных проблем, которые будут решены после внедрения инновации, в этом случае рассматриваются показатели компании (эффективность, производительность, годовой оборот средств, увеличение стоимости акций публичной компании), которые будут увеличены, улучшены после внедрения инновации. При этом предполагается, что будут автоматически решены многочисленные частные проблемы организации, в том числе и те, о которых в данный момент ничего не известно.

Пример использования стратегии organization pull – внедрение новых средств тестирования в ситуации, когда высоки требования по качеству в проекте, либо когда качество программной системы не удовлетворяет заказчика.

Пример использования стратегии technology push – переход компании со средств структурной разработки на объектно-ориентированные. Еще один пример использования той же стратегии – внедрение стандартов качества ISO 9000 или CMMI. В обоих этих случаях компания не решает какую-то одну проблему или ряд проблем – она хочет радикально изменить ситуацию, выйти на новые рубежи и т.д.

Проблемы применения стратегии technology push в том, что требуется глобальная перестройка процесса. Но компанию нельзя "закрыть на реконструкцию" – за это время положение на рынке может оказаться занято конкурентами, акции компании могут упасть и т.д. Таким образом, внедрение инноваций, как правило, происходит параллельно с обычной деятельностью компании, поэтапно, что в

случае с technology push сопряжено с большими трудностями и рисками.

Использование стратегии organization pull менее рискованно, вносимые ею изменения в процесс менее глобальны, более локальны. Но и выгод такие инновации приносят меньше, по сравнению с удачными внедрениями в соответствии со стратегией technology push.

Необходимо также отметить, что существуют проблемы, которые невозможно устранить точечными переделками процесса, то есть необходимо применять стратегию technology push. Приведем в качестве примера зашедший в тупик процесс сопровождения и развития семейства программных продуктов – компания терпит большие убытки, сопровождая уже поставленные продукты, инструментальные средства проекта, безнадежно устарели и находятся в плачевном состоянии, менеджмент расстроен, все попытки руководства изменить процесс наталкиваются на непонимание коллектива, ссоры и конфликты. Возможно, что в таком случае без "революции" не обойтись.

Еще одно различие обеих стратегий: в случае с organization pull, как правило, возврат инвестиций от внедрения происходит быстрее, чем в случае с technology push.

1.3. Классические модели процесса

Определение модели процесса. Процесс создания программного обеспечения не является однородным. Тот или иной метод разработки ПО, как правило, определяет некоторую динамику развертывания тех или иных видов деятельности, то есть, определяет *модель процесса* (process model).

Модель является хорошей абстракцией различных методов разработки ПО, позволяя лаконично, сжато и информативно их представить. Однако, сама идея модели процесса является одной из самых ранних в программной инженерии, когда считалось, что удачная модель – самое главное, что способствует успеху разработки. Позднее пришло осознание, что существует множество других аспектов (принципы управления и разработки, структуру команды и т.д.), которые должны быть определены, согласовано друг с другом. И стали развиваться интегральные методологии разработки. Тем не

менее существует несколько классических моделей процесса, которые полезны на практике и которые будут рассмотрены ниже.

Фазы и виды деятельности. Говоря о моделях процессов, необходимо различать фазы и виды деятельности.

Фаза (phase) – это определенный этап процесса, имеющий начало, конец и выходной результат. Например, фаза проверки осуществимости проекта, сдачи проекта и т.д. Фазы следуют друг за другом в линейном порядке, характеризуются предоставлением отчетности заказчику и, часто, выплатой денег за выполненную часть работы.

Редко какой заказчик согласится первый раз увидеть результаты только после завершения проекта. С другой стороны, подрядчики предпочитают получать деньги постепенно, по мере того, как выполняются отдельные части работы. Таким образом, появляются фазы, позволяющие создавать и предъявлять промежуточные результаты проекта. Фазы полезны также безотносительно взаимодействия с заказчиком – с их помощью можно синхронизировать деятельность разных рабочих групп, а также отслеживать продвижение проекта. Примерами фаз может служить согласование с заказчиком технического задания, реализация определенной функциональности ПО, этап разработки, оканчивающийся сдачей системы на тестирование или выпуском альфа-версии.

Вид деятельности (activity) – это определенный тип работы, выполняемый в процессе разработки ПО. Разные виды деятельности часто требуют разные профессиональные навыки и выполняются разными специалистами. Например, управление проектом выполняется менеджером проекта, кодирование – программистом, тестирование – тестировщиком. Есть виды деятельности, которые могут выполняться одними и теми же специалистами – например, кодирование и проектирование (особенно в небольшом проекте) часто выполняют одни и те же люди.

В рамках одной фазы может выполняться много различных видов деятельности. Кроме того, один вид деятельности может выполняться на разных фазах – например, тестирование: на фазе анализа и проектирования можно писать тесты и налаживать тестовое окружение, при разработке и перед сдачей производить, собственно,

само тестирование. На настоящий момент для сложного программного обеспечения используются многомерные модели процесса, в которых отделение фаз от видов деятельности существенно облегчает управление разработкой ПО.

Виды деятельности, фактически, присутствуют, под разными названиями, в каждом методе разработки ПО. В RUP они называются рабочими процессами (work flow), в CMM – ключевыми областями процесса (key process area). Мы будем сохранять традиционные названия, принятые в том или ином методе, чтобы не создавать путаницы.

Водопадная модель была предложена в 1970 году Винстоном Ройсом. Фактически, впервые в процессе разработки ПО были выделены различные шаги разработки и поколеблены примитивные представления о разработке ПО в виде анализа системы и ее кодирования.

Были определены следующие шаги: разработка системных требований, разработка требований к ПО, анализ, проектирование, кодирование, тестирование.

Достоинством этой модели явилось ограничение возможности возвратов на произвольный шаг назад, например, от тестирования – к анализу, от разработки – к работе над требованиями и т.д. Отмечалось, что такие возвраты могут катастрофически увеличить стоимость проекта и сроки его выполнения. Например, если при тестировании обнаруживаются ошибки проектирования или анализа, то их исправление часто приводит к полной переделке системы. Этой моделью допускались возвраты только на предыдущий шаг, например, от тестирования к кодированию, от кодирования к проектированию и т.д.

Наконец, в рамках этой модели было введено прототипирование, то есть предлагалось разрабатывать систему дважды, чтобы уменьшить риски разработки. Первая версия – прототип – позволяет увидеть основные риски и обосновано принять главные архитектурные решения. На создание прототипа отводилось до одной трети времени всей разработки.

В 70-80 годах прошлого века эта модель прочно укоренилась в разработке ПО в силу своей простоты и сходности с моделями разработки иных, не программных систем. В дальнейшем, в связи с

развитием программной инженерии и осознанием итеративного характера процесса разработки ПО эта модель активно критиковалась, практически, каждым автором соответствующих статей и учебников.

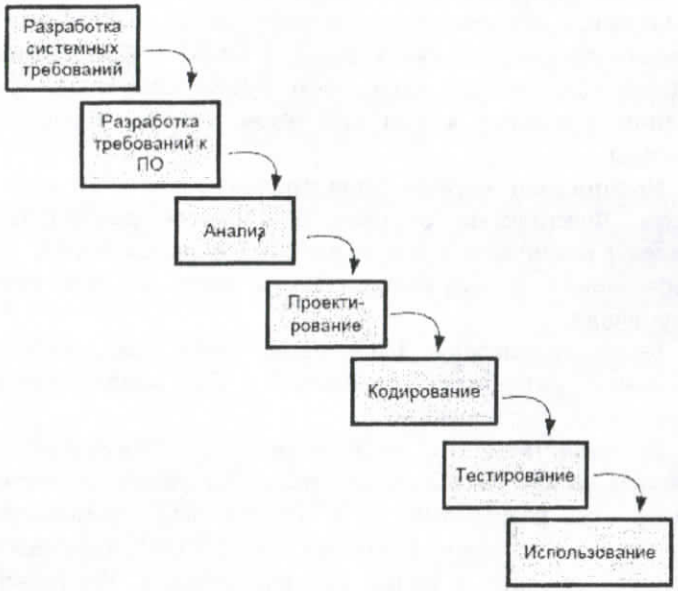

Рис. 2.1. Водопадная модель процесса

Стало общепринятым мнение, что она не отражает особенностей разработки ПО. Недостатками водопадной модели являются:

- отождествление фаз и видов деятельности, что влечет потерю гибкости разработки, в частности, трудности поддержки итеративного процесса разработки;
- требование полного окончания фазы-деятельности, закрепление результатов в виде подробного исходного документа (технического задания, проектной спецификации); однако опыт разработки ПО показывает, что невозможно полностью завершить разработку требований, дизайн системы и т.д. – все это подвержено изменениям; и причины тут не только в том, что подвижно окружение проекта, но и в том, что заранее не удается точно определить и сформулировать многие решения, они проясняются и уточняются лишь впоследствии;

- интеграция всех результатов разработки происходит в конце, вследствие чего интеграционные проблемы дают о себе знать слишком поздно;
- пользователи и заказчик не могут ознакомиться с вариантами системы во время разработки, и видят результат только в самом конце; тем самым, они не могут повлиять на процесс создания системы, и поэтому увеличиваются риски непонимания между разработчиками и пользователями/заказчиком;
- модель неустойчива к сбоям в финансировании проекта или перераспределению денежных средств, начатая разработка, фактически, не имеет альтернатив "по ходу дела".

Однако данная модель продолжает использоваться на практике – для небольших проектов или при разработке типовых систем, где итеративность не так востребована. С ее помощью удобно отслеживать разработку и осуществлять поэтапный контроль за проектом. Эта модель также часто используется в оффшорных проектах[1] с почасовой оплатой труда. Водопадная модель вошла в качестве составной части в другие модели и методологии, например, в MSF.

Спиральная модель была предложена Бэри Боемом в 1988 году для преодоления недостатков водопадной модели, прежде всего, для лучшего управления рисками. Согласно этой модели разработка продукта осуществляется по спирали, каждый виток которой является определенной фазой разработки. В отличие от водопадной модели в спиральной нет предопределенного и обязательного набора витков, каждый виток может стать последним при разработке системы, при его завершении составляются планы следующего витка. Наконец, виток является именно фазой, а не видом деятельности, как в водопадной модели, в его рамках может осуществляться много различных видов деятельности, то есть модель является двумерной.

Последовательность витков может быть такой: на первом витке принимается решение о целесообразности создания ПО, на следующем определяются системные требования, потом осуществляется проектирование системы и т.д. Витки могут иметь и иные значения.

Каждый виток имеет следующую структуру (секторы):

- определение целей, ограничений и альтернатив проекта;

- оценка альтернатив, оценка и разрешение рисков; возможно использование прототипирования (в том числе создание серии прототипов), симуляция системы, визуальное моделирование и анализ спецификаций; фокусировка на самых рисковых частях проекта;
- разработка и тестирование – здесь возможна водопадная модель или использование иных моделей и методов разработки ПО;
- планирование следующих итераций – анализируются результаты, планы и ресурсы на последующую разработку, принимается (или не принимается) решение о новом витке; анализируется, имеет ли смысл продолжать разрабатывать систему или нет; разработку можно и приостановить, например, из-за сбоев в финансировании; спиральная модель позволяет сделать это корректно.

Отдельная спираль может соответствовать разработке некоторой программной компоненты или внесению очередных изменений в продукт. Таким образом, у модели может появиться третье измерение.

Спиральную модель нецелесообразно применять в проектах с небольшой степенью риска, с ограниченным бюджетом, для небольших проектов. Кроме того, отсутствие хороших средств прототипирования может также сделать неудобным использование спиральной модели.

Спиральная модель не нашла широкого применения в индустрии и важна, скорее в историко-методологическом плане: она является первой итеративной моделью, имеет красивую метафору – спираль, – и, подобно водопадной модели, использовалась в дальнейшем при создании других моделей процесса и методологий разработки ПО.

Контрольные вопросы

1. Что такое универсальный процесс разработки ПО?
2. Что включает в себя стандартный процесс разработки ПО?
3. Дайте определение совершенствования процесса разработки ПО.
4. В чем заключаются Pull/Push стратегии?
5. В чем различие organization pull и technology push стратегий?
6. Чем характеризуются фазы деятельности?

7. Перечислить шаги разработки ПО.
8. В чем отличие водопадной и спиральной моделей разработки ПО?

Ключевые слова: *универсальный процесс, текущий процесс, конкретный процесс, стандартный процесс, совершенствование процесса, Pull/Push стратегии, Organization pull , Technology push, модель процесса, фазы деятельности, виды деятельности, водопадная модель, прототипирование, спиральная модель, виток спирали.*

Keywords: *universal process, the current process, the specific process, a standard process, process improvement, Pull / Push Strategy, Organization pull, - Technology push, the process model, the phases of activity, activities, waterfall model, prototyping, spiral model, turn of the helix.*

Kalit so'zlar: *universal jarayon, hozirgi jarayon, muayyan jarayon, standart jarayon, jarayonni takomillashtirish, Pull/Push strategiyalari, Organization pull , Technology push, jarayon modeli, faoliyat davri, faoliyat turlari, sharshara modeli, prototiplashtirish, spiral model, spiral o'rami.*

Упражнения

1. Перечислите по степени важности причины актуальности совершенствования процесса.
2. Выберите наиболее выгодную, по вашему мнению, Pull/Push стратегию. Обоснуйте.
3. Разделите процесс разработки ПО, на ваше усмотрение, на фазы.
4. Какая модель разработки выгодней по-вашему? Объясните.
5. Составьте спиральную модель разработки вашего ПО.

ГЛАВА 2. УПРАВЛЕНИЕ ПРОЕКТАМИ

В силу творческого характера программирования, существенной молодости участников разработки *ПО*, оказываются актуальными некоторые вопросы обычного промышленного производства, ставшие давно общим местом. Прежде всего, это дисциплина обязательств и *рабочий продукт*. Данные знания, будучи освоенными на практике, чрезвычайно полезны в командной работе. Кроме того, широко применяемые сейчас на практике *методологии разработки ПО*, поддержанные соответствующим программным *инструментарием*, *активно* используют эти понятия, уточняя и конкретизируя их.

2.1. Рабочий продукт

Одним из существенных условий для управляемости промышленного процесса является наличие отдельно оформленных результатов работы – как в окончательной поставке, так и промежуточных. Эти отдельные результаты в составе общих результатов *работ* помогают *идентифицировать*, планировать и оценивать различные части результата. Промежуточные результаты помогают *менеджерам* разных уровней отслеживать процесс воплощения проекта, заказчик получает возможность ознакомиться с результатами задолго до окончания проекта. Более того, сами участники проекта в своей ежедневной работе получают простой и эффективный способ обмена рабочей информацией – обмен результатами.

Таким результатом является **рабочий продукт** (*work product*) – любой *артефакт*, *произведенный* в процессе разработки *ПО*, например, *файл* или набор файлов, документы, составные части продукта, *сервисы*, процессы, спецификации, счета и т.д.

Ключевая разница между рабочим продуктом и компонентой *ПО* заключается в том, что первый необязательно материален и осязаем (*not* to be *engineered*), хотя может быть таковым. Нематериальный *рабочий продукт* – это, как правило, некоторый налаженный процесс – промышленный процесс производства какой-либо продукции, учебный процесс в университете (на факультете, на кафедре) и т.д.

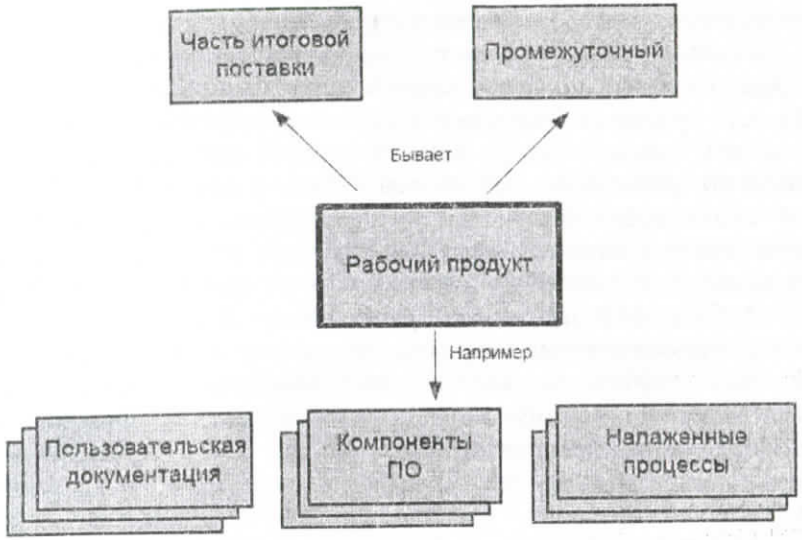

Рис. 2.1. Результаты рабочего продукта

Важно отметить, что *рабочий продукт* совсем не обязательно является составной частью итоговой поставки. Например, налаженный процесс тестирования системы не поставляется заказчику вместе с самой системой. Умение управлять проектами (не только в области программирования) во многом связано с искусством определять нужные рабочие продукты, настаивать на их создании и в их *терминах* вести приемку промежуточных этапов работы, организовывать синхронизацию различных *рабочих групп* отдельных специалистов.

Многие методологии включают в себя описание специфичных рабочих продуктов, используемых в процессе – *CMMI*, *MSF*, *RUP* и др. Например, в *MSF* это программный код, *диаграммы* приложений и классов (*application diagrams* и *class diagrams*), план *итераций*(*iteration plan*), *модульный* тест (*unit test*) и др. Для каждого из них точно описано содержание, ответственные за разработку,*место* в процессе и др. аспекты.

Остановимся чуть детальнее на промежуточных рабочих продуктах. Компонента *ПО*, созданная в проекте одним разработчиком и предоставленная для использования другому разработчику, оказывается рабочим продуктом. Ее надо минимально

протестировать, поправить имена интерфейсных классов и методов, быть может, убрать лишнее, не имеющее *отношение* к функциональности данной компоненты, разделить *public* и *private*, и т.д. То есть проделать некоторую дополнительную работу, которую, быть может, разработчик и не стал делать, если бы продолжал использовать компоненту только сам. Объем этих дополнительных *работ* существенно возрастает, если компонента должна быть представлена для использования в разработке, например, в другой центр разработки (например, иностранным партнерам, что является частой ситуацией в оффшорной разработке). Итак, изготовление хороших промежуточных рабочих продуктов очень важно для успешности проекта, но требует дополнительной работы от их *авторов*. Работать одному, не предоставляя рабочих продуктов – легче и для многих предпочтительнее. Но работа в команде требует накладных издержек, в том числе и в виде трат на создание промежуточных рабочих продуктов. Конечно, качество этих продуктов и трудозатраты на их изготовление сильно варьируются в зависимости от ситуации, но тут важно понимать сам принцип.

Рис. 2.2. Рабочий продукт

Итак, подытожим, что промежуточный *рабочий продукт* должен обязательно иметь ясную цель и конкретных пользователей, чтобы минимизировать накладные *расходы* на его создание.

2.2. Дисциплина обязательств

В основе *разделения обязанностей* в бизнесе и промышленном производстве, корпоративных правил и *норм* лежит определенная деловая этика, форма отношений – **дисциплина обязательств**. Она широко используется на практике и является одной из возможных форм социального взаимоотношения между людьми. Привнесение в бизнес и промышленность иных моделей человеческих отношений – семейных, сексуальных, дружеских и т.д. часто наносит делам серьезный урон, порождает конфликтность, понижает эффективность.

Основой этой формы отношений являются обязательства, которые:

- даются добровольно;
- не даются легко – работа, ресурсы, расписание должны быть тщательно учтены;
- между сторонами включает в себя то, *что* будет сделано, *кем* и в какие *сроки* ;
- открыто и публично сформулированы (то есть это не "тайное знание").

Кроме того:

- ответственная сторона стремится выполнить обязательства, даже если нужна помощь;
- *до* наступления *deadline*, как только становится очевидно, что работа не может быть закончена в срок, обсуждаются новые обязательства.

Отметим, что дисциплина обязательств не является каким-то сводом правил, законов, она отличается также от корпоративной культуры. Это – определенный групповой психический феномен, существующий в обществе современных людей. Приведенные выше пункты не являются исчерпывающим описанием этого феномена, но лишь проявляют и обозначают его, так сказать, вызывают нужные воспоминания.

Дисциплина обязательств, несмотря на очевидность, порой, не просто реализуется на практике, например, в творческих областях

человеческой *деятельности*, в области обучения и т.д. Существуют отдельные люди, которым эта дисциплина внутренне чужда вне зависимости от их рода *деятельности*.

С другой стороны, люди, освоившие эту дисциплину, часто стремятся применять ее в других областях жизни и человеческих отношений, что оказывается не всегда оправданным. Подчеркнем, что данная дисциплина является далеко не единственной моделью отношений между людьми. В качестве примера можно рассмотреть отношения в семье или дружбу, что, с очевидностью, не могут быть выражены дисциплиной обязательств. Так, вместо точности и пунктуальности в этих отношениях важно эмоционально-чувственное сопереживание, без которого они невозможны.

Дисциплине обязательств уделяется много внимания в рамках *MSF*, поскольку там в модели команды нет лидера, начальника. Эта дисциплина реализована также в Scrum: Scrum-*команда* имеет много свобод, и в силу этого – большую ответственность. Регламентируются также правила действий, когда обязательства не могут быть выполнены такой командой.

2.3. Проект и управление проектами

Классическое операционное разделение труда идет еще от Адама Смита и является сутью массового индустриального производства. То есть существует четко налаженный процесс работы и имеются области специализации – один цех точит, другой строгает, третий собирает, четвертый красит и т.д. *Пропускная способность* такого производства намного превосходит выполнение всей работы одним человеком или одной группой. Таким образом в XIX веке операционное разделение труда стало основой мануфактур, вытеснивших индивидуальное, ремесленное производство. В начале *XX* века эту структуру *работ* перенесли и на управление – то есть многочисленные менеджеры контролировали отдельные участки *работ*.

Однако высокий уровень сложности ряда задач в промышленности и бизнесе не позволяет (к счастью!) так работать везде. Существует много творческих, новых задач, где, быть может, в будущем и удастся создать конвейеры, но в данный момент для их решения требуется существенная концентрация сил и *энергии* людей,

неожиданные решения, а также удача и легкая рука. Это и есть область проектов.

Проект – это уникальная (в отличии от традиционной пооперационного промышленного производства) *деятельность*, имеющая начало и конец во времени, направленная на достижение определённого результата/цель, создание определённого, уникального продукта или услуги, при заданных ограничениях *по* ресурсам и срокам, а также требованиям к качеству и допустимому уровню риска.

В частности, *разработка программного обеспечения*, является, преимущественно, проектной областью.

Необходимо различать проекты промышленные и проекты творческие. У них разные принципы управления. Сложность промышленных проектов – в большом количестве разных организаций, компаний и относительной уникальности самих *работ*. Пример – строительство многоэтажного дома. Сюда же относятся различные международные проекты и не только промышленные – образовательные, культурные и пр. Задача в управлении такими проектами – это все охватить, все проконтролировать, ничего не забыть, все свести воедино, добиться движения, причем движения согласованного.

Творческие проекты характеризуются абсолютной новизной идеи – новый сервис, абсолютно новый *программный продукт*, какого еще не было на рынке, проекты в области искусства и науки. Любой начинающий бизнес, как правило, является таким вот творческим проектом. Причем новизна в подобных проектах не только абсолютная – такого еще не было. Такое, может, уже и было, но только не с нами, командой проекта. То есть присутствует огромный объем относительной новизны для самих людей, которые воплощают этот проект.

Проекты *по* разработке *программного обеспечения* находятся между двумя этими полюсами, занимая в этом пространстве различное положение. Часто они сложны потому, что объемны и находятся на стыке различных дисциплин – того целевого бизнеса, куда должен встроиться *программный продукт*, и сложного, нетривиального программирования. Часто сюда добавляется еще разработка уникального электронно-механического оборудования. С

другой стороны, поскольку *программирование активно* продвигается в разные сферы человеческой *деятельности*, то происходит это путем создания абсолютно новых, уникальных продуктов, и их разработка и продвижение обладают всеми чертами творческих проектов.

Управление проектами (*project management*) – область *деятельности*, в ходе которой, в рамках *определенных проектов*, определяются и достигаются четкие цели при нахождении компромисса между объемом *работ*, ресурсами (такими как время, деньги, труд, материалы, *энергия*, *пространство* и др.), временем, качеством и рисками.

Отметим несколько важных аспектов управления проектами.

- **Stakeholders** – это люди со стороны, которые не участвуют непосредственно в проекте, но влияют на него и/или заинтересованы в его результатах. Это могут быть будущие пользователи системы (например, в ситуации, когда они и заказчик – это не одно и то же), высшее руководство компании-разработчика и т.д. *Идентификация* всех *stakeholders* и грамотная работа с ними – важная составляющая успешного проектного менеджмента
- **Project scope** – это границы проекта. Это очень важное понятие для программных проектов в виду *изменчивости требований*. Часто бывает, что разработчики начинают создавать одну систему, а после, постепенно, она превращается в другую. Причем для менеджеров по продажам, а также заказчика, ничего радикально не произошло, а с точки зрения внутреннего устройства ПО, технологий, *алгоритмов* реализации, архитектуры – все радикально меняется. За подобными тенденциями должен следить и грамотно с ними разбираться проектный *менеджмент*.
- **Компромиссы** – важнейший аспект управления программными проектами в силу согласовываемости ПО. Важно не потерять все согласуемые параметры и стороны и найти приемлемый *компромисс*. Одна из техник управления *компромиссами* будет рассказана в контексте изучения методологии *MSF*.

2.4. Управление требованиями. Понятия о требованиях

Например, строители строят дома, пусть разные: многоэтажные, отдельные коттеджи, офисные здания и пр. – однако, весь этот спектр

вполне может охватить одна компания. Но все это дома. Строительной компании не приходится строить летающую тарелку, гиперболоид инженера Гарина, *луноход*, систему мгновенной телепортации и пр. А разработчики *ПО*, во многом, находятся именно в таком положении.

Велико разнообразие систем, которые создает одна компания, одна *команда*. Хотя сейчас и намечаются тенденции к специализации рынка разработки *ПО*, однако, причуды мировой экономики и многие другие причины приводят к тому, что строго специализированных компаний не так много, как хотелось бы. Многие области испытывают большой дефицит отдельных программистов и целых коллективов и компаний, хорошо разбирающихся в их специфике. Примером такой области может служить телевидение, где о данной проблеме открыто говорят на заседаниях различных международных сообществ.

Кроме того, *ПО* продолжает проникать во все новые и новые области человеческой *деятельности*, и сформулировать *адекватные* требования в этом случае вообще оказывается сверхтрудной задачей.

Но даже если речь идет об одной, определенной области, то *процент* новых, уникальных черт систем, принадлежащих этой области, высок: *по сочетанию* пользовательских характеристик, *по* особенностям среды исполнения и требованиям к *интеграции*, *по распределенности информации* о требованиях среди работников компании-заказчика. Все это несет на себе очень большой отпечаток индивидуальности заказчика – персональной или его компании, – сильно связано со спецификой его бизнеса, используемого в этой области оборудования.

Кроме того, существуют трудности в понимании между заказчиком и программистами, а еще – в изменчивости *ПО* (требования имеют тенденцию меняться в ходе разработки).

В итоге, далеко не очевидно, что та система, которую хочет заказчик, вообще может быть сделана. Трудно найти черную кошку в темной комнате, особенно если ее там нет. Или то, как поняли и воплотили задачу разработчики, окажется удобным, востребованным на рынке.

Ошибки и разночтения, которые возникают при выявлении требований к системе, оказываются одними из самых дорогих.

Требования – это то исходное понимание задачи разработчиками, которое является основой всей разработки.

Несколько слов о трудности взаимопонимания заказчика и разработчиков. Здесь сказывается большой разрыв между программистами и другими людьми. Во-первых, потому, что чтобы хорошо разобраться, какой должна быть система *автоматизации* больницы и система поддержки химических экспериментов – надо поработать в соответствующей области достаточное время. Или как-то иным способом научиться видеть проблемы данной *предметной области* изнутри. Во-вторых, сказывается специфичность программирования как сферы *деятельности*. Для большинства пользователей и заказчиков крайне не просто сформулировать точное *знание*, которое необходимо программистам. На вопрос, сколько типов анализов существует в вашей лаборатории, доктор, подумав, отвечает - 43. И уже потом, случайно, программист уточнил, а нет ли других типов? Конечно, есть, ответил доктор, только они случаются редко и могут быть в некотором смысле, какими угодно. В первый же раз он назвал лишь типовые. Но, конечно же, *информационная система* должна хранить информацию обо всех анализах, проведенных в лаборатории.

Теперь чуть подробнее об изменчивости *ПО* и ее причинах.

- Меняется ситуация на рынке, для которого предназначалась система или требования к системе ползут из-за быстро сменяющихся перспектив продажи еще неготовой системы.
- В ходе разработки возникают проблемы и трудности, в силу которых итоговая функциональность меняется (видоизменяется, урезается).
- Заказчик может менять свое собственное видение системы: то ли он лучше понимает, что же ему на самом деле надо, то ли выясняется, что он что-то упустил с самого начала, то ли выясняется, что разработчики его не так поняли. В общем, всякое бывает, важно лишь, что теперь заказчик определенно хочет иного.

Нечего и говорить, что *изменчивость требований по* ходу разработки очень болезненно сказывается на продукте. *Авторы* сталкивались, например, с такой ситуацией, что еще не созданную систему отдел продаж начинает *активно* продавать, в силу чего поступает огромный *поток* дополнительных требований. Все их

реализовать в полном объеме не удается, в итоге система оказывается набором демо-функциональности….

2.5. Виды и свойства требований

Разделим требования на две большие группы – функциональные и нефункциональные.

Функциональные требования являются детальным описанием поведения и *сервисов* системы, ее функционала. Они определяют то, что система должна уметь делать.

Нефункциональные требования не являются описанием функций системы. Этот вид требований описывает такие характеристики системы, как *надежность*, особенности поставки (наличие инсталлятора, документации), определенный уровень качества (например, для новой Java-машины это будет означать, что она удовлетворяет набору тестов, поддерживаемому компанией *Sun*). Сюда же могут относиться требования на средства и процесс разработки системы, требования к *переносимости*, соответствию стандартам и т.д. Требования этого вида часто относятся ко всей системе в целом. На практике, особенно начинающие специалисты, часто забывают про некоторые важные нефункциональные требования.

Сформулируем ряд важных свойств требований.

- *Ясность, недвусмысленность* — однозначность понимания *требований заказчиком* и разработчиками. Часто этого трудно достичь, поскольку конечная формализация требований, выполненная с точки зрения потребностей дальнейшей разработки, трудна для восприятия заказчиком или *специалистом предметной области*, которые должны проинспектировать правильность формализации.
- *Полнота и непротиворечивость.*
- *Необходимый уровень детализации.* Требования должны обладать ясно осознаваемым уровнем *детализации*, стилем описания, способом формализации: либо это описание свойств *предметной области*, для которой предназначается ПО, либо это *техническое задание*, которое прилагается к контракту, либо это проектная спецификация, которая должна быть уточнена в дальнейшем, при детальном проектировании. Либо это еще что-нибудь. Важно также ясно видеть и понимать тех, для кого данное описание требований

предназначено, иначе не избежать недопонимания и последующих за этим трудностей. Ведь в разработке ПО задействовано много различных специалистов – инженеров, программистов, *тестировщиков*, представителей заказчика, возможно, будущих пользователей – и все они имеют разное образование, профессиональные навыки и специализацию, часто говорят на разных языках. Здесь также важно, чтобы требования были максимально абстрактны и независимы от реализации.

- *Прослеживаемость* — важно видеть то или иное требование в различных моделях, документах, наконец, в коде системы. А то часто возникают вопросы типа – "Кто знает, почему мы решили, что такой-то модуль должен работать следующим образом ….?". Прослеживаемость функциональных требований достигается путем их дробления на отдельные, элементарные требования, присвоение им *идентификаторов* и создание трассировочной модели, которая в идеале должна протягиваться до программного кода. Хочется, например, знать, где нужно изменить код, если данное требование изменилось. На практике полная формальная прослеживаемость труднодостижима, поскольку логика и структура реализации системы могут сильно не совпадать с таковыми для модели требований. В итоге одно требование оказывается сильно "размазано" по коду, а тот или иной участок кода может влиять на много требований. Но стремиться к прослеживаемости необходимо, разумно совмещая формальные и неформальные подходы.
- *Тестируемость и проверяемость* — необходимо, чтобы существовали способы оттестировать и проверить данное требование. Причем, важны оба аспекта, поскольку часто проверить-то заказчик может, а вот тестировать данное требование очень трудно или невозможно в виду *ограниченности доступа* (например, по соображениям безопасности) к окружению системы для команды разработчика. Итак, необходимы процедуры проверки –выполнение тестов, проведение инспекций, проведение формальной *верификации* части требований и пр. Нужно также определять "планку" качества (чем выше качество, тем оно дороже стоит!), а также критерии полноты проверок, чтобы выполняющие их и *руководители проекта* четко осознавали, что именно проверено, а что еще нет.

- *Модифицируемость*. Определяет процедуры внесения изменений в требования.

2.6. Варианты формализации требований

Вообще говоря, требования как таковые – это некоторая *абстракция*. В реальной практике они всегда существуют в виде какого-то представления – документа, модели, *формальной спецификации*, списка и т.д. Требования важны как таковые, потому что оседают в виде понимания разработчиками нужд заказчика и будущих пользователей создаваемой системы. Но так как в программном проекте много различных аспектов, видов *деятельности* и фаз разработки, то это понимание может принимать очень разные представления. Каждое *представление требований* выполняет определенную задачу, например, служит "мостом", фиксацией соглашения между разными группами специалистов, или используется для оперативного *управления проектом* (отслеживается, в какой фазе реализации находится то или иное требование, кто за него отвечает и пр.), или используется для *верификации* и модельно-ориентированного тестирования. И в первом, и во втором, и в третьем примере мы имеем дело с требованиями, но формализованы они будут *по*-разному.

Итак, формализация требований в проекте может быть очень разной – это зависит от его величины, принятого процесса разработки, используемых инструментальных средств, а также тех задач, которые решают формализованные требования. Более того, может существовать параллельно несколько формализаций, решающих различные задачи. Рассмотрим варианты.

1. *Неформальная постановка требований в переписке по электронной почте*. Хорошо работает в небольших проектах, при вовлеченности заказчика в разработку (например, команда выполняет субподряд). Хорошо также при таком стиле, когда есть взаимопонимание между заказчиком и командой, то есть лишние формальности не требуются. Однако, электронные письма в такой ситуации часто оказываются важными документами – важно уметь вести деловую переписку, подводить итоги, хранить важные письма и пользоваться ими при разногласиях. Важно также вовремя понять,

когда такой способ перестает работать и необходимы более формальные подходы.

2. *Требования в виде документа* – описание *предметной области* и ее свойств, *техническое задание* как приложение к контракту, *функциональная спецификация* для разработчиков и т.д.

3. *Требования в виде графа с зависимостями* в одном из средств поддержки требований (*IBM* Rational RequisitePro, *DOORS*, Borland CaliberRM и нек. др.). Такое представление удобно при частом изменении требований, при отслеживании выполнения требований, при организации "привязки" к требованиям задач, людей, тестов, кода. Важно также, чтобы была возможность легко создавать такие графы из текстовых документов, и наоборот, создавать презентационные документы по таким графам.

4. *Формальная модель требований* для *верификации*, модельно-ориентированного тестирования и т.д.

Итак, каждый способ представления требований должен отвечать на следующие вопросы: кто потребитель, *пользователь* этого представления, как именно, с какой целью это *представление* используется.

2.7. Некоторые ошибки при документировании требований.

Перечислим ряд ошибок, встречающихся при составлении технических заданий и иных документов с требованиями.

- Описание возможных решений вместо требований.
- Нечеткие требования, которые не допускают однозначную проверку, оставляют недосказанности, имеют оттенок советов, обсуждений, рекомендаций: "Возможно, что имеет смысл реализовать также…..", "и т.д.".
- Игнорирование аудитории, для которой предназначено *представление требований*. Например, если спецификацию составляет инженер заказчика, то часто встречается переизбыток информации об оборудовании, с которым должна работать программная система, отсутствует *глоссарий* терминов и определений основных понятий, используются многочисленные синонимы и т.д. Или допущен слишком большой уклон в сторону программирования, что делает данную спецификацию непонятной всем непрограммистам.

- Пропуск важных аспектов, связанных с нефункциональными требованиями, в частности, информации об окружении системы, о сроках готовности других систем, с которыми должна взаимодействовать данная. Последнее случается, например, когда данная программная система является частью более крупного проекта. Типичны проблемы при создании программно-аппаратных систем, когда аппаратура не успевает вовремя и ПО невозможно тестировать, а в сроках и требованиях это не предусмотрено….

2.8. Цикл работы с требованиями

В своде знаний *по программной инженерии* SWEBOK определяются следующие виды *деятельности* при работе с требованиями.

- *Выделение требований (requirements elicitation)*, нацеленное на выявление всех возможных источников требований и ограничений на работу системы и *извлечение требований* из этих источников.
- *Анализ требований (requirements analysis)*, целью которого является обнаружение и устранение противоречий и неоднозначностей в требованиях, их уточнение и систематизация.
- *Описание требований (requirements specification)*. В результате этой *деятельности* требования должны быть оформлены в виде структурированного набора документов и моделей, который может *систематически* анализироваться, оцениваться с разных *позиций* и в итоге должен быть утвержден как официальная формулировка требований к системе.
- *Валидация требований (requirements validation)*, которая решает задачу оценки понятности сформулированных требований и их характеристик, необходимых, чтобы разрабатывать ПО на их основе, в первую очередь, непротиворечивости и полноты, а также соответствия корпоративным стандартам на техническую документацию.

Контрольные вопросы

1. Чем обусловлена изменчивость ПО?
2. Перечислить свойства требований.
3. Представление требований.

4. Перечислить ошибки при документировании требований.
5. Перечислить виды деятельности при работе с требованиями.

Ключевые слова: *изменчивость ПО, изменчивость требований, функциональные требования, нефункциональные требования, ясность, полнота и непротиворечивость, уровень детализации, техническое задание, прослеживаемость, тестируемость и проверяемость, модифицируемость, формальная спецификация, неформальная постановка требований, требование документ, требование-граф, формальная модель требований, документирование требований, выделение требований, анализ требований, описание требований, валидация требований*

Keywords: *variability, variability of requirements, functional requirements, non-functional requirements, clarity, completeness and consistency, level of detail, the terms of reference, traceability, testability and testability, modifiability, formal specification, informal setting requirements, requirement document requirement graph, formal model requirements, documentation requirements, allocation of requirements, requirements analysis, description of requirements, validation requirements.*

Kalit so'zlar: *DT o'zgaruvchanligi, talablar o'zgaruvchanligi, funksional talablar, funksional bo'lmagan talablar, oydinlik, to'liqlik, zidmaslik, detalizatsiya darajasi, tehnik topshiriq, tekshirib turish, test o'tkazish, turlanish, formal spetsifikatsiya, talablarning noformal qo'yilishi, talablarni dokumentatsiasi, talablar belgilanishi, talablar tahlili, talablar ta'rifi, talablar to'g'riligi*

Упражнения

1. Определите функциональные требования для произвольного ПО.
2. Определите нефункциональные требования для произвольного ПО.
3. Какой вариант формализации требований подходит лучше всех? Почему?
4. Проанализируйте требования в задании 1.
5. Выполните валидацию требований в задании 1.

ГЛАВА 3. СИСТЕМНОЕ МОДЕЛИРОВАНИЕ

3.1. Понятие о модели

Пользовательские требования обычно пишутся на естественном языке, поскольку они должны быть понятны даже не специалистам в области разработки ПО. Однако более детализированные системные требования должны описываться более "техническим" способом. Одной из широко используемых методик документирования системных требований является построение ряда моделей системы. Эти модели используют графические представления, показывающие решение как исходной задачи, для которой создается система, так и разрабатываемой системы. Как правило, графические представления более понятны, чем детальное описание системных требований на естественном языке. Модели являются связующим звеном между процессом анализа исходной задачи и процессом проектирования системы.

Модели можно использовать в процессе анализа существующей системы, которую нужно или заменить, или модифицировать, а также для формирования системных требований. Модели могут представить систему в различных аспектах.

1. Внешнее представление, когда моделируется окружение или рабочая среда системы.
2. Описание поведения системы, когда моделируется ее поведение.
3. Описание структуры системы, когда моделируется системная архитектура или структуры данных, обрабатываемых системой.

Эти три типа представления систем раскрыты в данной главе; кроме того, здесь рассматривается объектное моделирование, которое до некоторой степени объединяет поведенческое и структурное моделирование.

Такие структурные методы, как структурный анализ систем и объектно-ориентированный анализ, обеспечивают основу для детального моделирования системы как части процесса постановки и анализа требований. Большинство структурных методов работают с определенными типами системных моделей. Эти методы обычно определяют процесс, который используются для построения моделей,

и набор правил, которые применяются к этим моделям. Для поддержки структурных методов существуют различные CASE-средства (обсуждаемые в разделе 5.5), включающие редакторы моделей, автоматизированную систему документирования и инструменты проверки моделей.

Однако методы структурного анализа имеют ряд недостатков.

1. Они не обеспечивают эффективной поддержки формирования нефункциональных системных требований.
2. Обычно руководства по этим методам не содержат советов, которые помогали бы пользователям решить, подходит ли данный метод для решения конкретной задачи.
3. В результате применения этих методов часто получается объемная документация, при этом суть системных требований скрывается за массой несущественных деталей.
4. Построенные модели очень детализированы и трудны для понимания. Обычные пользователи не могут реально проверить действенность этих моделей.

Практическая разработка требований не должна ограничиваться только моделями, построенными с помощью тех или иных методов. Например, объектно-ориентированные методы обычно не предполагают применения для разработки моделей потоков данных. Однако, исходя из моего опыта, такие модели полезны для объектно-ориентированного анализа, поскольку отражают понимание системы конечным пользователем и могут помочь идентифицировать объекты и действия с ними.

Наиболее важным аспектом системного моделирования является то, что оно опускает детали. Модель является абстракцией системы и легче поддается анализу, чем любое другое представление этой системы. В идеале *представление* системы должно сохранять всю информацию относительно представляемого объекта. *Абстракция* является упрощением и определяется выбором наиболее важных характеристик системы.

Различные типы системных моделей основаны на разных подхода, к абстракции. Например, модель потоков данных концентрирует внимание на прохождении данных через систему и на функциональных преобразованиях этих данных. Модель оставляет без внимания структуру данных. И наоборот, модель "сущность-

связь" предполагает документирование системных данных и их взаимосвязь, не касаясь системных функций.

Приведем типы системных моделей, которые могут создаваться в процессе анализа систем.

1. *Модель обработки данных.* Диаграммы потоков данных показывают последовательность обработки данных в системе.
2. *Композиционная модель.* Диаграммы "сущность-связь" показывают, как системные сущности составляются из других сущностей.
3. *Архитектурная модель.* Эти модели показывают основные подсистемы, из которых строится система.
4. *Классификационная модель.* Диаграммы наследования классов показывают, какие объекты имеют общие характеристики.
5. *Модель "стимул-ответ".* Диаграммы изменения состояний показывают, как система реагирует на внутренние и внешние события.

Эти типы моделей описаны далее в главе. Везде, где возможно, используют обозначения унифицированного языка моделирования UML, который является стандартом языка моделирования, особенно для объектно-ориентированного моделирования. В тех случаях, когда UML не предусматривает подходящих нотаций, для описания моделей используют простые интуитивные обозначения.

3.2. Модели системного окружения

На ранних этапах формирования требований необходимо определить границы системы. Этот этап предполагает работу с лицами, участвующими в формировании требований (см. главу 2), для того чтобы разграничить систему и ее рабочее окружение. В некоторых случаях границы между системой и ее окружением относительно ясны. Например, когда новая система заменяет существующую, ее рабочее окружение обычно совпадает с окружением существующей системы.

В других случаях необходим дополнительный анализ. Например, рабочее окружение разрабатываемого набора CASE-средств может включать существующую базу данных, сервисы которой используются системой, но набор средств может также иметь внутреннюю базу данных. Если база данных уже существует,

определение границ между ними может оказаться сложной технической и управленческой проблемой. Только после проведения дополнительного анализа можно будет принять решение о том, что является, а что не является частью разрабатываемой системы.

На определение системного окружения могут также влиять социальные и организационные ограничения, т.е. граница системы может определяться не только техническими факторами. Например, система может быть очерчена так, что при ее разработке не будет необходимости консультироваться с менеджерами; при другом определении границ может возрасти ее стоимость или возникнет необходимость расширить отдел разработки и т.п.

После определения границ между системой и ее окружением далее специфицируется само рабочее окружение и связи между ним и системой. Обычно на этом этапе строится простая структурная модель, подобная представленной на рис. 3.1 модели структуры окружения информационной системы, управляющей сетью банкоматов. Структурные модели высокого уровня обычно являются простыми блок-схемами, где каждая подсистема представлена именованным прямоугольником, а линии показывают, что существуют некоторые связи между подсистемами.

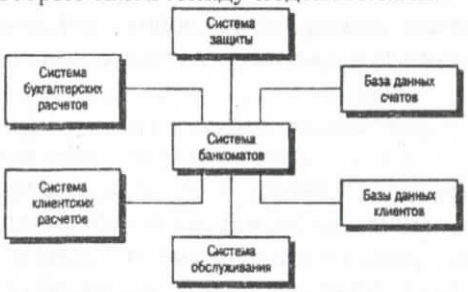

Рис. 3.1. Рабочее окружение системы управления банкоматами

На рис. 3.1 показано, что каждый банкомат присоединен к базе данных счетов, к локальной системе клиентских расчетов, к системе защиты и к системе обслуживания банкоматов. Система также соединена с базой данных клиентов, контролирующей сеть банкоматов, и с локальной бухгалтерской системой.

Структурные модели описывают непосредственное рабочее окружение системы. Но они не показывают связи между другими

системами в окружающей среде, которые не соединены непосредственно с разрабатываемой системой, но могут на нее влиять. Например, внешние системы могут производить данные для системы или использовать данные, произведенные системой. При этом они могут быть соединены между собой и системой через сеть или не соединены вообще. Они могут физически соприкасаться или располагаться в разных зданиях. Все эти взаимоотношения могут влиять на требования к разрабатываемой системе и должны быть приняты во внимание.

Таким образом, простые структурные модели обычно дополняются моделями других типов, например моделями процессов, которые показывают взаимодействия в системе, или моделями потоков данных (описаны в следующем разделе), которые показывают последовательность обработки и перемещения данных внутри системы и между другими системами в окружающей среде.

На рис. 3.2 представлена модель процесса заказа оборудования организацией. Она включает определение необходимого оборудования, поиск и выбор поставщиков, заказ, поставку и проверку оборудования после поставки. Для определения системы компьютерной поддержки этого процесса необходимо решить, какие из этих действий будут выполняться системой, а какие окажутся внешними по отношению к ней. На рис. 3.2 пунктирной линией ограничены действия, выполняемые внутри такой системы.

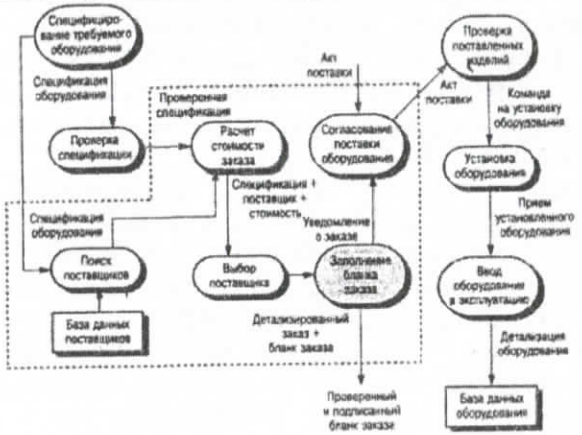

Рис. 3.2. Модель процесса приобретения оборудования

3.3. Поведенческие модели

Эти модели используются для описания общего поведения системы. Здесь рассматривается два типа поведенческих моделей – модель потоков данных, которая представляет обработку данных в системе, и модель конечного автомата, которая показывает реакцию системы на события. Эти модели можно использовать отдельно или совместно, в зависимости от типа разрабатываемой системы.

Большинство бизнес-систем прежде всего управляют данными. Они также управляют вводом данных в систему и сравнительно мало занимаются обработкой внешних событий. Для таких систем модель потоков данных может содержать все, что необходимо для описания поведения системы. В противоположность им системы реального времени управляют событиями с минимальной обработкой данных. Модель конечного автомата (обсуждаемая в разделе 3.2.2) является наиболее эффективным способом описания их поведение. Другие классы систем управляют как данными, так и событиями, поэтому для их представления необходимы оба типа моделей.

3.4. Модели потоков данных

Модели потока данных – это интуитивно понятный способ показа последовательности обработки данных внутри системы. Нотации, используемые в этих моделях, описывают обработку данных с помощью системных функций, а также хранение и перемещения данных между системными функциями. Модели потоков данных стали широко использоваться после публикации книги о структурном системном анализе. На базе этого фундаментального исследования было разработано множество методов анализа систем.

Модели потоков данных используются для показа последовательности шагов обработки данных. Эти шаги обработки или преобразования данных выполняются программными функциями. В сущности, диаграммы потоков данных используются для документирования программных функций перед проектированием системы. Анализ модели обработки данных может быть выполнен специалистами вручную или с помощью компьютера.

Модель потоков данных, показанная на рис. 3.3, представляет действия, выполняемые при оформлении заказа на оборудование. Это

описание части процесса размещения заказа на оборудования, представленного на рис. 3.2. Данная модель показывает процесс перемещения бланка заказа при его обработке.

Рис. 3.3. Диаграмма потоков данных при обработке бланка заказа

В диаграммах потоков данных используются следующие обозначения (см. рис. 3.3): закругленные прямоугольники соответствуют этапам обработки данных; стрелки, снабженные примечаниями с названием данных, представляют потоки данных; прямоугольники соответствуют хранилищам или источникам данных.

Модели потоков данных ценны тем, что они прослеживают и документируют перемещение данных по системе, помогая тем самым аналитикам понять этот процесс. Преимущество диаграмм потоков данных в том, что они, в отличие от других моделей, просты и интуитивно понятны. Поэтому их можно объяснить потенциальным пользователям системы, которые затем могут участвовать в ее анализе.

Модели потоков данных показывают функциональную структуру системы, где каждое преобразование данных соответствует одной системной функции. Иногда модели потоков данных используют для описания потоков данных в рабочем окружении системы. Такая модель показывает, как различные системы и подсистемы обмениваются информацией. Подсистемы окружения не обязаны быть простыми функциями. Например, одна подсистема может быть сервером базы данных с довольно сложным интерфейсом. На рис. 3.4 показана подобная диаграмма потоков

данных. В этом примере закругленные прямоугольники представляют подсистемы.

Рис. 3.4. Диаграмма потоков данных комплекса CASE-средств

3.5. Модели конечных автоматов

Модели конечных автоматов* используются для моделирования поведения системы, реагирующей на внутренние или внешние события. Такая модель показывает состояние системы и события, которые служат причиной перехода системы из одного состояния в другое. Модель не показывает поток данных внутри системы. Этот тип модели особенно полезен для моделирования систем реального времени, поскольку этими системами обычно управляют входные сигналы, приходящие из окружения системы. Модели конечных автоматов являются неотъемлемой частью методов проектирования систем реального времени. В работе определены *диаграммы состояний*, которые стали основой системы нотаций в языке моделирования UML.

Модель конечного автомата системы предполагает, что в любое время система находится в одном из возможных состояний. При получении входного сигнала или стимула система может изменить свое состояние. Например, система, управляющая клапаном, при получении команды оператора (стимул) может перейти из состояния "Клапан открыт" к состоянию "Клапан закрыт".

На рис. 3.5 показана модель конечного автомата (диаграмма состояний) простой микроволновой печи, оборудованной кнопками включения питания, таймера и запуска системы.

Реальная микроволновая печь на самом деле намного сложнее описанной здесь системы. Вместе с тем эта модель показывает все основные средства системы. Для упрощения модели предполагают такую последовательность действий при использовании печи.

1. Выбор уровня мощности (половинная или полная).

2. Ввод времени работы печи.

3. Нажатие кнопки запуска, после чего печь работает заданное время.

Для безопасности печь не должна действовать при открытой двери, по окончании работы должен прозвучать звуковой сигнал. Печь имеет простой дисплей, на котором отображаются различные предупреждения и сообщения.

Нотация UML, которую используют для описания модели конечного автомата и диаграммы состояний, разработана для моделирования поведения объектов. Но ее можно использовать для любого типа моделей конечного автомата. В этой нотации закругленные прямоугольники соответствуют состояниям системы. Они содержат краткое описание действий, выполняемых в этом состоянии. Помеченные стрелки представляют стимулы (или входные сигналы), которые приводят к переходу системы из одного состояния в другое.

На рис. 3.5 видно, что система первоначально реагирует на нажатие кнопок полной или половинной мощности. Пользователь может изменить свое решение после нажатия одной из этих кнопок и выбрать другую кнопку. Кнопка запуска сработает только после задания времени работы печи и закрытия двери. Нажатием кнопки запуска начинается работа печи в течение указанного времени.

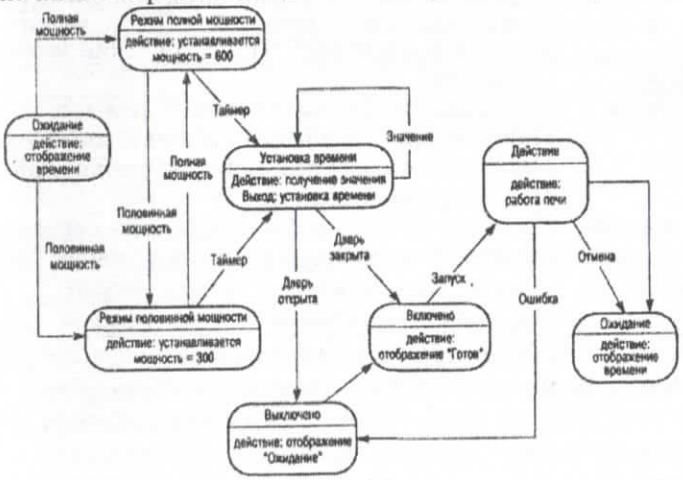

Рис. 3.5. Диаграмма состояний автомата микроволновой печи

Нотации UML позволяют определять действия, которые выполняются в том или ином состоянии. Но для создания системной спецификации необходима более детальная информация о стимулах и состояниях системы (табл. 3.1). Эта информация может храниться в словаре данных, как будет показано далее в этом разделе.

Таблица 3.1. Описание состояний и стимулов микроволновой печи

Состояние	Описание
Ожидание	Печь находится в состоянии ожидания входных данных. На дисплее высвечивается текущее время
Режим половинной мощности	Мощность печи устанавливается на 300 Вт. На дисплее отображается "Половинная мощность"
Режим полной мощности	Мощность печи устанавливается на 600 Вт. На дисплее отображается "Полная мощность"
Установка времени	Пользователем устанавливается время работы печи. Дисплей показывает заданное время
Выключено	В целях безопасности печь выключена. Внутренность печи освещена. Дисплей показывает "Не готов"
Включено	Питание печи включено. Внутри печи света нет.
Работа	Печь работает. Внутри печи включается свет. Дисплей отображает обратный отсчет таймера. По окончании работы звучит звуковой сигнал в течение 5 секунд. Свет включен. Пока звучит сигнал, дисплей высвечивает "Приготовление закончено"
Стимулы	**Описание**
Половинная	Пользователь нажимает кнопку режима половинной
Полная	Пользователь нажимает кнопку режима полной
Таймер	Пользователь нажимает одну из кнопок таймера
Число	Пользователь вводит число
Дверь открыта	Переключатель двери печи в состоянии "Не
Дверь закрыта	Переключатель двери печи в положении "Закрыто"
Запуск	Пользователь нажимает кнопку запуска
Отмена	Пользователь нажимает кнопку отмены

Основная проблема метода конечного автомата состоит в том, что число возможных состояний может быть очень велико. Поэтому для моделей больших систем необходима структуризация возможных состояний системы. Один из способов структуризации состоит в использовании суперсостояний, которые объединяют ряд отдельных

состояний. Такое суперсостояние подобно одному состоянию модели высокого уровня, которое детализируется на отдельной диаграмме. Для иллюстрации этого понятия рассмотрим состояние Работа модели микроволновой печи (см. рис. 3.6). Это суперсостояние более детально показано на рис. 3.6.

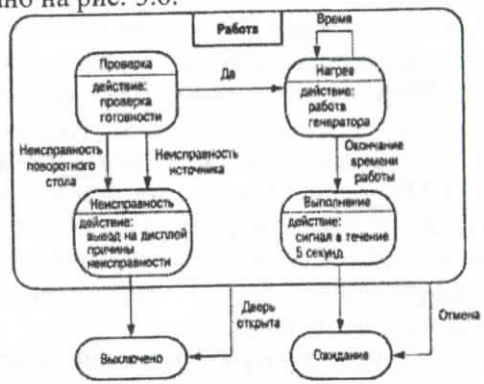

Рис. 3.6. Работа микроволновой печи

Состояние Работа включает ряд подсостояний. Диаграмма на рис. 3.6 показывает, что в состоянии Работа сначала проверяется готовность печи к работе и, если обнаружены какие-либо проблемы, включается аварийная сигнализация и печь выключается. В состоянии Нагрев работает микроволновой генератор в течение указанного времени, по завершении его работы автоматически подается звуковой сигнал. Если во время работы печи будет открыта дверь, система переходит в состояние Выключено, как показано на рис. 3.6.

3.6. Модели данных

Многие большие программные системы используют информационные базы данных. В одних случаях эта база данных существует независимо от программной системы, в других – специально создается для разрабатываемой системы. Важной частью моделирования систем является определение логической формы данных, обрабатываемых системой.

Наиболее широко используемой методологией моделирования данных является моделирование типа "сущность-связь-атрибут",

которое показывает структуру данных, их атрибуты и отношения между ними*. Этот метод моделирования был предложен в середине 1970-х годов Ченом (Chen); с тех пор разработано несколько вариантов этого метода.

Язык моделирования UML не имеет определенных обозначений для этого типа моделей данных, что желательно для объектно-ориентированного процесса разработки ПО, где для описания систем используются объекты и их отношения. Если сущностям поставить в соответствие простейшие классы объектов (без ассоциированных методов), тогда в качестве моделей данных можно использовать модели классов UML совместно с именованными ассоциациями между классами. Хотя такие модели данных не могут служить примером "хорошего" языка моделирования, удобство использования стандартных обозначений UML перевешивает возможные несовершенства таких конструкций.

Для описания структуры обрабатываемой информации модели данных часто используются совместно с моделями потоков данных. На рис. 3.7 представлена модель данных для системы проектирования ПО. Такую систему можно реализовать на основе комплекса инструментальных CASE-средств, показанного на рис. 3.4.

Проекты структуры ПО представляются ориентированными графами. Они состоят из набора узлов различных типов, соединенных дугами, отображающими связи между структурными узлами. В системе проектирования присутствуют средства вывода на дисплей этого графа (т.е. структурной диаграммы) и его преобразования к виду, удобному для хранения в базе данных проектов. Система редактирования выполняет преобразования структурной диаграммы из формата базы данных в формат, позволяющий отобразить ее на экране монитора в виде блок-схемы. Информация, предоставляемая редактором другим средствам анализа проекта, должна включить логическое представление графа проекта. Конечно, эти средства "не интересуются" деталями физического представления растрового изображения графа. Они работают с объектами, их логическими атрибутами и связями между ними.

На рис. 3.7 видно, что проект имеет атрибуты **имя, описание, дата создания и дата изменения**. Проект состоит из узлов и связей между ними (т.е. дуг). Узлы и связи имеют атрибуты **имя и тип**. Они

могут иметь набор меток, которые хранят другую описательную информацию. Каждая метка имеет атрибуты **имя, пиктограмма и текст**.

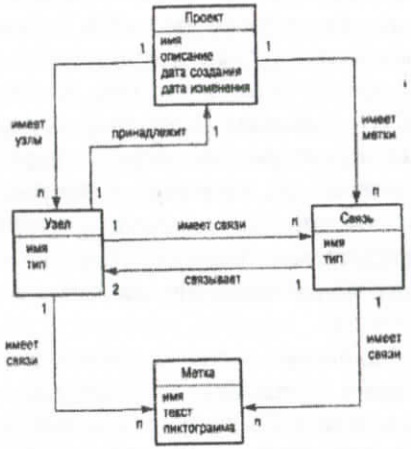

Рис. 3.7. Модель данных для системы проектирования ПО

Подобно всем графическим моделям, модели "сущность-связь-атрибут" недостаточно детализированы, поэтому они обычно дополняются более подробным описанием объектов, связей и атрибутов, включенных в модель. Эти описания собираются в словари данных или репозитории. Словари данных необходимы при разработке моделей системы и могут использоваться для управления информацией, содержащейся во всех моделях системы.

Упрощенно словарь данных – это просто алфавитный список имен, которые включены в различные модели системы. Вместе с именем словарь должен содержать описание именованного объекта, а если имя соответствует сложному объекту, может быть представлено описание построения этого объекта. Другая информация, например дата создания или фамилия разработчика, может приводиться в зависимости от типа разрабатываемой модели.

Перечислим преимущества использования словаря данных.

1. Существует механизм управления именами. При разработке большой системной модели, вероятно, придется изобретать имена для сущностей и связей. Эти имена должны быть

уникальными. Программа словаря данных может проверять уникальность имен и сообщать об их дублировании.

2. Словарь может служить хранилищем организационной информации, которая может связать анализ, проектирование, реализацию и модернизацию системы. Вся информация о системных объектах и сущностях находится в одном месте.

Все имена, используемые в системе (имена сущностей, типов, связей, атрибутов и системных сервисов), должны быть введены в словарь данных. Программные средства словаря должны обеспечить создание новых записей, их хранение и запросы к словарю. Такое программное обеспечение может быть интегрировано с другими программными средствами. Большинство CASE-средств, которые применяются для моделирования систем, могут поддерживать словари данных.

В примере словаря данных, приведенном в табл. 3.2, представлены имена, взятые из модели данных системы проектирования (см. рис. 3.7). Это упрощенный пример, где опущены некоторые имена и сокращена соответствующая информация.

Таблица 3.2. Пример словаря данных

Имя	Описание	Тип	Дата
имеет метки	Отношение 1:N между сущностями типа Узел или Связь и сущностями типа Метка*	Связь	5.10.1998
Метка	Содержит информацию об узлах и связях. Метки представляются пиктограммами и соответствующим текстом	Сущность	8.12.1998
Связь	Отношение 1:1 между сущностями, представленными узлами. Связи имеют тип и имя	Связь	8.12.1998
имя (метка)	Каждая метка имеет имя, которое должно быть уникальным	Атрибут	8.12.1998
имя (узел)	Каждый узел имеет имя, которое должно быть уникальным. Имя может содержать до 64 символов	Атрибут	5.11.1998

3.7. Объектные модели

Объектно-ориентированный подход широко используется при разработке программного обеспечения, особенно для разработки интерактивных систем. В этом случае системные требования формируются на основе объектной модели, а программирование выполняется с помощью объектно-ориентированных языков, таких, как Java или C++.

Объектные модели, разработанные для формирования требований, могут использоваться как для представления данных, так и для процессов их обработки. В этом отношении они объединяют модели потоков данных и семантические модели данных. Они также полезны для классификации системных сущностей и могут представлять сущности, состоящие из других сущностей.

Для некоторых классов систем объектные модели – естественный способ отображения реально существующих объектов, которые находятся под управлением системы. Например, для систем, обрабатывающих информацию относительно конкретных объектов (таких, как автомобили, самолеты, книги и т.д.), которые имеют четко определенные атрибуты. Более абстрактные высокоуровневые сущности, например библиотеки, медицинские регистрирующие системы или текстовые редакторы, труднее моделировать в виде классов объектов, поскольку они имеют достаточно сложный интерфейс, состоящий из независимых атрибутов и методов.

Объектные модели, разработанные во время анализа требований, несомненно, упрощают переход к объектно-ориентированному проектированию и программированию. Однако конечные пользователи часто считают объектные модели неестественными и трудными для понимания. Часто они предпочитают функциональные представления процессов обработки данных. Поэтому полезно дополнить их моделями потоков данных, чтобы показать сквозную обработку данных в системе.

Класс объектов – это абстракция множества объектов, которые определяются общими атрибутами (как в семантической модели данных) и сервисами или операциями, которые обеспечиваются каждым объектом. Объекты - это исполняемые сущности с атрибутами и сервисами класса объектов. Объекты представляют собой реализацию класса, на основе одного класса можно создать

много различных объектов. Обычно при разработке объектных моделей основное внимание сосредоточено на классах объектов и их отношениях.

Модели систем, разрабатываемые при формировании требований, должны отображать реальные сущности, принадлежащие классам объектов. Классы не должны содержать информацию об отдельных системных объектах. Можно разработать различные типы объектных моделей, показывающие, как классы связаны друг с другом, как объекты агрегируются из других объектов, как объекты взаимодействуют с другими объектами и т.д. Эти модели расширяют понимание разрабатываемой системы.

Идентификация объектов и классов объектов считается наиболее сложной задачей в процессе объектно-ориентированной разработки систем. Определение объектов – это основа для анализа и проектирования системы.

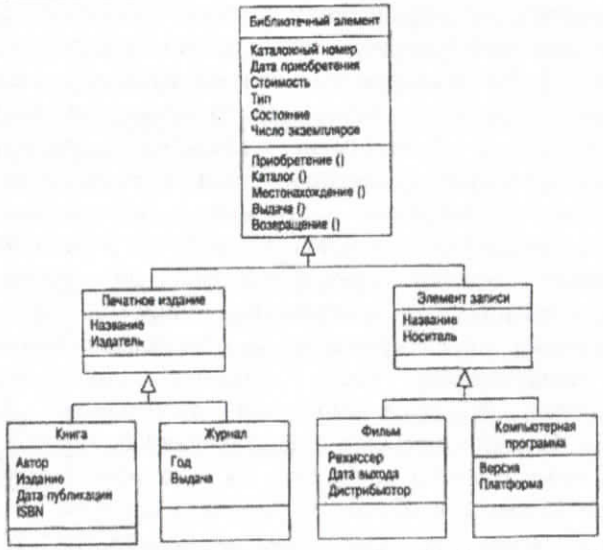

Рис. 3.8. Часть иерархии классов для библиотечной системы

Различные методы объектно-ориентированного анализа были предложены в 1990-х годах Бучем (Booch), Кодом и Джордоном (Goad and Yourdon), а также Рамбо (Rumbaugh). Эти методы имели много общего, поэтому три главных разработчика – Буч, Рамбо и

Якобсон (Jacobson) – решили интегрировать их для разработки унифицированного метода. В результате разработанный ими унифицированный язык моделирования (Unified Modeling Language – UML) стал фактическим стандартом для моделирования объектов. UML предлагает нотации для различных типов системных моделей. Вы уже видели модели вариантов использования и диаграммы последовательностей в главе 2, а также диаграммы состояний ранее в этой главе.

В UML класс объектов представлен вертикально ориентированным прямоугольником с тремя секциями (рис. 3.8).

1. В верхней секции располагается имя класса.
2. Атрибуты класса находятся в средней секции.
3. Операции, связанные с классом, приводятся в нижней секции прямоугольника.

Поскольку полностью описать UML в данной книге нет возможности, здесь на объектных моделях будет показано, как классифицируются объекты, как наследуются атрибуты и операции от других объектов, а также будут приведены модели агрегирования и простые поведенческие модели.

3.8. Инструментальные CASE-средства

Это пакет программных средств, который поддерживает отдельные этапы процесса разработки программного обеспечения: проектирование, написание программного кода или тестирование. Преимущество группирования CASE-средств в инструментальный пакет заключается в том, что, работая вместе, они обеспечивают более всестороннюю поддержку процесса разработки ПО, чем могут предложить отдельные инструментальные средства. Общие сервисы могут вызываться всеми средствами. Инструментальные средства можно объединить в пакет с помощью общих файлов, репозитория или общей структуры данных.

Инструментальные средства анализа и проектирования ПО созданы для поддержки моделирования систем на этапах анализа и проектирования процесса разработки программного обеспечения. Они поддерживают создание, редактирование и анализ графических нотаций, используемых в структурных методах. Инструментальные средства анализа и проектирования часто поддерживают только

определенные методы проектирования и анализа, например объектно-ориентированные. Другие инструментальные средства являются общими системами редактирования диаграмм многих типов, которые используются разными методами проектирования и анализа. Инструментальные средства, ориентированные на определенные методы, обычно автоматически поддерживают правила и базовые принципы этих методов, что позволяет выполнять автоматический контроль диаграмм.

На рис. 3.13 показана схема пакета инструментальных средств поддержки анализа и проектирования ПО. Инструментальные средства обычно объединяются через общий репозиторий, структура которого является собственностью разработчика пакета инструментальных средств. Пакеты инструментальных средств обычно закрыты, т.е. не рассчитаны на добавление пользователями собственных инструментов или на изменение средств пакета.

Рис. 3.13. Пакет инструментальных средств для анализа и проектирования ПО

Ниже перечислены средства, которые входят в пакет инструментальных средств, показанный на рис. 3.13.

1. *Редакторы диаграмм* предназначены для создания диаграмм потоков данных, иерархий объектов, диаграмм "сущность-связь" и т.д. Эти редакторы не только имеют средства рисования, но и поддерживают различные типы объектов, используемые в диаграммах.

2. *Средства проектирования, анализа и проверки* выполняют проектирование ПО и создают отчет об ошибках и дефектах в системной архитектуре. Они могут работать совместно с системой

редактирования, поэтому обнаруженные ошибки можно устранить на ранней стадии процесса проектирования.

3. *Центральный репозиторий* позволяет проектировщику найти нужный проект и соответствующую проектную информацию.

4. *Словарь* данных хранит информацию об объектах, которые используются в структуре системы.

5. *Средства генерирования отчетов* на основе информации из центрального репозитория автоматически генерируют системную документацию.

6. *Средства создания форм* определяют форматы документов и экранных форм.

7. *Средства импортирования и экспортирования* позволяют обмениваться информацией из центрального репозитория различным инструментальным средствам.

8. *Генераторы программного кода* автоматически генерируют программы на основе проектов, хранящихся в центральном репозитории.

В некоторых случаях возможно генерировать программы или фрагменты программ на основе информации, представленной в системной модели. Генераторы кода, которые включены в пакеты инструментальных средств, могут генерировать код на таких языках, как Java, C++ или C. Поскольку в моделях не предусмотрена детализация низкого уровня, генератор программного кода не в состоянии сгенерировать законченную систему. Обычно необходимы программисты для завершения автоматически сгенерированных программ.

Некоторые пакеты инструментальных средств анализа и проектирования предназначены для поддержки методов разработки программных приложений деловой сферы. Обычно для создания общего репозитория инструментов они используют системы баз данных типа Sybase или Oracle. Эти пакеты инструментальных средств содержат большое количество средств языков программирования четвертого поколения, предназначенных для генерирования программного кода на основе системной архитектуры, они также могут генерировать базы данных с использованием языков программирования четвертого поколения.

Контрольные вопросы

1. Аспекты представления моделей.
2. Что такое абстракция?
3. Перечислите типы системных моделей
4. Место системного окружения в системе разработки ПО, определение границ.
5. Что такое модели потоков данных?
6. Для чего используют модели конечных автоматов?

Ключевые слова: *модель системы, внешнее представление, поведение системы, структура системы, структурный анализ, абстракция, модель обработки данных, композиционная модель, архитектурная модель, классификационная модель, модель "стимул-ответ", границы системы, системное окружение, поведение системы, модели потоков данных.*

Keywords: system model, external presentation, behaviour of the system, structure of the system, structured analysis, abstraction, model data processing, compositional model, architectral model, taxonomic model, model "stimulus-answer", borders of the system, system encirclement, behaviour of the system, models dataflow.

Kalit so'zlar: tizim modeli, tashqi tasavvur, tizim hulqi, tizim strukturasi, strukturaviy tahlil, abstraktlash, ma'lumotlar bilan ishlash modeli, kompozitsion model, arhitekturaviy model, klassifikatsion model, "stiml-javob" modeli, tizim chegaralari, tizim muhiti, ma'lumotlar oqimi modeli.

Упражнения

1. Разработайте модель рабочего окружения для информационной системы больницы. Модель должна предусматривать ввод данных о новых пациентах и систему хранения рентгеновских снимков.
2. Создайте модель обработки данных в системе электронной почты. Необходимо отдельно смоделировать отправку почты и ее получение.
3. Нарисуйте модель конечного автомата управляющей системы:
   - для автоматической стиральной машины, которая имеет различные программы для разных типов белья;
   - для программного обеспечения проигрывателя компакт-дисков;

   - для телефонного автоответчика, который регистрирует входные сообщения и показывает число принятых сообщений на дисплее. Система должна соединять владельца телефона с абонентом после ввода им последовательности чисел (телефонного номера абонента), а также, имея записанные сообщения, повторять их по телефону.
4. Разработайте модель классов объектов для системы электронной почты. Если вы выполнили упражнение 4.3, опишите различия и сходства между моделью обработки данных и объектной моделью.
5. Используя подход "сущность-связь", опишите возможную модель данных для системы библиотечного каталога.
6. Разработайте объектную модель, включающую диаграммы иерархии классов и агрегирования, и показывающую основные элементы системы персонального компьютера и его программного обеспечения.
7. Разработайте диаграмму последовательностей, которая показывает действия студента, регистрирующегося на определенный курс в университете. Курс может иметь ограниченное число мест, поэтому процесс регистрации должен проверять количество доступных мест. Предположите, что студент обращается к электронному каталогу курсов, чтобы выяснить количество доступных мест.
8. Опишите три действия, выполняемых при моделировании систем, которые могут быть поддержаны пакетом инструментальных CASE-средств при выполнении некоторых методов анализа систем. Опишите три действия, которые невозможно легко автоматизировать.

ГЛАВА 4. ПРОТОТИПИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

4.1. Прототип программного обеспечения

Заказчикам программного обеспечения и конечным пользователям обычно сложно четко сформулировать требования к разрабатываемой программной системе. Трудно предвидеть, как система будет влиять на трудовой процесс, как она будет взаимодействовать с другими системами и какие операции, выполняемые пользователями, необходимо автоматизировать. Тщательный анализ требований помогает уменьшить неопределенность относительно того, что система должна делать. Однако реально проверить требования, прежде чем их утвердить, практически невозможно. В этой ситуации может помочь прототип системы.

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

Прототип ПО помогает на двух этапах процесса разработки системных требований.

1. *Постановка требований.* Пользователи могут экспериментировать с системными прототипами, что позволяет им проверять, как будет работать система. Пользователи получают новые идеи для постановки требований, могут определить сильные и слабые стороны ПО. В результате могут сформироваться новые требования.

2. *Проверка требований.* Прототип позволяет обнаружить ошибки и упущения в ранее принятых требованиях. Например, системные функции, определенные в требованиях, могут быть полезными и нужными (с точки зрения пользователя). Однако в процессе применения этих функций совместно с другими функциями пользователи могут изменить первоначальное мнение о них. В

результате требования к системе изменятся, отражая измененное понимание пользователями системных функций.

Прототипирование можно использовать при анализе рисков и на начальном этапе разработки планов управления программным проектом (см. главу 3). Основной опасностью при разработке ПО являются ошибки и упущения в требованиях. Затраты на устранение ошибок в требованиях на более поздних стадиях процесса разработки могут быть очень высокими. Эксперименты показывают, что прототипирование уменьшает число проблем, связанных с разработкой требований. Кроме того, прототипирование уменьшает общую стоимость разработки системы. По этим причинам оно часто используется в процессе разработки требований.

Однако различие между прототипированием, как отдельным этапом процесса разработки ПО, и разработкой основной программной системы неочевидно. В настоящее время многие системы разрабатываются с использованием эволюционного подхода, когда быстро создается первоначальная версия системы, которая затем постепенно изменяется до ее окончательного варианта. При этом часто используются методы быстрой разработки приложений, которые также можно использовать при создании прототипов. Эти вопросы обсуждаются в разделе 4.2.

Наряду с тем что прототипы помогают формировать требования, они имеют и другие достоинства.

1. Различное толкование требований разработчиками ПО и пользователями можно выявить при демонстрации действующего прототипа системы.
2. В процессе создания прототипа разработчики могут выявить неполные или несогласованные требования.
3. Работая, хотя и ограниченно, в виде прототипа, система может продемонстрировать свои слабые и сильные стороны.
4. Прототип может служить основой для написания спецификации высококачественной системы. Разработка прототипа обычно ведет к улучшению спецификации системы.

Действующий прототип может также использоваться для других целей.

1. *Обучение пользователя.* Прототип системы можно использовать для обучения персонала перед поставкой окончательного варианта системы.

2. *Тестирование системы.* Прототипы позволяют "прокручивать" тесты. Один и тот же тест запускается на прототипе и на системе. Если получаются одинаковые результаты, это означает, что тест не обнаружил дефектов в системе. Если результаты отличаются, то необходимо исследовать причины различия, что позволяет выявить возможные ошибки в системе.

На основе изучения 39 различных программных проектов, использовавших прототипирование, в работе сделан вывод, что эффективность применения прототипов при разработке ПО состоит в следующем.

1. Улучшаются эксплуатационные качества системы.
2. Система больше соответствует потребностям пользователей.
3. Системная архитектура становится более совершенной.
4. Сопровождение системы упрощается и становится более удобным.
5. Сокращаются расходы на разработку системы.

Эти исследования показывают, что улучшение эксплуатационных качеств системы и увеличение соответствия системы потребностям пользователя не требуют увеличения общей стоимости разработки системы. Прототипирование обычно повышает стоимость начальных этапов разработки ПО, но снижает затраты на более поздних этапах.

Рис. 4.1. Процесс разработки прототипа

Модель процесса разработки прототипа показана на рис.4.1. На первом этапе данного процесса определяются назначение прототипа и цель прототипирования. Целью может быть разработка макета пользовательского интерфейса, проверка функциональных системных требований или демонстрация реализуемости системы для руководства. Один и тот же прототип не может служить

одновременно всем целям. Если цели определены неточно, функции прототипа могут быть восприняты неверно.

На следующем этапе процесса разработки прототипа определяются его функциональные возможности, т.е. принимается решение о том, какие свойства системы должен отражать прототип, а какие (что, возможно, более важно) – нет. Для уменьшения затрат на создание прототипа можно исключить некоторые системные функции. Например, можно ослабить временные характеристики и требования к использованию памяти. Средства управления и обработки ошибок могут игнорироваться либо быть элементарными, если, конечно, целью прототипирования не является модель интерфейса пользователя. Также могут быть снижены требования к надежности и качеству программ.

Заключительный этап процесса прототипирования – оценивание созданного прототипа. В работе утверждается, что это наиболее важный этап процесса прототипирования. Здесь проверяется, насколько созданный прототип соответствует своему назначению и целям, а также на его основе создается план мероприятий по совершенствованию разрабатываемой системы.

4.2. Прототипирование в процессе разработки ПО

Как уже отмечалось, конечным пользователям трудно представить, как они будут использовать новую систему ПО в повседневной работе. Если система большая и сложная, то это невозможно сделать, прежде чем система будет создана и введена в эксплуатацию.

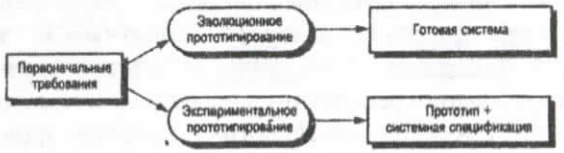

Рис.4.2. Эволюционное и экспериментальное прототипирование

Один из способов преодоления этой трудности состоит в использовании эволюционного метода разработки систем. Это означает, что пользователю предоставляется незавершенная система, которая затем изменяется и дополняется до тех пор, пока не станут ясны все требования пользователя. В качестве альтернативы можно

построить "экспериментальный" прототип, который поможет проанализировать и проверить требования. После этого создается система. На рис. 4.2 показаны оба подхода к использованию прототипов.

Эволюционное прототипирование начинается с построения относительно простой системы, которая реализует наиболее важные требования пользователя. По мере выявления новых требований прототип изменяется и дополняется. В конечном счете он становится той системой, которая требуется. В этом процессе не используется детальная системная спецификация, во многих случаях нет даже формального документа с системными требованиями. В настоящее время эволюционное прототипирование является обычной технологией разработки программных систем, которая широко используется при разработке Web-узлов и приложений электронной коммерции.

В противоположность эволюционному подходу метод экспериментального прототипирования предназначен для разработки и уточнения системной спецификации. Прототип создается, оценивается и модифицируется. Данные оценивания прототипа используются для дальнейшей детализации спецификации. Когда системные требования сформированы, прототип больше не нужен.

Существует различие между целями эволюционного и экспериментального прототипирования.

Целью эволюционного прототипирования является поставка работающей системы конечному пользователю. Это означает, что необходимо начать создание системы, реализующей требования пользователя, которые наиболее понятны и которые имеют наивысший приоритет. Требования с более низким приоритетом и нечеткие требования реализуются по запросам пользователей.

Целью экспериментального прототипирования является проверка и формирование системных требований. Здесь сначала создается прототип, реализующий те требования, которые сформулированы нечетко и с которыми необходимо "разобраться". Требования, которые сформулированы четко и понятно, не нуждаются в прототипировании.

Другое важное различие между этими подходами касается управления качеством разрабатываемой системы.

Экспериментальные прототипы имеют очень короткий срок жизни. Они быстро меняются и для них высокая эксплуатационная надежность не требуется. Для экспериментального прототипа допускается пониженная эффективность и безотказность, поскольку прототип должен выполнить только свою основную функцию – помочь в понимании требований.

В противоположность этому прототипы, которые эволюционируют в законченную систему, должны быть разработаны с такими же стандартами качества, что и любое другое программное обеспечение. Они должны иметь устойчивую структуру и высокую эксплуатационную надежность. Они должны быть безотказны, эффективны и отвечать соответствующим стандартам.

4.3. Эволюционное прототипирование

В основе эволюционного прототипирования лежит идея разработки первоначальной версии системы, демонстрации ее пользователям и последующей модификации вплоть до получения системы, отвечающей всем требованиям (рис. 4.3). Такой подход сначала использовался для разработки систем, которые трудно или невозможно специфицировать (например, систем искусственного интеллекта). В настоящее время он становится основной методикой при разработке программных систем. Эволюционное прототипирование имеет много общего с методами быстрой разработки приложений и часто входит в эти методы как их составная часть.

Рис. 4.3. Эволюционное прототипирование

Этот метод прототипирования имеет два основных преимущества.

1. *Ускорение разработки системы.* Как указывалось во введении, современные темпы изменений в деловой сфере требуют быстрых изменений программного обеспечения. В некоторых случаях быстрая поставка ПО, удобство и простота его использования более важны, чем полный спектр функциональных возможностей системы или долгосрочные возможности ее сопровождения.

2. *Взаимодействие пользователя с системой.* Участие пользователей в процессе разработки означает, что в системе более полно будут учтены пользовательские требования.

Между отдельными методами быстрой разработки ПО существуют различия, но все они имеют некоторые общие свойства.

1. Этапы разработки технических требований, проектирования и реализации перемежаются. Не существует детальной системной спецификации, проектная документация обычно зависит от инструментальных средств, используемых для реализации системы. Пользовательские требования определяют только наиболее важные характеристики системы.

2. Система разрабатывается пошагово. Конечные пользователи и другие лица, формирующие требования, участвуют на каждом шаге проектирования и оценивания новой версии системы. Они могут предлагать изменения и новые требования, которые будут реализованы в следующей версии системы.

3. Применение методов быстрой разработки систем (см. раздел 4.2). Они могут использовать инструментальные CASE-средства и языки четвертого поколения.

4. Пользовательский интерфейс системы обычно создается с использованием интерактивных систем разработки (см. раздел 4.3), которые позволяют быстро спроектировать и создать интерфейс.

Эволюционное прототипирование и методы, основанные на использовании детальной системной спецификации, отличаются подходами к верификации и аттестации систем. Верификация – процесс проверки системы на соответствие спецификации. Поскольку для прототипа не создается подробной спецификации, его верификация невозможна.

Аттестация системы должна показать, что программа соответствует тем целям, для которых она создавалась. Аттестацию также трудно провести без детальной спецификации, поскольку нет

четких формулировок целей. Конечные пользователи, участвующие в процессе разработки, могут быть удовлетворены системой, в то время как другие пользователи – не удовлетворены, поскольку система не полностью соответствует тем целям, которые они неявно перед ней поставили.

Верификацию и аттестацию системы, разработанной с использованием эволюционного прототипирования, можно осуществить, если она в достаточной степени соответствует поставленной цели и своему назначению. Это соответствие, конечно, нельзя измерить, можно сделать лишь субъективные оценки. Такой подход, как будет показано ниже, может породить проблемы, если программная система создается сторонними организациями-разработчиками.

Существует три основные проблемы эволюционного прототипирования, которые необходимо учитывать, особенно при разработке больших систем с длительным сроком жизненного цикла.

1. *Проблемы управления.* Структура управления разработкой программных систем строится в соответствии с утвержденной моделью процесса создания ПО, где для оценивания очередного этапа разработки используются специальные контрольные проектные элементы. Прототипы эволюционируют настолько быстро, что создавать контрольные элементы становится нерентабельно. Кроме того, быстрая разработка прототипа может потребовать применения новых технологий. В этом случае может возникнуть необходимость привлечения специалистов с более высокой квалификацией.

2. *Проблемы сопровождения системы.* Из-за непрерывных изменений в прототипах изменяется также структура системы. Это означает, что система будет трудна для понимания всем, кроме первоначальных разработчиков. Кроме того, может устареть специальная технология быстрой разработки, которая использовалась при создании прототипов. Поэтому могут возникнуть трудности при поиске людей, которые имеют знания, необходимые для сопровождения системы.

3. *Проблемы заключения контрактов.* Обычно контракт на разработку систем между заказчиком и разработчиками ПО основывается на системной спецификации. При отсутствии таковой трудно составить контракт на разработку системы. Для заказчика может быть невыгоден

контракт, по которому приходится просто платить разработчикам за время, потраченное на разработку проекта; также маловероятно, что разработчики согласятся на контракт с фиксированной ценой, поскольку они не могут предвидеть все прототипы, которые потребуется создать в процессе разработки системы.

Из этих проблем вытекает, что заказчики должны понимать, насколько эффективно эволюционное прототипирование в качестве метода разработки ПО. Этот метод позволяет быстро создавать системы малого и среднего размера, при этом стоимость разработки снижается, а качество повышается. Если к процессу разработки привлекаются конечные пользователи, то, вероятно, система будет соответствовать их реальным потребностям. Однако организации-разработчики, использующие этот метод, должны учитывать, что жизненный цикл таких систем будет относительно короток. При возрастании проблем с сопровождением систему необходимо заменить или полностью переписать. Для больших систем, когда к разработке привлекаются субподрядчики, на первый план выходят проблемы управления эволюционным прототипированием. В этом случае лучше применять экспериментальное прототипирование.

Пошаговая разработка (рис. 4.4) позволяет избежать некоторых проблем, характерных для эволюционного прототипирования. Общая архитектура системы, определенная на раннем этапе ее разработки, выступает в роли системного каркаса. Компоненты системы разрабатываются пошагово, затем включаются в этот каркас. Если компоненты аттестованы и включены в каркас, ни архитектура, ни компоненты уже не меняются, за исключением случая, когда обнаруживаются ошибки.

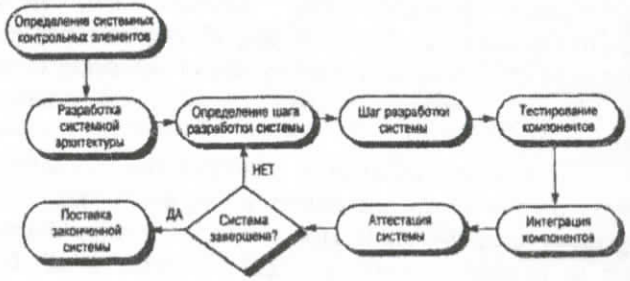

Рис.4.4. Пошаговый процесс разработки

Процесс пошаговой разработки более управляем, чем эволюционное прототипирование, поскольку следует обычным стандартам разработки ПО. Здесь планы и документация создаются для каждого шага разработки системы, что уменьшает количество ошибок. Как только системные компоненты интегрированы в каркас, их интерфейсы больше не изменяются.

4.4. Экспериментальное прототипирование

Модель процесса разработки ПО, основанная на экспериментальном прототипировании, показана на рис. 4.5. В этой модели расширен этап анализа требований в целях уменьшения общих затрат на разработку. Основное назначение прототипа – сделать понятными требования и предоставить дополнительную информацию для оценки рисков. После этого прототип больше не используется и не участвует в дальнейшем процессе разработки системы.

Рис.4.5. Разработка ПО с использованием экспериментальных прототипов

Этот метод прототипирования обычно используется для разработки аппаратных систем. Прежде чем будет начата дорогостоящая разработка системы, создается макет (прототип), который используется для проверки структуры системы. Электронный макет системы создается с использованием готовых компонентов, что позволяет разработать версию системы до того, как будут вложены денежные средства в разработку специализированных интегральных схем.

Экспериментальный прототип программных систем обычно не используется для проверки архитектуры системы, он помогает разработать системные требования. Прототип часто совершенно не похож на конечную систему. Система разрабатывается по

возможности быстро, поэтому для ускорения формирования требований используется упрощенный прототип системы. В экспериментальный прототип закладываются только обязательные системные функции, стандарты качества для прототипа могут быть снижены, критерии эффективности игнорируются. Язык программирования прототипа часто отличается от языка программирования, на котором будет создаваться окончательный вариант системы.

В модели процесса разработки ПО, показанной на рис. 4.5, предполагается, что прототип разрабатывается исходя из обобщенных системных требований, далее над прототипом проводятся эксперименты и он изменяется до тех пор, пока его функциональные возможности не удовлетворят заказчика. После этого на основе прототипа детализируются системные требования, реализуется обычная для организации-разработчика технология разработки ПО и система доводится до окончательной версии. Некоторые компоненты прототипа могут использоваться в системе, поэтому стоимость разработки может быть снижена.

В описываемой модели разработки ПО основная проблема состоит в том, что экспериментальный прототип может не соответствовать конечной системе, поставляемой заказчику. Специалист, тестирующий прототип, может иметь собственные интересы к системе, не типичные для ее пользователей. Время тестирования прототипа может быть недостаточным для его полного оценивания. Если прототип работает медленно, эксперты могут внести в него изменения, которые ускоряют работу, но не будут совпадать со средствами ускорения работы конечной системы.

Разработчики иногда подвергаются давлению менеджеров для ускорения работы над прототипом, особенно если намечается задержка в поставке окончательной версии системы. Обычно такое "ускорение" порождает ряд проблем.

1. Невозможно быстро настроить прототип для выполнения таких нефункциональных требований, как производительность, защищенность, устойчивость к сбоям и безотказность, которые игнорировались во время разработки прототипа.

2. Частые изменение во время разработки неизбежно приводят к тому, что прототип плохо документирован. Для

разработки прототипа используется только спецификация системной архитектуры. Этого недостаточно для долговременного сопровождения системы.

3. Изменения, сделанные во время разработки прототипа могут нарушить архитектуру системы. Ее обслуживание будет сложным и дорогостоящим.

4. В процессе разработки прототипа ослабляются стандарты качества.

Чтобы быть полезными в процессе разработки требований, экспериментальные прототипы не обязательно должны выполнять роль реальных макетов систем. Бумажные формы, имитирующие пользовательские интерфейсы систем, показали свою эффективность при формировании требований пользователя, в уточнении проекта интерфейса и при создании сценариев работы конечного пользователя. Они очень дешевы в разработке и могут быть созданы за несколько дней. Расширением этой методики является макет пользовательского интерфейса "Wizard of Oz" (Волшебник страны Оз). Пользователи взаимодействуют с этим интерфейсом, но их запросы направлены к специалисту, который интерпретирует их и имитирует соответствующую реакцию. Подобные подходы к прототипированию рассмотрены в работе.

4.5. Технологии быстрого прототипирования

Эти технологии рассчитаны главным образом на обеспечение быстрой разработки прототипов, а не на такие их системные характеристики, как производительность, удобство эксплуатации или безотказность. Существует три основных метода быстрой разработки прототипов.

1. Разработка с применением динамических языков высокого уровня.

2. Использование языков программирования баз данных.

3. Сборка приложений с повторным использованием компонентов.

Для удобства эти методы описаны в отдельных разделах. Но на практике они часто совместно используются при разработке прототипов систем. Например, язык программирования баз данных может применяться для извлечения данных с их последующей

обработкой с помощью повторно используемых компонентов. Интерфейс пользователя системы можно разработать, используя визуальное программирование. В статье описано смешанное применение этих методов при создании прототипа управляющей системы.

В настоящее время разработка прототипов обычно опирается на набор инструментов, поддерживающих по крайней мере два из этих методов. Например, система Smalltalk Visual Works поддерживает язык очень высокого уровня и обеспечивает повторное использование компонентов. Пакет Lotus Notes включает поддержку программирования баз данных с помощью языка высокого уровня и повторное использование компонентов, которые могут обеспечить операции над базой данных.

Большинство систем прототипирования сегодня поддерживают визуальное программирование, при котором некоторые части или весь прототип разрабатываются в интерактивном режиме. Вместо последовательного написания программ разработчик прототипа предпочитает работать с графическими пиктограммами, представляющими функции, данные или компоненты интерфейса пользователя, и соответствующими сценариями управления этими пиктограммами. Программа, готовая к исполнению, генерируется автоматически из визуального представления системы. Это упрощает разработку программы и уменьшает затраты на прототипирование.

4.6. Применение динамических языков высокого уровня

Динамические языки высокого уровня – это языки программирования, которые имеют мощные средства контроля данных во время выполнения программы. Они упрощают разработку программ, так как уменьшают число проблем, связанных с распределением памяти и управлением ею. Такие языки имеют средства, которые обычно должны быть построены из более примитивных конструкций в языках, подобных Ada или C. Примеры языков очень высокого уровня – Lisp (основанный на структурах списков), Prolog (основанный на алгебре логики) и Smalltalk (основанный на объектах).

До недавнего времени динамические языки высокого уровня широко не использовались для разработки больших систем,

поскольку они нуждаются в основательных средствах динамической поддержки. Эти средства увеличивали объем необходимой памяти и уменьшали скорость выполнения программ, написанных на этих языках. Однако возрастание мощности и снижение стоимости компьютерного оборудования сделало эти факторы не столь существенными!

Таким образом, для многих деловых приложений эти языки могут заменить такие традиционные языки программирования, как C, COBOL и Ada. Язык Java, несомненно, является основным языком разработки, имеющим корни в языке C++, но с включением многих средств языка Smalltalk наподобие платформенной независимости и автоматического управления памятью. Язык Java объединяет в себе многие преимущества языков высокого уровня, совмещая это с точностью и возможностью оптимизации выполнения, обычно предлагаемой языками третьего поколения. В языке Java много компонентов, доступных для повторного использования, все это делает его подходящим для эволюционного прототипирования.

В табл.4.1 представлены динамические языки, которые более всего используются при разработке прототипов. При выборе языка для написания прототипа необходимо ответить на ряд вопросов.

1. *Каков тип разрабатываемого приложения?* Как показано в табл. 6.1, для каждого типа приложения можно применить несколько различных языков. Если необходим прототип приложения, которое обрабатывает данных на естественном языке, то языки: Lisp или Prolog более подходят, чем Java или Smalltalk.

2. *Каков тип взаимодействия с пользователем?* Различные языки обеспечивают разные типы взаимодействия с пользователем. Некоторые языки, такие как Smalltalk и Java, хорошо интегрируются с Web-браузерами, в то время как язык Prolog лучше всего подходит для разработки текстовых интерфейсов.

3. *Какую рабочую среду обеспечивает язык?* Развитая рабочая среда поддержки языка со своими инструментальными средствами и легким доступом к повторно используемым компонентам упрощает процесс разработки прототипа.

Динамические языки высокого уровня для создания прототипа можно использовать совместно, когда различные части прототипа программируются на разных языках. В работе описывается

разработка прототипа телефонной сетевой системы, где были использованы четыре различных языка: Prolog для макетирования баз данных, Awk для составления счетов, CSP для спецификации протоколов и PAISLey для имитирования работы системы.

Таблица 4.1. Языки высокого уровня, используемые при прототипировании

Язык	Тип языка	Тип приложения
Smalltalk	Объектно-ориентированный	Интерактивные системы
Java	Объектно-ориентированный	Интерактивные системы
Prolog	Логический	Системы обработки символьной информации
Lisp	Основанный на списках	Системы обработки символьной информации

Не существует идеального языка для прототипирования больших систем, поскольку обычно различные части системы разнотипны. Преимущество многоязычного подхода в том, что для создания каждого компонента можно подобрать наиболее подходящий язык и таким образом ускорить разработку прототипа. Недостаток такого подхода в том, что трудно разработать коммуникационные связи для компонентов, написанных на разнородных языках.

Контрольные вопросы

1. Что такое прототип?
2. Для чего применяют прототип?
3. На каких этапах процесса разработки системных требований помогает прототип ПО?
4. Перечислите преимущества прототипов?
5. Какие различия между целями эволюционного и экспериментального прототипирования?
6. В чем преимущества и недостатки эволюционного прототипирования?

7. В чем преимущества и недостатки экспериментального прототипирования?
8. Какие методы существуют для быстрой разработки прототипов?
9. В чем особенность применения языков четвертого поколения?

Ключевые слова: *прототипирование, постановка требований, проверка требований, проблемы управления, проблемы сопровождения системы, проблемы заключения контрактов, язык четвертого поколения, генераторы интерфейсов.*

Keywords: prototyping, production requirements, inspection requirements, management problems, issues tracking system, contracting problems, fourth-generation languages, generators interfaces.

Kalit so'zlar: prototiplashtirish, talablar o'rnatilishi, talablarni tekshirish, boshqaruv muammolarim tizim kuzatuvining muammolari, shartnomalar tuzishdagi muammolar, to'rtinchi avlod tili, interfeys generatorlari.

Упражнения

1. Исследуйте возможность прототипирования в процессе разработки программного обеспечения в вашей организации. Напишите отчет для вашего менеджера, показывая классы проектов, где должно использоваться прототипирование, и рассчитайте ожидаемые затраты и выгоды от использования прототипирования.
2. Объясните, почему для разработки больших систем рекомендуется экспериментальное прототипирование.
3. Какие особенности языков, подобных Smalltalk и Lisp, способствуют поддержке быстрого прототипирования?
4. В каких обстоятельствах вы рекомендовали бы прототипирование как средство обоснования системных требований?
5. Опишите трудности, которые могут возникнуть при прототипировании встроенных компьютерных систем реального времени.

6. Спроектируйте программную систему преобразования требований в формальную спецификацию. Прокомментируйте преимущества и недостатки следующих стратегий разработки такой системы.
   - Разработайте экспериментальный прототип с помощью языка, подобного Smalltalk. Оцените этот прототип, затем сделайте обзор требований. Разработайте конечную систему, используя язык C.
   - Разработайте систему согласно существующим требованиям, используя язык Java, и затем модифицируйте ее, чтобы адаптировать к изменениям требований пользователя.
   - Разработайте систему, используя эволюционное прототипирование, с помощью языка типа Smalltalk. Измените прототип в соответствии с новыми пользовательскими запросами.
7. Обсудите прототипирование на основе повторного использования компонентов и опишите проблемы, которые могут при этом возникнуть. Как наиболее эффективно определить пригодные для повторного использования компоненты?
8. Каковы преимущества и недостатки использования механизма OLE для быстрой разработки приложений?
9. Благотворительная организация попросила вас создать макет системы, которая следила бы за всеми получаемыми ими пожертвованиями. Эта система должна сохранять имена и адреса жертвующих, их интересы, пожертвованную сумму и дату пожертвования. Если пожертвование достигает определенной суммы, жертвующий может добавить условия к пожертвованию (например, пожертвование должно быть израсходовано на определенный проект), система должна следить за такими пожертвованиями и за тем, как они были израсходованы. Обсудите, как использовать прототип системы, имея в виду, что системой будут пользоваться как постоянные работники благотворительной организации, так и добровольцы. Многие из добровольцев – пенсионеры, которые имеют малый опыт работы с компьютером или вовсе не имеют такого опыта.
10. Вы разработали экспериментальный прототип системы для заказчика, который его полностью удовлетворил. Заказчик утверждает, что нет необходимости разрабатывать конечную

систему, а вы можете поставить прототип, и предлагает за это хорошую цену. Вы знаете, что в будущем могут быть проблемы с сопровождением системы. Обсудите, что вы ответите этому заказчику.

ГЛАВА 5. АРХИТЕКТУРНОЕ ПРОЕКТИРОВАНИЕ

Большие системы всегда можно разбить на подсистемы, предоставляющие связанные наборы сервисов. *Архитектурным проектированием* называют первый этап процесса проектирования, на котором определяются подсистемы, а также структура управления и взаимодействия подсистем. Целью архитектурного проектирования является описание *архитектуры программного обеспечения.*

В идеале в спецификации требований не должно быть информации о структуре системы. В действительности же это справедливо только для небольших систем. Архитектурная декомпозиция системы необходима для структуризации и организации системной спецификации. Хорошим примером тому может служить изображенная на рис. 1.3 система управления воздушными полетами. Модель системной архитектуры часто является отправной точкой для создания спецификации различных частей системы. В процессе архитектурного проектирования разрабатывается базовая структура системы, т.е. определяются основные компоненты системы и взаимодействия между ними.

Существуют различные подходы к процессу архитектурного проектирования, которые зависят от профессионального опыта, а также мастерства и интуиции разработчиков. И все же можно выделить несколько этапов, общих для всех процессов архитектурного проектирования.

1. *Структурирование системы.* Программная система структурируется в виде совокупности относительно независимых подсистем. Также определяются взаимодействия между подсистемами. Этот этап рассматривается в разделе 5.1.
2. *Моделирование управления.* Разрабатывается базовая модель управления взаимоотношениями между частями системы. Этот этап рассматривается в разделе 5.2.
3. *Модульная декомпозиция.* Каждая определенная на первом этапе подсистема разбивается на отдельные модули. Здесь определяются типы модулей и типы их взаимосвязей. Этот этап рассматривается в разделе 5.3.

Как правило, эти этапы перемежаются и накладываются друг на друга. Этапы повторяются для все более детальной проработки

архитектуры до тех пор, пока архитектурный проект не будет удовлетворять системным требованиям.

Четких различий между подсистемами и модулями нет, но, думаю, будут полезными следующие определения.

1. *Подсистема* – это система (т.е. удовлетворяет "классическому" определению "система"), операции (методы) которой не зависят от сервисов, предоставляемых другими подсистемами. Подсистемы состоят из модулей и имеют определенные интерфейсы, с помощью которых взаимодействуют с другими подсистемами.
2. *Модуль* – это обычно компонент системы, который предоставляет один или несколько сервисов для других модулей. Модуль может использовать сервисы, поддерживаемые другими модулями. Как правило, модуль никогда не рассматривается как независимая система. Модули обычно состоят из ряда других, более простых компонентов.

Результатом процесса архитектурного проектирования является документ, отображающий архитектуру системы. Он состоит из набора графических схем представлений моделей системы с соответствующим описанием. В описании должно быть указано, из каких подсистем состоит система и из каких модулей слагается каждая подсистема. Графические схемы моделей системы позволяют взглянуть на архитектуру с разных сторон. Как правило, разрабатывается четыре архитектурные модели.

1. Статическая структурная модель, в которой представлены подсистемы или компоненты, разрабатываемые в дальнейшем независимо.
2. Динамическая модель процессов, в которой представлена организация процессов во время работы системы.
3. Интерфейсная модель, которая определяет сервисы, предоставляемые каждой подсистемой через общий интерфейс.
4. Модели отношений, в которых показаны взаимоотношения между частями системы, например поток данных между подсистемами.

Ряд исследователей при описании архитектуры систем предлагают использовать специальные языки описания архитектур. В книге рассматриваются основные свойства этих языков. В них

основными архитектурными элементами являются компоненты и коннекторы (объединяющие звенья); эти языки также предлагают принципы и правила построения архитектур. Однако, как и другие специализированные языки, они имеют один недостаток, а именно: все они понятны только освоившим их специалистам и почти не используются на практике. Фактически использование языков описания архитектур только усложняет анализ систем. Поэтому считают, что для описания архитектур лучше использовать неформальные модели и системы нотации, подобные предлагаемой, например унифицированный язык моделирования UML.

Архитектура системы может строиться в соответствии с определенной архитектурной моделью. Очень важно знать эти модели, их недостатки, преимущества и возможности применения. В этой главе рассматриваются структурные модели, модели управления и декомпозиции.

Вместе с тем архитектуру больших систем невозможно описать с помощью какой-либо одной модели. При разработке отдельных частей больших систем можно использовать разные архитектурные модели. Но в этом случае архитектура системы может оказаться слишком сложной, поскольку будет построена на комбинации различных архитектурных моделей. Разработчик должен подобрать наиболее подходящую модель, затем модифицировать ее соответственно требованиям разрабатываемого ПО. В разделе 5.4 рассматривается пример архитектуры компилятора, базирующейся на комбинации модели репозитория и модели потоков данных.

Архитектура системы влияет на производительность, надежность, удобство сопровождения и другие характеристики системы. Поэтому модели архитектуры, выбранные для данной системы, могут зависеть от нефункциональных системных требований.

1. *Производительность.* Если критическим требованием является производительность системы, следует разработать такую архитектуру, чтобы за все критические операции отвечало как можно меньше подсистем с максимально малым взаимодействием между ними. Чтобы уменьшить взаимодействие между компонентами, лучше использовать крупно модульные компоненты, а не мелкие структурные элементы.

2. *Защищенность.* В этом случае архитектура должна иметь многоуровневую структуру, в которой наиболее критические системные элементы защищены на внутренних уровнях, а проверка безопасности этих уровней осуществляется на более высоком уровне.

3. *Безопасность.* В этом случае архитектуру следует спроектировать так, чтобы за все операции, влияющие на безопасность системы, отвечало как можно меньше подсистем. Такой подход позволяет снизить стоимость разработки и решает проблему проверки надежности.

4. *Надежность.* В этом случае следует разработать архитектуру с включением избыточных компонентов, чтобы можно было заменять и обновлять их, не прерывая работу системы. Архитектуры отказоустойчивых систем с высокой работоспособностью рассматриваются в главе 18.

5. *Удобство сопровождения.* В этом случае архитектуру системы следует проектировать на уровне мелких структурных компонентов, которые можно легко изменять. Программы, создающие данные, должны быть отделены от программ, использующих эти данные. Следует также избегать структуры совместного использования данных.

Очевидно, что некоторые из перечисленных архитектур противоречат друг другу. Например, для того чтобы повысить производительность, необходимо использовать крупно модульные компоненты, в то же время сопровождение системы намного упрощается, если она состоит из мелких структурных компонентов. Если необходимо учесть оба требования, следует искать компромиссное решение. Ранее уже было сказано, что один из способов решения подобных проблем состоит в применении различных архитектурных моделей для разных частей системы.

5.1. Структурирование системы

На первом этапе процесса проектирования архитектуры система разбивается на несколько взаимодействующих подсистем. На самом абстрактном уровне архитектуру системы можно изобразить графически с помощью блок-схемы, в которой отдельные подсистемы представлены отдельными блоками. Если подсистему также можно разбить на несколько частей, на диаграмме эти части

изображаются прямоугольниками внутри больших блоков. Потоки данных и/или потоки управления между подсистемами обозначается стрелками. Такая блок-схема дает общее представление о структуре системы.

На рис. 5.1 представлена структурная модель архитектуры для системы управления автоматической упаковкой различных типов объектов. Она состоит из нескольких частей. Подсистема наблюдения изучает объекты на конвейере, определяет тип объекта и выбирает для него соответствующий тип упаковки. Затем объекты снимаются с конвейера, упаковываются и помещаются на другой конвейер. Примеры других архитектур приведены на рис. 1.2 и 1.3.

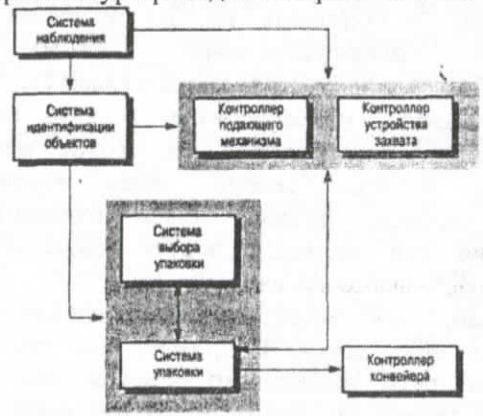

Рис. 5.1. Блок-схема системы управления автоматической упаковкой

Бэсс (Bass) считает, что подобные блок-схемы являются бесполезными представлениями системной архитектуры, поскольку из них нельзя ничего узнать ни о природе взаимоотношений между компонентами системы, ни об их свойствах. С точки зрения разработчика программного обеспечения, это абсолютно верно. Однако такие модели оказываются эффективными на этапе предварительного проектирования системы. Эта модель не перегружена деталями, с ее помощью удобно представить структуру системы. В структурной модели определены все основные подсистемы, которые можно разрабатывать независимо от остальных подсистем, следовательно, руководитель проекта может распределить разработку этих подсистем между различными исполнителями.

Конечно, для представления архитектуры используются не только блок-схемы, однако подобное представление системы не менее полезно, чем другие архитектурные модели.

Конечно, можно разрабатывать более детализированные модели структуры, в которых было бы показано, как именно подсистемы разделяют данные и как взаимодействуют друг с другом. В этом разделе рассматриваются три стандартные модели, а именно: модель репозитория, модель клиент/сервер и модель абстрактной машины.

5.2. Модель репозитория

Для того чтобы подсистемы, составляющие систему, работали эффективнее, между ними должен идти обмен информацией. Обмен можно организовать двумя способами.

1. Все совместно используемые данные хранятся в центральной базе данных, доступной всем подсистемам. Модель системы, основанная на совместном использовании базы данных, часто называют *моделью репозитория.*
2. Каждая подсистема имеет собственную базу данных. Взаимообмен данными между подсистемами происходит посредством передачи сообщений.

Большинство систем, обрабатывающих большие объемы данных, организованы вокруг совместно используемой базы данных, или репозитория. Поэтому *такая* модель подойдет к приложениям, в которых данные создаются в одной подсистеме, а используются в другой. Примерами могут служить системы управления информацией, системы автоматического проектирования и CASE-средства.

На рис. 5.2 представлен пример архитектуры интегрированного набора CASE-инструментов, основанный на совместно используемом репозитории. Считается, что для CASE-средств первый совместно используемый репозитории был разработан в начале 1970-х годов английской компанией ICL в процессе создания своей операционной системы. Широкую известность эта модель получила после того, как была применена для поддержки разработки систем, написанных на языке Ada. С тех пор многие CASE-средства разрабатываются с использованием общего репозитория.

Рис. 5.2. Архитектура интегрированного набора CASE-средств

Совместно используемые репозитории имеют как преимущества, так и недостатки.

1. Очевидно, что совместное использование больших объемов данных эффективно, поскольку не требуется передавать данные из одной подсистемы в другие.

2. С другой стороны, подсистемы должны быть согласованы с моделью репозитория данных. Это всегда приводит к необходимости компромисса между требованиями, предъявляемыми к каждой подсистеме. Компромиссное решение может понизить их производительность. Если форматы данных новых подсистем не подходят под согласованную модель представления данных, интегрировать такие подсистемы сложно или невозможно.

3. Подсистемам, в которых создаются данные, не нужно знать, как эти данные используются в других подсистемах.

4. Поскольку в соответствии с согласованной моделью данных генерируются большие объемы информации, модернизация таких систем проблематична. Перевод системы на новую модель данных будет дорогостоящим и сложным, а порой даже невозможным.

5. В системах с репозиторием такие средства, как резервное копирование, обеспечение безопасности, управление доступом и восстановление данных, централизованы, поскольку входят в систему управления репозиторием. Эти средства выполняют только свои основные операции и не занимаются другими вопросами.

6. С другой стороны, к разным подсистемам предъявляются разные требования, касающиеся безопасности, восстановления и резервирования данных. В модели репозитория ко всем подсистемам применяется одинаковая политика.

7. Модель совместного использования репозитория прозрачна: если новые подсистемы совместимы с согласованной моделью данных, их можно непосредственно интегрировать в систему.

8. Однако сложно разместить репозитории на нескольких машинах, поскольку могут возникнуть проблемы, связанные с избыточностью и нарушением целостности данных.

В рассматриваемой модели репозитории является пассивным элементом, а управление им возложено на подсистемы, использующие данные из репозитория. Для систем искусственного интеллекта разработан альтернативный подход. Он основан на модели "рабочей области", которая инициирует подсистемы тогда, когда конкретные данные становятся доступными. Такой подход применим к системам, в которых форма данных хорошо структурирована. Эта модель обсуждается в работе.

5.3. Модель клиент/сервер

Модель архитектуры клиент/сервер – это модель распределенной системы, в которой показано распределение данных и процессов между несколькими процессорами. Модель включает три основных компонента.

1. Набор автономных серверов, предоставляющих сервисы другим подсистемам. Например, сервер печати, который предоставляет услуги печати, файловые серверы, предоставляющие сервисы управления файлами, и сервер-компилятор, который предлагает сервисы по компилированию исходных кодов программ.

2. Набор клиентов, которые вызывают сервисы, предоставляемые серверами. В контексте системы клиенты являются обычными подсистемами. Допускается параллельное выполнение нескольких экземпляров клиентской программы.

3. Сеть, посредством которой клиенты получают доступ к сервисам. В принципе нет никакого запрета на то, чтобы клиенты и серверы запускались на одной машине. На практике, однако, модель клиент/сервер в такой ситуации не используется.

Клиенты должны знать имена доступных серверов и сервисов, которые они предоставляют. В то же время серверам не нужно знать ни имена клиентов, ни их количество. Клиенты получают доступ к

сервисам, предоставляемым сервером, посредством удаленного вызова процедур.

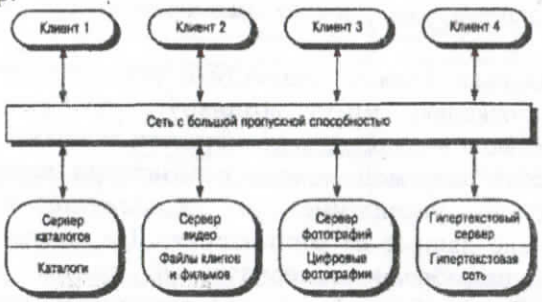

Рис. 5.3. Архитектура библиотечной системы фильмов и фотографий

Пример системы, организованной по типу модели клиент/сервер, показан на рис. 5.3. Это многопользовательская гипертекстовая система, предназначенная для поддержки библиотек фильмов и фотографий. В ней содержится несколько серверов, которые размещают различные типы медиафайлов и управляют ими. Видеофайлы требуется передавать быстро и синхронно, но с относительно малым разрешением. Они могут храниться в сжатом состоянии. Фотографии должны передаваться с высоким разрешением. Каталоги должны обеспечивать работу с множеством запросов и поддерживать связи с использованием гипертекстовой системы. Здесь клиентская программа является просто интегрированным интерфейсом пользователя.

Подход клиент/сервер можно использовать при реализации систем, основанных на репозитории, который поддерживается как сервер системы. Подсистемы, имеющие доступ к репозиторию, являются клиентами. Но обычно каждая подсистема управляет собственными данными. Во время работы серверы и клиенты обмениваются данными, однако при обмене большими объемами данных могут возникнуть проблемы, связанные с пропускной способностью сети. Правда, с развитием все более быстрых сетей эта проблема теряет свое значение.

Наиболее важное преимущество модели клиент/сервер состоит в том, что она является распределенной архитектурой. Ее эффективно использовать в сетевых системах с множеством распределенных

процессоров. В систему легко добавить новый сервер и интегрировать его с остальной частью системы или же обновить серверы, не воздействуя на другие части системы.

5.4. Модель абстрактной машины

Модель архитектуры абстрактной машины (иногда называемая многоуровневой моделью) моделирует взаимодействие подсистем. Она организует систему в виде набора уровней, каждый из которых предоставляет свои сервисы. Каждый уровень определяет *абстрактную машину,* машинный язык которой (сервисы, предоставляемые уровнем) используется для реализации следующего уровня абстрактной машины. Например, наиболее распространенный способ реализации языка программирования состоит в определении идеальной "языковой машины" и компилировании программ, написанных на данном языке, в код этой машины. На следующем шаге трансляции код абстрактной машины конвертируется в реальный машинный код.

Хорошо известным примером такого похода может служить модель OSI* сетевых протоколов, обсуждаемая в разделе 5.4. Другим примером является трехуровневая модель среды программирования на языке Ada. На рис. 5.4 изображена подобная модель и показано, как с помощью модели абстрактной машины можно представить систему администрирования версий.

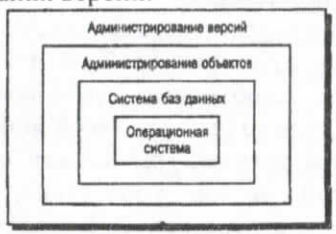

Рис. 5.4. Модель абстрактной машины для системы администрирования версий

Система администрирования версий основана на управлении версиями объектов и предоставляет средства для полного управления конфигурацией системы. Для поддержки средств управления конфигурацией используется система администрирования объектов, поддерживающая систему базы данных и сервисы управления объектами. В свою очередь, в системе баз данных поддерживаются

различные сервисы, например управления транзакциями, отката назад, восстановления и управления доступом. Для управления базами данных используются средства основной операционной системы и ее файловая система.

Многоуровневый подход обеспечивает пошаговое развитие систем – при разработке какого-либо уровня предоставляемые им сервисы становятся доступны пользователям. Кроме того, такая архитектура легко изменяема и переносима на разные платформы. Изменение интерфейса любого уровня повлияет только на смежный уровень. Так как в многоуровневых системах зависимости от машинной платформы локализованы на внутренних уровнях, такие системы можно реализовать на других платформах, поскольку потребуется изменить только самые внутренние уровни.

Недостатком многоуровневого подхода является довольно сложная структура системы. Основные средства, такие как управление, файлами, необходимые всем абстрактным машинам, предоставляются внутренними уровнями. Поэтому сервисам, запрашиваемым пользователем, возможно, потребуется доступ к внутренним уровням абстрактной машины. Такая ситуация приводит к разрушению модели, так как внешний уровень зависит не только от предшествующего ему уровня, но и от более низких уровней.

5.5. Модели управления

В модели структуры системы показаны все подсистемы, из которых она состоит. Для того чтобы подсистемы функционировали как единое целое, необходимо управлять ими. В структурных моделях нет (и не должно быть) никакой информации по управлению. Однако разработчик архитектуры должен организовать подсистемы согласно некоторой модели управления, которая дополняла бы имеющуюся модель структуры. В моделях управления на уровне архитектуры проектируется поток управления между подсистемами.

Можно выделить два основных типа управления в программных системах.

1. *Централизованное управление.* Одна из подсистем полностью отвечает за управление, запускает и завершает работу остальных подсистем. Управление от первой подсистемы может

перейти к другой подсистеме, однако потом обязательно возвращается к первой.

2. *Управление, основанное на событиях.* Здесь вместо одной подсистемы, ответственной за управление, на внешние события может отвечать любая подсистема. События, на которые реагирует система, могут происходить либо в других подсистемах, либо во внешнем окружении системы.

Модель управления дополняет структурные модели. Все описанные ранее структурные модели можно реализовать с помощью централизованного управления или управления, основанного на событиях.

5.6. Централизованное управление

В модели централизованного управления одна из систем назначается главной и управляет работой других подсистем. Такие модели можно разбить на два класса, в зависимости от того, последовательно или параллельно реализовано выполнение управляемых подсистем.

1. *Модель вызова-возврата.* Это известная модель организации вызова программных процедур "сверху вниз", в которой управление начинается на вершине иерархии процедур и через вызовы передается на более нижние уровни иерархии. Данная модель применима только в последовательных системах.

2. *Модель диспетчера.* Применяется в параллельных системах. Один системный компонент назначается диспетчером и управляет запуском, завершением и координированием других процессов системы. Процесс (выполняемая подсистема или модуль) может протекать параллельно с другими процессами. Модель такого типа применима также в последовательных системах, где управляющая программа вызывает отдельные подсистемы в зависимости от значений некоторых переменных состояния. Обычно такое управление реализуется через оператор case.

Модель вызова-возврата представлена на рис. 5.5. Из главной программы можно вызвать подпрограммы 1, 2 и 3, из подпрограммы 1 – подпрограммы 1.1 и 1.2, из подпрограммы 3 – подпрограммы 3.1 и 3.2 и т.д. Такая модель выполнения подпрограмм *не является*

структурной – подпрограмма 1.1 не обязательно является частью подпрограммы 1.

Рис. 5.5. Модель вызова-возврата

Подобная модель встроена в языки программирования Ada, Pascal и C. Управление переходит от программы, расположенной на самом верхнем уровне иерархии, к подпрограмме более нижнего уровня. Затем происходит возврат управления в точку вызова подпрограммы. За управление отвечает та подпрограмма, которая выполняется в текущий момент; она может либо вызывать другие подпрограммы, либо вернуть управление вызвавшей ее подпрограмме. Несовершенство данного стиля программирования при возврате к определенной точке в программе очевидно.

Модель вызова-возврата можно использовать на уровне модулей для управления функциями и объектами. Подпрограммы в языке программирования, которые вызываются из других подпрограмм, являются естественно функциональными. Однако во многих объектно-ориентированных системах операции в объектах (методы) реализованы в виде процедур или функций. Например, объект Java запрашивает сервис из другого объекта посредством вызова соответствующего метода.

Жесткая и ограниченная природа модели вызова-возврата является одновременно и преимуществом и недостатком. Преимущества модели проявляются в относительно простом анализе потоков управления, а также при выборе системы, отвечающей за конкретный ввод данных.

На рис. 6.6 представлена модель централизованного управления для параллельной системы. Подобная модель часто используется в "мягких" системах реального времени, в которых нет чересчур строгих временных. ограничений.

Рис. 5.6. Модель централизованного управления для системы реального времени

Контроллер системы, в зависимости от переменных состояния системы, определяет моменты запуска или завершения процессов. Он проверяет, генерируется ли в остальных процессах информация, для того чтобы затем обработать ее или передать другим процессам на обработку. Обычно контроллер работает постоянно, проверяя датчики и другие процессы или отслеживая изменения состояния, поэтому данную модель иногда называют моделью с обратной связью.

5.7. Архитектура распределенных систем

В настоящее время практически все большие программные системы являются распределенными. Распределенной называется такая система, в которой обработка информации сосредоточена не на одной вычислительной машине, а распределена между несколькими компьютерами. При проектировании распределенных систем, которое имеет много общего с проектированием любого другого ПО, все же следует учитывать ряд специфических особенностей. Некоторые из них уже упоминалось во введении к главе 10 при рассмотрении архитектуры клиент/сервер, здесь они обсуждаются более подробно.

Поскольку в наши дни распределенные системы получили широкое распространение, разработчики ПО должны быть знакомы с особенностями их проектирования. До недавнего времени все большие системы в основном являлись централизованными, которые запускались на одной главной вычислительной машине (мэйнфрейме) с подключенными к ней терминалами. Терминалы практически не

занимались обработкой информации – все вычисления выполнялись на главной машине. Разработчикам таких систем не приходилось задумываться о проблемах распределенных вычислений.

Все современные программные системы можно разделить на три больших класса.

1. Прикладные программные системы, предназначенные для работы только на одном персональном компьютере или рабочей станции. К ним относятся текстовые процессоры, электронные таблицы, графические системы и т.п.

2. Встроенные системы, предназначенные для работы на одном процессоре либо на интегрированной группе процессоров. К ним относятся системы управления бытовыми устройствами, различными приборами и др.

3. Распределенные системы, в которых программное обеспечение выполняется на слабо интегрированной группе параллельно работающих процессоров, связанных через сеть. К ним относятся системы банкоматов, принадлежащих какому-либо банку, издательские системы, системы ПО коллективного пользования и др.

В настоящее время между перечисленными классами программных систем существуют четкие границы, которые в дальнейшем будут все более стираться. Со временем, когда высокоскоростные беспроводные сети станут широкодоступными, появится возможность динамически интегрировать устройства со встроенными программными системами, например электронные органайзеры с более общими системами.

В книге выделено шесть основных характеристик распределенных систем.

1. *Совместное использование ресурсов.* Распределенные системы допускают совместное использование аппаратных и программных ресурсов, например жестких дисков, принтеров, файлов, компиляторов и т.п., связанных посредством сети. Очевидно, что разделение ресурсов возможно также в многопользовательских системах, однако в этом случае за предоставление ресурсов и их управление должен отвечать центральный компьютер.

2. *Открытость.* Это возможность расширять систему путем добавления новых ресурсов. Распределенные системы – это открытые

системы, к которым подключают аппаратное и программное обеспечение от разных производителей.

3. *Параллельность.* В распределенных системах несколько процессов могут одновременно выполняться на разных компьютерах в сети. Эти процессы могут (но не обязательно) взаимодействовать друг с другом во время их выполнения.

4. *Масштабируемость.* В принципе все распределенные системы являются масштабируемыми: чтобы система соответствовала новым требованиям, ее можно наращивать посредством добавления новых вычислительных ресурсов. Но на практике наращивание может ограничиваться сетью, объединяющей отдельные компьютеры системы. Если подключить много новых машин, пропускная способность сети может оказаться недостаточной.

5. *Отказоустойчивость.* Наличие нескольких компьютеров и возможность дублирования информации означает, что распределенные системы устойчивы к определенным аппаратным и программным ошибкам (см. главу 18). Большинство распределенных систем в случае ошибки, как правило, могут поддерживать хотя бы частичную функциональность. Полный сбой в работе системы происходит только в случае сетевых ошибок.

6. *Прозрачность.* Это свойство означает, что пользователям предоставлен полностью прозрачный доступ к ресурсам и в то же время от них скрыта информация о распределении ресурсов в системе. Однако во многих случаях конкретные знания об организации системы помогают пользователю лучше использовать ресурсы.

Разумеется, распределенным системам присущ ряд недостатков.

- *Сложность.* Распределенные системы сложнее централизованных. Намного труднее понять и оценить свойства распределенных систем в целом, а также тестировать эти системы. Например, здесь производительность системы зависит не от скорости работы одного процессора, а от полосы пропускания сети и скорости работы разных процессоров. Перемещая ресурсы из одной части системы в другую, можно радикально повлиять на производительность системы.
- *Безопасность.* Обычно доступ к системе можно получить с нескольких разных машин, сообщения в сети могут просматриваться

или перехватываться. Поэтому, в распределенной системе намного сложнее поддерживать безопасность.

- *Управляемость.* Система может состоять из разнотипных компьютеров, на которых могут быть установлены разные версии операционных систем. Ошибки на одной машине могут распространиться на другие машины с непредсказуемыми последствиями. Поэтому требуется значительно больше усилий, чтобы управлять и поддерживать систему в рабочем состоянии.
- *Непредсказуемость.* Как известно всем пользователям Web-сети, реакция распределенных систем на определенные события непредсказуема и зависит от полной загрузки системы, ее организации и сетевой нагрузки. Так как все эти параметры могут постоянно меняться, время, затраченное на выполнение запроса пользователя, в тот или иной момент может существенно различаться.

При обсуждении преимуществ и недостатков распределенных систем в книге определяется ряд критических проблем проектирования таких систем (табл. 5.1). В этой главе основное внимание уделяется архитектуре распределенного ПО, так как полагают, что при разработке программных продуктов наиболее значимым является именно этот момент. Если вас интересуют другие темы, обратитесь к специализированным книгам по распределенным системам.

Таблица 5.1. Проблемы проектирования распределенных систем

Проблема проектирования	Описание
Идентификация ресурсов	Ресурсы в распределенной системе располагаются на разных компьютерах, поэтому систему имен ресурсов следует продумать так, чтобы пользователи могли без труда открывать необходимые им ресурсы и ссылаться на них. Примером может служить система унифицированного указателя ресурсов URL, которая определяет адреса Web-страниц. Без легко воспринимаемой и универсальной системы идентификации большая часть ресурсов окажется недоступной пользователям системы

Коммуникации	Универсальная работоспособность Internet и эффективная реализация протоколов TCP/IP в Internet для большинства распределенных систем служат примером наиболее эффективного способа организации взаимодействия между компьютерами. Однако там, где на производительность, надежность и прочее накладываются специальные требования, можно воспользоваться альтернативными способами системных коммуникаций
Качество системного сервиса	Качество сервиса, предлагаемое системой, отражает ее производительность, работоспособность и надежность. На качество сервиса влияет целый ряд факторов: распределение системных процессов, распределение ресурсов, системные и сетевые аппаратные средства и возможности адаптации системы
Архитектура программного обеспечения	Архитектура программного обеспечения описывает распределение системных функций по компонентам системы, а также распределение этих компонентов по процессорам. Если необходимо поддерживать высокое качество системного сервиса, выбор правильной архитектуры оказывается решающим фактором

Задача разработчиков распределенных систем – спроектировать программное или аппаратное обеспечение так, чтобы предоставить все необходимые характеристики распределенной системы. А для этого требуется знать преимущества и недостатки различных архитектур распределенных систем. Здесь выделяется два родственных типа архитектур распределенных систем.

1. *Архитектура клиент/сервер.* В этой модели систему можно представить как набор сервисов, предоставляемых серверами

клиентам. В таких системах серверы и клиенты значительно отличаются друг от друга.

2. *Архитектура распределенных объектов.* В этом случае между серверами и клиентами нет различий и систему можно представить как набор взаимодействующих объектов, местоположение которых не имеет особого значения. Между поставщиком сервисов и их пользователями не существует различий.

В распределенной системе разные системные компоненты могут быть реализованы на разных языках программирования и выполняться на разных типах процессоров. Модели данных, представление информации и протоколы взаимодействия – все это не обязательно будет однотипным в распределенной системе. Следовательно, для распределенных систем необходимо такое программное обеспечение, которое могло бы управлять этими разнотипными частями и гарантировать взаимодействие и обмен данными между ними. *Промежуточное программное обеспечение* относится именно к такому классу ПО. Оно находится как бы посередине между разными частями распределенных компонентов системы.

В статье описаны различные типы промежуточного ПО, которое может поддерживать распределенные вычисления. Как правило, такое ПО составляется из готовых компонентов и не требует от разработчиков специальных доработок. В качестве примеров промежуточного ПО можно привести программы управления взаимодействием с базами данных, менеджеры транзакций, преобразователи данных, коммуникационные инспекторы и др. Далее в главе будет описана структура распределенных систем как класс промежуточного ПО.

Распределенные системы обычно разрабатываются на основе объектно-ориентированного подхода. Эти системы создаются из слабо интегрированных частей, каждая из которых может непосредственно взаимодействовать как с пользователем, так и с другими частями системы. Эти части по возможности должны реагировать на независимые события. Программные объекты, построенные на основе таких принципов, являются естественными компонентами распределенных систем.

5.7.1. Многопроцессорная архитектура

Самой простой распределенной системой является многопроцессорная система. Она состоит из множества различных процессов, которые могут (но не обязательно) выполняться на разных процессорах. Данная модель часто используется в больших системах реального времени. В принципе все процессы, связанные со сбором информации, принятием решений и управлением исполнительным механизмом, могут выполняться на одном процессоре под управлением планировщика заданий. Использование нескольких процессоров повышает производительность системы и ее способность к восстановлению. Распределение процессов между процессорами может переопределяться (присуще критическим системам) или же находиться под управлением диспетчера процессов.

На рис. 5.7 показан пример системы такого типа. Это упрощенная модель системы управления транспортным потоком. Группа распределенных датчиков собирает информацию о величине потока. Собранные данные перед отправкой в диспетчерскую обрабатываются на месте. На основании полученной информации операторы принимают решения и управляют светофорами. В этом примере для управления датчиками, диспетчерской и светофорами имеются отдельные логические процессы. Это могут быть как отдельные процессы, так и группа процессов. В нашем примере они выполняются на разных процессорах.

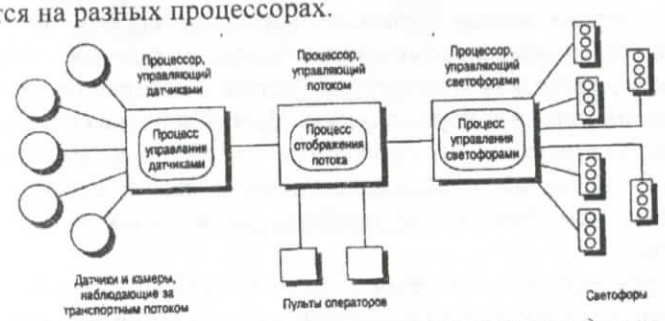

Рис. 5.7. Многопроцессорная система управления движением транспорта

Системы ПО, одновременно выполняющие множество процессов, не обязательно являются распределенными. Если в

системе более одного процессора, реализовать распределение процессов не представляет труда. Однако при создании многопроцессорных программных систем не обязательно отталкиваться только от распределенных систем.

5.8. Архитектура клиент/сервер

В архитектуре клиент/сервер программное приложение моделируется как набор сервисов, предоставляемых серверами, и множество клиентов, использующих эти сервисы. Клиенты должны знать о доступных (имеющихся) серверах, хотя могут и не иметь представления о существовании других клиентов. Как видно из рис. 5.8, на котором представлена схема распределенной архитектуры клиент/сервер, клиенты и серверы представляют разные процессы.

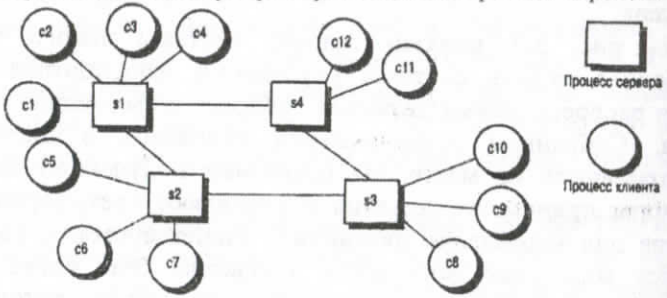

Рис. 5.8. Система клиент/сервер

В системе между процессами и процессорами не обязательно должно соблюдаться отношение "один к одному". На рис. 5.7 показана физическая архитектура системы, которая состоит из шести клиентских машин и двух серверов. На них запускаются клиентские и серверные процессы, изображенные на рис. 5.8. В общем случае, говоря о клиентах и серверах, подразумевают скорее логические процессы, чем физические машины, на которых выполняются эти процессы.

Архитектура системы клиент/сервер должна отражать логическую структуру разрабатываемого программного приложения. На рис. 5.9 предлагается еще один взгляд на программное приложение, структурированное в виде трех уровней. Уровень представления обеспечивает информацию для пользователей и взаимодействие с ними. Уровень выполнения приложения реализует

логику работы приложения. На уровне управления данными выполняются все операции с базами данных. В централизованных системах между этими уровнями нет четкого разделения. Однако при проектировании распределенных систем необходимо разделять эти уровни, чтобы затем расположить каждый уровень на разных компьютерах.

Самой простой архитектурой клиент/сервер является двухуровневая, в которой приложение состоит из сервера (или множества идентичных серверов) и группы клиентов. Существует два вида такой архитектуры (рис. 5.9).

1. *Модель тонкого клиента.* В этой модели вся работа приложения и управление данными выполняются на сервере. На клиентской машине запускается только ПО уровня представления.
2. *Модель толстого клиента.* В этой модели сервер только управляет данными. На клиентской машине реализована работа приложения и взаимодействие с пользователем системы.

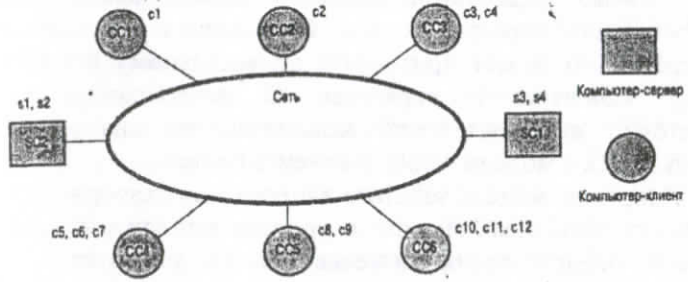

Рис. 5.9. Компьютеры в сети клиент/сервер

Рис. 5.10. Уровни программного приложения

Тонкий клиент двухуровневой архитектуры – самый простой способ перевода существующих централизованных систем в архитектуру клиент/сервер. Пользовательский интерфейс в этих

системах "переселяется" на персональный компьютер, а само программное приложение выполняет функции сервера, т.е. выполняет все процессы приложения и управляет данными. Модель тонкого клиента можно также реализовать там, где клиенты представляют собой обычные сетевые устройства, а не персональные компьютеры или рабочие станции. Сетевые устройства запускают Internet-браузер и пользовательский интерфейс, реализованный внутри системы.

Рис. 5.11. Модели тонкого и толстого клиентов

Главный недостаток модели тонкого клиента – большая загруженность сервера и сети. Все вычисления выполняются на сервере, а это может привести к значительному сетевому трафику между клиентом и сервером. В современных компьютерах достаточно вычислительной мощности, но она практически не используется в модели тонкого клиента банка.

Напротив, модель толстого клиента использует вычислительную мощность локальных машин: и уровень выполнения приложения, и уровень представления помещаются на клиентский компьютер. Сервер здесь, по существу, является сервером транзакций, который управляет всеми транзакциями баз данных. Примером архитектуры такого типа могут служить системы банкоматов, в которых банкомат является клиентом, а сервер – центральным компьютером, обслуживающим базу данных по расчетам с клиентами.

На рис. 5.11. показана сетевая система банкоматов. Заметим, что банкоматы связаны с базой данных расчетов не напрямую, а через монитор телеобработки. Этот монитор является промежуточным звеном, которое взаимодействует с удаленными клиентами и организует запросы клиентов в последовательность транзакций для работы с базой данных. Использование последовательных транзакций при возникновении сбоев позволяет системе восстановиться без потери данных.

Рис. 5.12. Система клиент/сервер для сети банкоматов

Поскольку в модели толстого клиента выполнение программного приложения организовано более эффективно, чем в модели тонкого клиента, управлять такой системой сложнее. Здесь функции приложения распределены между множеством разных машин. Необходимость замены приложения приводит к его повторной инсталляции на всех клиентских компьютерах, что требует больших расходов, если в системе сотни клиентов.

Появление языка Java и загружаемых апплетов позволили разрабатывать модели клиент/сервер, которые находятся где-то посередине между моделями тонкого и толстого клиента. Часть программ, составляющих приложение, можно загружать на клиентской машине как апплеты Java и тем самым разгрузить сервер. Интерфейс пользователя строится посредством Web-браузера, который запускает апплеты Java. Однако Web-браузеры от различных производителей и даже различные версии Web-браузеров от одного производителя не всегда выполняются одинаково. Более ранние версии браузеров на старых машинах не всегда могут запустить апплеты Java. Следовательно, такой подход можно использовать только тогда, когда вы уверены, что у всех пользователей системы установлены браузеры, совместимые с Java.

В двухуровневой модели клиент/сервер существенной проблемой является размещение на двух компьютерных системах трех логических уровней – представления, выполнения приложения и управления данными. Поэтому в данной модели часто возникают либо проблемы с масштабируемостью и производительностью, если выбрана модель тонкого клиента, либо проблемы, связанные с управлением системой, если используется модель толстого клиента. Чтобы избежать этих проблем, необходимо применить альтернативный подход – трехуровневую модель архитектуры

клиент/сервер (рис. 5.13). В этой архитектуре уровням представления, выполнения приложения и управления данными соответствуют отдельные процессы.

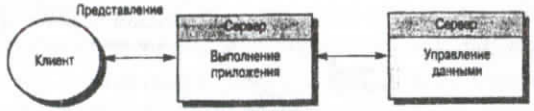

Рис. 5.13. Трехуровневая архитектура клиент/сервер

Архитектура ПО, построенная по трехуровневой модели клиент/сервер, не требует, чтобы в сеть были объединены три компьютерных системы. На одном компьютере-сервере можно запустить и выполнение приложения, и управление данными как отдельные логические серверы. В то же время, если требования к системе возрастут, можно будет относительно просто разделить выполнение приложения и управление данными и выполнять их на разных процессорах.

Банковскую систему, использующую Internet-сервисы, можно реализовать с помощью трехуровневой архитектуры клиент/сервер. База данных расчетов (обычно расположенная на главном компьютере) предоставляет сервисы управления данными, Web-сервер поддерживает сервисы приложения, например средства перевода денег, генерацию отчетов, оплату счетов и др. А компьютер пользователя с Internet-браузером является клиентом. Как показано на рис. 5.14, эта система масштабируема, так как в нее относительно просто добавить новые Web-серверы при увеличении количества клиентов.

Использование трехуровневой архитектуры в этом примере позволило оптимизировать передачу данных между Web-сервером и сервером базы данных. Взаимодействие между этими системами не обязательно строить на стандартах Internet, можно использовать более быстрые коммуникационные протоколы низкого уровня. Обычно информацию от базы данных обрабатывает эффективное промежуточное ПО, которое поддерживает запросы к базе данных на языке структурированных запросов SQL.

В некоторых случаях трехуровневую модель клиент/сервер можно перевести в многоуровневую, добавив в систему дополнительные серверы. Многоуровневые системы можно

использовать и там, где приложениям необходимо иметь доступ к информации, находящейся в разных базах данных. В этом случае объединяющий сервер располагается между сервером, на котором выполняется приложение, и серверами баз данных. Объединяющий сервер собирает распределенные данные и представляет их в приложении, таким образом, будто они находятся в одной базе данных.

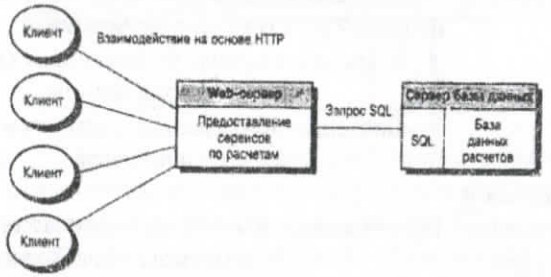

Рис. 5.14. Распределенная архитектура банковской системы с использованием Internet-сервисов

Разработчики архитектур клиент/сервер, выбирая наиболее подходящую, должны учитывать ряд факторов. В табл. 5.2 перечислены различные случаи применения архитектуры клиент/сервер.

Таблица 5.2. Применение разных типов архитектуры клиент/сервер

Архитектура	Приложения
	Наследуемые системы, в которых нецелесообразно разделять выполнение приложения и управления данными.
Двухуровневая архитектура тонкого клиента	Приложения с интенсивными вычислениями, например компиляторы, но с незначительным объемом управления данными.
	Приложения, в которых обрабатываются большие массивы данных (запросы), но с небольшим объемом вычислений в самом приложении

- Гибкость и масштабируемость системы. Для того чтобы справиться с системными нагрузками, можно создавать экземпляры системы с одинаковыми сервисами, которые будут предоставляться разными объектами или разными экземплярами (копиями) объектов. При увеличении нагрузки в систему можно добавить новые объекты, не прерывая при этом работу других ее объектов.
- Существует возможность динамически переконфигурировать систему посредством объектов, мигрирующих в сети по запросам. Объекты, предоставляющие сервисы, могут мигрировать на тот же процессор, что и объекты, запрашивающие сервисы, тем самым повышая производительность системы.

В процессе проектирования систем архитектуру распределенных объектов можно использовать двояко.

1. В виде логической модели, которая позволяет разработчикам структурировать и спланировать систему. В этом случае функциональность приложения описывается только в терминах и комбинациях сервисов. Затем разрабатываются способы предоставления сервисов с помощью нескольких распределенных объектов. На этом уровне, как правило, проектируют крупно модульные объекты, которые предоставляют сервисы, отражающие специфику конкретной области приложения. Например, в программу учета розничной торговли можно включить объекты, которые бы вели учет состояния запасов, отслеживали взаимодействие с клиентами, классифицировали товары и др.
2. Как гибкий подход к реализации систем клиент/сервер. В этом случае логическая модель системы – это модель клиент/сервер, в которой клиенты и серверы реализованы как распределенные объекты, взаимодействующие посредством программной шины. При таком подходе легко заменить систему, например двухуровневую на многоуровневую. В этом случае ни сервер, ни клиент не могут быть реализованы в одном объекте, однако могут состоять из множества небольших объектов, каждый из которых предоставляет определенный сервис.

Примером системы, которой подходит архитектура распределенных объектов, может служить система обработки данных, хранящихся в разных базах данных (рис. 5.16). В этом примере любую базу данных можно представить как объект с

интерфейсом, предоставляющим доступ к данным "только чтение". Каждый из объектов-интеграторов занимается определенными типами зависимостей между данными, собирая информацию из баз данных, чтобы попытаться проследить эти зависимости.

Объекты-визуализаторы взаимодействуют с объектами-интеграторами для представления данных в графическом виде либо для составления отчетов по анализируемым данным.

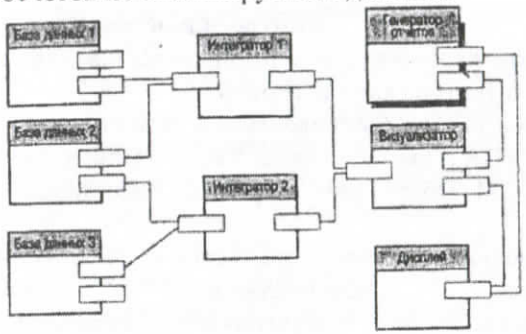

Рис. 5.16. Архитектура распределенной системы обработки данных

Для такого типа приложений архитектура распределенных объектов подходит больше, чем архитектура клиент/сервер, по трем причинам.

1. В этих системах (в отличие, например, от системы банкоматов) нет одного поставщика сервиса, на котором были бы сосредоточены все сервисы управления данными.
2. Можно увеличивать количество доступных баз данных, не прерывая работу системы, поскольку каждая база данных представляет собой просто объект. Эти объекты поддерживают упрощенный интерфейс, который управляет доступом к данным. Доступные базы данных можно разместить на разных машинах.
3. Посредством добавления новых объектов-интеграторов можно отслеживать новые типы зависимостей между данными.

Главным недостатком архитектур распределенных объектов является то, что их сложнее проектировать, чем системы клиент/сервер. Оказывается, что системы клиент/сервер предоставляют более естественный подход к созданию распределенных систем. В нем отражаются взаимоотношения между

людьми, при которых одни люди пользуются услугами других людей, специализирующихся на предоставлении конкретных услуг. Намного труднее разработать систему в соответствии с архитектурой распределенных объектов, поскольку индустрия создания ПО пока еще не накопила достаточного опыта в проектировании и разработке крупно модульных объектов.

Контрольные вопросы

1. Какие характеристики распределенных систем существуют?
2. Архитектура клиент/сервер.
3. Архитектура распределенных объектов
4. CORBA (Common Object Request Broker Architecture).
5. DCOM (Distributed Component Object Model)

Ключевые слова: *совместное использование ресурсов, открытость, параллельность, масштабируемость, отказоустойчивость, прозрачность, сложность, безопасность, управляемость, непредсказуемость, идентификация ресурсов, коммуникации, качество системного сервиса, архитектура программного обеспечения, архитектура клиент/сервер, архитектура распределенных объектов, промежуточное программное обеспечение, модель тонкого клиента, модель толстого клиента, CORBA, DCOM.*

Keywords: *resource sharing, openness, concurrency, scalability, fault tolerance, transparency, complexity, security, manageability, unpredictability, identification of resources, communication, quality of service system, software architecture, client / server architecture, the architecture of distributed object middleware model thin client, thick client model, CORBA, DCOM.*

Kalit so'zlar: *manbalarni umumiy ishlatish, ochiqlik, paralellik, masshtablashtirish, rad etishga chidamiylik, shaffoflik, qiyinlik, havfsizlik, boshqarilish, oldindan ko'rib bilmaslik, manbalarni identifikatsiyalash, kommunikatsiyalar, tizim servisi sifati, dasturiy ta'minot arhitekturasi, klient/server arhitekturasi, ajratilgan obektlar arhitekturasi, o'rtadagi dasturiy ta'minot, nozik mijoz modeli, qalin mijoz modeli, CORBA, DCOM.*

Упражнения

1. Объясните, почему распределенные системы всегда более масштабируемы, чем централизованные. Какой вероятный предел масштабируемости программных систем?
2. В чем основное отличие между моделями толстого и тонкого клиента в разработке систем клиент/сервер? Объясните, почему использование Java как языка реализации сглаживает различия между этими моделями?
3. На основе модели приложения, которые могут возникнуть при преобразовании системы 1980-х годов, реализованной на мейнфрейме и предназначенной для работы в сфере здравоохранения, в систему архитектуры клиент/сервер.
4. Распределенные системы, базирующиеся на модели клиент/сервер, разрабатывались с 1980-х годов, но только недавно такие системы, основанные на распределенных объектах, были реализованы. Приведите три причины, почему так получилось.
5. Объясните, почему использование распределенных объектов совместно с брокером запросов к объектам упрощает реализацию масштабируемых систем клиент/сервер. Проиллюстрируйте свой ответ примером.
6. Каким образом используется язык IDL для поддержки взаимодействия между объектами, реализованными на разных языках программирования? Объясните, почему такой подход может вызвать проблемы, связанные с производительностью, если между языками, которые используются при реализации объектов, имеются радикальные различия.
7. Какие базовые средства должен предоставлять брокер запросов к объектам?
8. Можно показать, что разработка стандартов CORBA для горизонтальных и вертикальных компонентов ограничивает конкуренцию. Если они уже созданы и адаптированы, это препятствует разработке лучших компонентов более мелкими компаниями. Обсудите роль стандартизации в поддержке или ограничении конкуренции на рынке программного обеспечения.

ГЛАВА 6. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

Объектно-ориентированное проектирование представляет собой стратегию, в рамках которой разработчики системы вместо операций и функций мыслят в понятиях *объекты*. Программная система состоит из взаимодействующих объектов, которые имеют собственное локальное состояние и могут выполнять определенный набор операций, определяемый состоянием объекта (рис. 6.1). Объекты скрывают информацию о представлении состояний и, следовательно, ограничивают к ним доступ. Под процессом объектно-ориентированного проектирования подразумевается проектирование классов объектов и взаимоотношений между этими классами. Когда проект реализован в виде исполняемой программы, все необходимые объекты создаются динамически с помощью определений

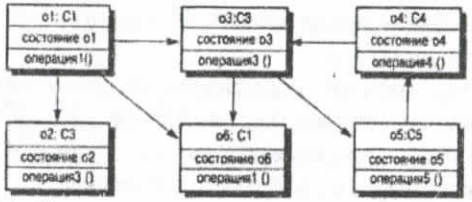

Рис. 6.1. Система взаимодействующих объектов

Объектно-ориентированное проектирование – только часть *объектно-ориентированного процесса разработки системы*, где на протяжении всего процесса создания ПО используется объектно-ориентированный подход. Этот подход подразумевает выполнение трех этапов.

- *Объектно-ориентированный анализ.* Создание объектно-ориентированной модели предметной области приложения ПО. Здесь объекты отражают реальные объекты-сущности, также определяются операции, выполняемые объектами.
- *Объектно-ориентированное проектирование.* Разработка объектно-ориентированной модели системы ПО (системной архитектуры) с учетом системных требований. В объектно-ориентированной модели определение всех объектов подчинено решению конкретной задачи.

- *Объектно-ориентированное программирование.* Реализация архитектуры (модели) системы с помощью объектно-ориентированного языка программирования. Такие языки, например Java, непосредственно выполняют реализацию определенных объектов и предоставляют средства для определения классов объектов.

Данные этапы могут "перетекать" друг в друга, т.е. могут не иметь четких рамок, причем на каждом этапе обычно применяется одна и та же система нотации. Переход на следующий этап приводит к усовершенствованию результатов предыдущего этапа путем более детального описания определенных ранее классов объектов и определения новых классов. Так как данные скрыты внутри объектов, детальные решения о представлении данных можно отложить до этапа реализации системы. В некоторых случаях можно также не спешить с принятием решений о расположении объектов и о том, будут ли эти объекты последовательными или параллельными. Все сказанное означает, что разработчики ПО не стеснены деталями реализации системы.

Объектно-ориентированные системы можно рассматривать как совокупность автономных и в определенной мере независимых объектов. Изменение реализации какого-нибудь объекта или добавление новых функций не влияет на другие объекты системы. Часто существует четкое соответствие между реальными объектами (например, аппаратными средствами) и управляющими ими объектами программной системы. Такой подход облегчает понимание и реализацию проекта.

Потенциально все объекты являются повторно используемыми компонентами, так как они независимо инкапсулируют данные о состоянии и операции. Архитектуру ПО можно разрабатывать на базе объектов, уже созданных в предыдущих проектах. Такой подход снижает стоимость проектирования, программирования и тестирования ПО. Кроме того, появляется возможность использовать стандартные объекты, что уменьшает риск, связанный с разработкой программного обеспечения. В книгах предлагаются различные методы объектно-ориентированного проектирования. В этих методах на протяжении всего процесса проектирования используется единообразная нотация, принятая в UML. В данной

методов реализованы аналогично вызовам процедур или функций в языках программирования, например таких, как C или Ada.

Если запросы к сервису реализованы именно таким образом, взаимодействие между объектами синхронно. Это означает, что объект, отправивший запрос к сервису, ожидает окончания выполнения запроса. Однако, если объекты реализованы как параллельные процессы или потоки, взаимодействие объектов может быть асинхронным. Отправив запрос к сервису, объект может продолжить работу и не ждать, пока сервис выполнит его запрос. Ниже в этом разделе показано, каким образом можно реализовать объекты как параллельные процессы.

Эти частные классы объектов полностью совместимы с основными классами, но содержат больше информации. В системе обозначений UML направление обобщения указывается стрелками, направленными на родительский класс. В объектно-ориентированных языках программирования обобщение обычно реализуется через механизм наследования. Производный класс (класс-потомок) наследует атрибуты и операции от родительского класса.

Пример такой иерархии изображен на рис. 6.3, где показаны различные классы работников. Классы, расположенные внизу иерархии, имеют те же атрибуты и операции, что и родительские классы, но могут содержать новые атрибуты и операции или же изменять имеющиеся в родительских классах. Если в модели используется имя родительского класса, значит, объект в системе может быть определен либо самим классом, либо любым из его потомков.

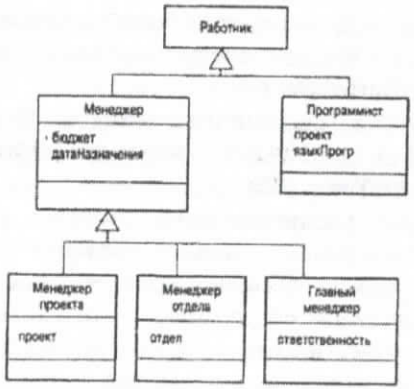

Рис. 6.3. Иерархия обобщения

На рис. 6.3 видно, что класс **Менеджер** обладает всеми атрибутами и операциями класса **Работник** и, кроме того, имеет два новых атрибута: ресурсы, которыми управляет менеджер (**бюджет**), и дата назначения его на должность менеджера (**датаНазначения**). Также добавлены новые атрибуты в класс **Программист**. Один из них определяет проект, над которым работает программист, другой характеризует уровень его профессионализма при использовании определенного языка программирования (**языкПрогр**). Таким образом, объекты класса **Менеджер** и **Программист** можно использовать вместо объектов класса **Работник**.

Объекты, являющиеся членами класса объектов, взаимодействуют с другими объектами. Эти взаимоотношения моделируются с помощью описания связей (ассоциаций) между классами объектов. В UML связь обозначается линией, которая соединяет классы объектов, причем линия может быть снабжена информацией о данной связи. На рис. 6.4 показаны связи между объектами классов **Работник** и **Отдел** и между объектами классов **Работник** и **Менеджер**.

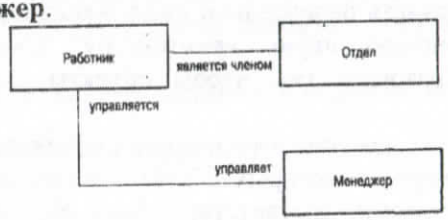

Рис. 6.4. Модель связей

Связи представляют самые общие отношения и часто используются в UML там, где требуется указать, что какое-то свойство объекта является связанным с объектом или же реализация метода объекта полагается на связанный объект. Однако в принципе тип связи может быть каким угодно. Одним из наиболее распространенных типов связи, который служит для создания новых объектов из уже имеющихся, является агрегирование.

6.2. Параллельные объекты

В общем случае объекты запрашивают сервис от любого объекта посредством передачи ему сообщения "запрос к сервису". Обычно нет необходимости в последовательном выполнении, при

котором один объект ожидает завершения работы сервиса по сделанному запросу. Общая модель взаимодействия объектов позволяет их одновременное выполнение в виде параллельных процессов. Такие объекты могут выполняться на одном компьютере или на разных машинах как распределенные объекты.

На практике в большинстве объектно-ориентированных языков программирования по умолчанию реализована модель последовательного выполнения, в которой запросы к сервисам объектов и вызовы функций реализованы одним и тем же способом. Например, на языке Java, когда объект, вызвавший объект **theList** (Список), создается из обычного класса объектов, это запишется так:

theList.append(17)

Здесь вызывается метод **append** (добавить), связанный с объектом **theList**, который добавляет элемент 17 в список **theList**, а выполнение объекта, сделавшего вызов, приостанавливается до тех пор, пока не завершится операция добавления. Однако в Java существует очень простой механизм потоков (threads), который позволяет создавать параллельно выполняющиеся объекты. Поэтому объектно-ориентированную архитектуру программной системы можно преобразовать так, чтобы объекты стали параллельными процессами.

Существует два типа параллельных объектов.

1. *Серверы,* в которых объект реализован как параллельный процесс с методами, соответствующими определенным операциям объекта. Методы запускаются в ответ на внешнее сообщение и могут выполняться параллельно с методами, связанными с другими объектами. По окончании всех действий выполнение объекта приостанавливается, и он ожидает дальнейших запросов к сервису.
2. *Активные объекты,* у которых состояние может изменяться посредством операций, выполняющихся внутри самого объекта. Процесс, представляющий объект, постоянно выполняет эти операции, а следовательно, никогда не останавливается.

Серверы наиболее полезны в распределенных средах, где вызывающий и вызываемый объекты выполняются на разных компьютерах. Время ответа, которое требуется сервису, заранее не известно, поэтому, где только можно следует спроектировать систему так, чтобы объект, отправивший запрос к сервису, не ждал, пока

сервис выполнит запрос. Также серверы могут использоваться на одной машине, где им требуется некоторое время для выполнения запроса (например, печать документа) и где есть вероятность отправки запросов к сервису от нескольких разных объектов.

Активные объекты используются там, где объектам необходимо обновлять свое состояние через определенные интервалы времени. Такие объекты характерны для систем реального времени, в которых объекты связаны с аппаратными устройствами, собирающими информацию из окружения среды. Методы объектов позволяют другим объектам получить доступ к информации, определяющей состояние объекта.

В листинге 6.1 показано, как на языке Java можно определить и реализовать активный объект. Данный класс объектов представляет бортовой радиомаяк-ответчик (transponder) самолета. С помощью спутниковой навигационной системы радиомаяк-ответчик отслеживает положение самолета. Он может отвечать на сообщения, приходящие от компьютеров, управляющих воздушными полетами. В ответ на запрос метод **givePosition** сообщает текущее положение самолета.

Листинг 6.1. Реализация активного объекта, использующего потоки языка Java

```
class Transponder extends Thread {
Position currentPosition ;
Coords c1, c2 ;
Satellite sat1, sat2 ;
Navigator theNavigator ;
public Position givePosition ()
{
return currentPosition;
}
public void run ()
{
while (true)
{
c1 = sat1.position ();
c2 = sat2.position ();
currentPosition = theNavigator.compute (c1, c2) ;
```

```
}
}
}//Transponder
```

Данный объект реализован как поток, где в непрерывном цикле метода **run** содержится код, вычисляющий положение самолета с помощью сигналов, полученных от спутников. В Java потоки создаются с помощью встроенного класса **Thread** (Поток), выступающего в объявлении классов в качестве базового.

6.3. Процесс объектно-ориентированного проектирования

В этом разделе процесс объектно-ориентированного проектирования показан на примере разработки структуры управляющей программной системы, встроенной в автоматизированную метеостанцию. Как отмечалось выше, есть несколько методов объектно-ориентированного проектирования, причем какого-либо предпочтительного метода или процесса проектирования не существует. Рассматриваемый здесь процесс является достаточно общим, т.е. состоит из операций, характерных для большинства процессов объектно-ориентированного проектирования. В этом отношении он сравним с процессом, предлагаемым языком UML, однако значительно упростили его.

Общий процесс объектно-ориентированного проектирования состоит из нескольких этапов.

1. Определение рабочего окружения системы и разработка моделей ее использования.
2. Проектирование архитектуры системы.
3. Определение основных объектов системы.
4. Разработка моделей архитектуры системы.
5. Определение интерфейсов объекта.

Процесс проектирования нельзя представить в виде простой схемы, в которой предполагается четкая последовательность этапов. Фактически все перечисленные этапы в значительной мере можно выполнять параллельно, с взаимным влиянием друг на друга. Как только разработана архитектура системы, определяются объекты и (частично или полностью) интерфейсы. После создания моделей объектов отдельные объекты можно переопределить, а это может привести к изменениям в архитектуре системы. Далее в этом разделе

каждый этап процесса проектирования обсуждается отдельно.

Пример, ПО, которым воспользуются для иллюстрации объектно-ориентированного проектирования, представляет собой часть системы, создающей метеорологические карты на основе автоматически собранных метеорологических данных. Подробное перечисление требований для такой системы займет много страниц. Однако, даже ограничившись кратким описанием системы, можно разработать ее общую архитектуру.

Одним из требований системы построения карты погоды является регулярное обновление метеорологических карт на основе данных, полученных от удаленных метеостанций и других источников, например наблюдателей, метеозондов и спутников. В ответ на запрос регионального компьютера системы обслуживания метеостанций передают ему свои данные.

Региональная компьютерная система объединяет данные из различных источников. Собранные данные архивируются и с помощью данных из этого архива и базы данных цифровых карт создается набор локальных метеорологических карт. Карты можно распечатать, направив их на специальный принтер, или же отобразить в разных форматах.

Из данного описания видно, что одна часть общей системы занимается сбором данных, другая обобщает данные, полученные из различных источников, третья выполняет архивирование данных и наконец четвертая создает метеорологические карты. На рис. 6.5 изображена одна из возможных архитектур системы, которую можно построить на основе предложенного описания. Она представляет собой многоуровневую архитектуру (обсуждаемую в главе 8), в которой отражены все этапы обработки данных в системе, т.е. сбор данных, обобщение данных, архивирование данных и создание карт. Такая многоуровневая архитектура вполне годится для нашей системы, так как каждый этап основывается только на обработке данных, выполненной на предыдущем этапе.

На рис. 6.5 показаны все уровни системы. Названия уровней заключены в прямоугольники, что в нотации UML обозначает подсистемы. Прямоугольники UML (т.е. подсистемы) – это набор объектов и других подсистем. Здесь используют это обозначение,

чтобы показать, что каждый уровень включает в себя множество других компонентов.

Рис. 6.5. Многоуровневая архитектура системы построения карт погоды

На рис. 6.6 изображена расширенная модель архитектуры, в которой показаны компоненты подсистем. Эти компоненты также очень абстрактны и построены на информации, содержащейся в описании системы. Продолжим рассматривать этот пример, уделяя особое внимание подсистеме **Метеостанция**, которая является частью уровня **Сбор данных**.

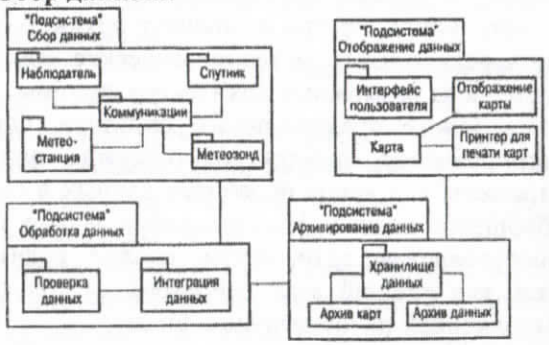

Рис. 6.6. Подсистемы в системе построения карт погоды

Окружение системы и модели ее использования

Первый этап в любом процессе проектирования состоит в выявлении взаимоотношений между проектируемым программным обеспечением и его окружением. Выявление этих взаимоотношений помогает решить, как обеспечить необходимую функциональность

системы и как структурировать систему, чтобы она могла эффективно взаимодействовать со своим окружением.

Модель окружения системы и модель использования системы представляют собой две дополняющие друг друга модели взаимоотношений между данной системой и ее окружением.

1. Модель окружения системы – это статическая модель, которая описывает другие системы из окружения разрабатываемого ПО.

2. Модель использования системы – динамическая модель, которая показывает взаимодействие данной системы со своим окружением.

Модель окружения системы можно представить с помощью схемы связей (см. рис. 6.4), которая дает простую блок-схему общей архитектуры системы. С помощью *пакетов* языка UML ее можно представить в развернутом виде как совокупность подсистем (см. рис. 6.6). Такое представление показывает, что рабочее окружение системы Метеостанция находится внутри подсистемы, занимающейся сбором данных. Там же показаны другие подсистемы, которые образуют систему построения карт погоды.

При моделировании взаимодействия системы с ее окружением применяется абстрактный подход, который не требует больших объемов данных для описания этих взаимодействий. Подход, предлагаемый UML, состоит в том, чтобы разработать модель вариантов использования, в которой каждый вариант представляет собой определенное взаимодействие с системой. В модели вариантов использования каждое возможное взаимодействие изображается в виде эллипса, а внешняя сущность, включенная во взаимодействие, представлена стилизованной фигуркой человека. В нашем примере внешняя сущность, хотя и представлена фигуркой человека, является системой обработки метеорологических данных.

Модель вариантов использования для метеостанции показана на рис. 6.7. В этой модели метеостанция взаимодействует с внешними объектами во время запуска и завершения работы, при составлении отчетов на основе собранных данных, а также при тестировании и калибровке метеорологических приборов.

Каждый из имеющихся вариантов использования можно описать с помощью простого естественного языка. Такое описание помогает

разработчикам проекта идентифицировать объекты в системе и понять, что система должна делать. Здесь используют стилизованную форму описания, которая четко определяет, как происходит обмен информацией, как инициируется взаимодействие и т.д. Эта форма описания показана в табл. 6.1, где представлен вариант использования **Отчет** (см. рис. 6.7).

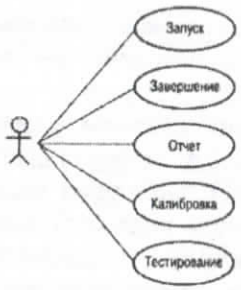

Рис. 6.7. Варианты использования метеостанции

Таблица 6.1. Описание варианта использования Отчет

Система	**Метеостанция**
Вариант использования	Отчет
Участники	Система сбора метеорологических данных, метеостанция
Данные	Метеостанция отправляет сводку с данными, снятыми с различных приборов в определенный временной период системой сбора метеорологических данных. В сообщении содержатся максимальные, минимальные и средние значения температуры почвы и воздуха, атмосферного давления, скорости ветра, общее количество выпавших осадков и направление ветра, взятые через пятиминутные интервалы времени
Входные сигналы	Система сбора метеорологических данных устанавливает модемную связь с метеостанцией и отправляет запрос на передачу данных

Ответ	Итоговые данные отправляются в систему сбора метеорологических данных
Комментарии	Обычно от метеостанций запрашивают отчет каждый час, но эта частота запросов может отличаться для разных станций, а также может измениться в будущее

Конечно, для описания вариантов использования можно прибегнуть к любой другой методике при условии, что предложенное описание краткое и понятное. Как правило, требуется разработать описания для всех вариантов использования, имеющихся в данной модели.

Описание вариантов использования помогает идентифицировать объекты и операции в системе. Из описания варианта использования Отчет видно, что в системе должны быть объекты, представляющие приборы для сбора метеорологических данных, а также объекты, предоставляющие итоговые метеорологические данные. Должны также быть операции, формирующие запрос, и операции, пересылающие метеорологические данные.

6.4. Проектирование архитектуры

Когда взаимодействия между проектируемой системой ПО и ее окружением определены, эти данные можно использовать как основу для разработки архитектуры системы. Конечно, при этом необходимо применять знания об общих принципах проектирования системных архитектур и данные о конкретной предметной области.

Автоматизированная метеостанция является относительно простой системой, поэтому ее архитектуру можно вновь представить как многоуровневую модель. На рис. 6.8 внутри большого прямоугольника **Метеостанция** расположены три прямоугольника UML. Здесь использовали систему нотации UML (текст в прямоугольниках с загнутыми углами) с тем, чтобы представить дополнительную информацию.

Рис. 6.8. Архитектура метеостанции

В программном обеспечении метеостанции можно выделить три уровня.

1. Уровень интерфейсов, который занимается всеми взаимодействиями с другими частями системы и предоставлением внешних интерфейсов системы.
2. Уровень сбора данных, управляющий сбором данных с приборов и обобщающий метеорологические данные перед отправкой их в систему построения карт погоды.
3. Уровень приборов, в котором представлены все приборы, используемые в процессе сбора исходных метеорологических данных.

В общем случае следует попытаться разложить систему на части так, чтобы архитектура была как можно проще. Согласно хорошему практическому правилу, модель архитектуры должна состоять не более чем из семи основных объектов. Каждый такой объект можно описать отдельно, однако для того, чтобы отобразить структуру этих объектов и их взаимосвязи, можно воспользоваться схемой, подобной показанной на рис. 6.6.

6.5. Определение объектов

Перед выполнением данного этапа проектирования уже должны быть сформированы представления относительно основных объектов проектируемой системы. В системе метеостанции, очевидно, что приборы являются объектами и требуется, по крайней мере один объект на каждом уровне архитектуры. Это проявление основного принципа, согласно которому объекты обычно появляются в процессе проектирования. Вместе с тем требуется определить и документировать все другие объекты системы.

Хотя этот раздел назван "Определение *объектов*", на самом деле на данном этапе проектирования определяются *классы* объектов. Структура системы описывается в терминах этих классов. Классы объектов, определенные ранее, неизбежно получают более детальное описание, поэтому иногда приходится возвращаться на данный этап проектирования для переопределения классов.

Существует множество подходов к определению классов объектов.

1. Использование грамматического анализа естественного языкового описания системы. Объекты и атрибуты – это существительные, операции и сервисы – глаголы. Такой подход реализован в иерархическом методе объектно-ориентированного проектирования, который широко используется в аэрокосмической промышленности Европы.
2. Использование в качестве объектов ПО событий, объектов и ситуаций реального мира из области приложения, например самолетов, ролевых ситуаций менеджера, взаимодействий, подобных интерактивному общению на научных конференциях и т.д.. Для реализации таких объектов могут потребоваться специальные структуры хранения данных (абстрактные структуры данных).
3. Применение подхода, при котором разработчик сначала полностью определяет поведение системы. Затем определяются компоненты системы, отвечающие за различные поведенческие акты (режимы работы системы), при этом основное внимание уделяется тому, кто инициирует и кто осуществляет данные режимы. Компоненты системы, отвечающие за основные режимы работы, считаются объектами.
4. Применение подхода, основанного на сценариях, в котором по очереди определяются и анализируются различные сценарии использования системы. Поскольку анализируется каждый сценарий, группа, отвечающая за анализ, должна идентифицировать необходимые объекты, атрибуты и операции. Метод анализа, при котором аналитики и разработчики присваивают роли объектам, показывает эффективность подхода, основанного на сценариях.

Каждый из описанных подходов помогает начать процесс определения объектов. Но для описания объектов и классов объектов необходимо использовать информацию, полученную из разных

3. Что такое окружение системы и какие модели ее использования вы знаете?
4. Какие подходы к определению классов объектов используют?

Ключевые слова: *объекты, объектно-ориентированный анализ, объектно-ориентированное проектирование, объектно-ориентированное программирование, объект, серверы, активные объекты, модель окружения системы, модель использования системы.*

Keywords: *objects, object-oriented analysis, object-oriented designing, object-oriented programming, object, servers, active objects, model of the encirclement of the system, model of the use the system.*

Kalit so'zlar: *obyektlar, obyektga yo'naltirilgan tahlil, obyektga yo'naltirilgan loyihalashtirish, obyektga yo'naltirilgan dasturlash, serverlar, faol obyektlar, tizim muhiti modeli, tizimdan foydalanish modeli.*

Упражнения

1. Объясните, почему в проектировании систем применение подхода, который полагается на слабо связанные объекты, скрывающие информацию о своем представлении, приводит к созданию системной архитектуры, которую затем можно легко модифицировать.
2. Покажите на примерах разницу между объектом и классом объектов.
3. При каких условиях можно разрабатывать систему, в которой объекты выполняются параллельно?
4. С помощью графической системы нотации UML спроектируйте следующие классы объектов с определенными атрибутами и операциями:
   - телефон;
   - принтер персонального компьютера;
   - персональная стереосистема;
   - банковские расчеты;
   - каталог библиотеки.
5. Разработайте более детальный проект метеостанции, добавив описания интерфейсов объектов. Они могут быть записаны с помощью языков Java, C++ или UML.

6. Разработайте проект метеостанции, показывающий взаимодействие между подсистемой сбора данных и приборами, собирающими данные. Воспользуйтесь диаграммой последовательностей.
7. Определите возможные объекты в следующих системах, применяя при этом объектно-ориентированный подход.
   - Система "Дневник группы" поддерживает расписание собраний и встреч в группе сотрудников. Для организации встречи, в которой участвует группа людей, система находит общие для всех личных дневников свободные "окна" и назначает эту встречу на определенное время. Если система не находит общих "окон", то начинает взаимодействовать с пользователями, чтобы реорганизовать личные дневники и тем самым создать "окно" для встречи.
   - Установлена полностью автоматизированная бензоколонка. Водитель вставляет кредитную карточку в считывающее устройство, связанное с насосом; карточка по линиям коммуникаций проверяется кредитной компанией, устанавливается требуемое количество бензина. Затем автомобиль заправляется горючим. Когда подача прекращается, с кредитной карточки водителя снимается стоимость полученного бензина. Кредитная карточка возвращается после вычета водителю. Если карточка неверна, она возвращается водителю перед подачей топлива.

ГЛАВА 7. ПРОЕКТИРОВАНИЕ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

В настоящее время компьютеры применяются для управления широким спектром разнообразных систем, начиная от простых домашних устройств и заканчивая крупными промышленными комплексами. Эти компьютеры непосредственно взаимодействуют с аппаратными устройствами. Программное обеспечение таких систем (управляющий компьютер плюс управляемые объекты) представляет собой встроенную систему реального времени, задача которой – реагировать на события, генерируемые оборудованием, т.е. в ответ на эти события вырабатывать управляющие сигналы. Такое ПО **встраивается** в большие аппаратные системы и должно обеспечивать реакцию на события, происходящие в окружении системы, в режиме **реального времени.**

Системы реального времени отличаются от других типов программных систем. Их корректное функционирование зависит от способности системы реагировать на события через заданный (как правило, короткий) интервал времени. Вот как определяют систему реального времени.

Система реального времени – это программная система, правильное функционирование которой зависит от результатов ее работы и от периода времени, в течение которого получен результат. "Мягкая" система реального времени – это система, в которой операции **удаляются**, если в течение определенного интервала времени не выдан результат. "Жесткая" система реального времени – это система, операции которой становятся **некорректными**, т.е. вырабатывается сигнал об ошибке, если в течение определенного интервала времени результат не выдан.

Систему реального времени можно рассматривать как систему "стимул-отклик". При получении определенного входного стимула (входного сигнала) система генерирует связанный с ним отклик (ответное действие или ответный сигнал). Следовательно, поведение системы реального времени можно определить с помощью списка входных сигналов, получаемых системой, связанных с ними ответных сигналов (откликов) и интервала времени, в течение которого система должна отреагировать на входной сигнал.

Входные сигналы делятся на два класса.

1. **Периодические сигналы** происходят через предопределенные интервалы времени. Например, система проверяет датчик каждые 50 миллисекунд, и предпринимает действия (реагирует) в зависимости от значений, полученных от датчика (стимула).

2. **Апериодические сигналы** происходят нерегулярно. Обычно они "сообщают о себе" посредством механизма прерываний. Примером апериодического сигнала может быть прерывание, которое вырабатывается по завершении передачи вход/выход и размещения данных в буфере обмена.

В системах реального времени периодические входные сигналы обычно генерируются сенсорами (датчиками), взаимодействующими с системой. Они предоставляют информацию о состоянии внешнего окружения системы. Системные отклики (ответные сигналы) направляются группе исполнительных механизмов, управляющих аппаратными устройствами, которые затем воздействуют на окружение системы. Апериодические входные сигналы могут генерироваться и сенсорами, и исполнительными механизмами. Как правило, апериодические сигналы означают исключительные ситуации, например ошибки в работе аппаратуры. На рис. 7.1 показана модель "сенсор-система-исполнительный механизм" для встроенной системы реального времени.

Рис. 7.1. Общая модель системы реального времени

Системы реального времени должны реагировать на входные сигналы, происходящие в разные моменты времени. Следовательно, архитектуру такой системы необходимо организовать так, чтобы управление переходило к соответствующему обработчику как можно быстрее после получения входного сигнала. В последовательных программах такой механизм передачи управления невозможен. Поэтому обычно системы реального времени проектируют как

множество параллельных взаимодействующих процессов. Часть системы - управляющая программа, часто называемая диспетчером, - управляет всеми процессами.

Большинство моделей "стимул-отклик" систем реального времени сводятся к обобщенной архитектурной модели, состоящей из трех типов процессов (рис. 7.2). Для каждого типа сенсора имеется процесс управления сенсором; вычислительный процесс определяет необходимый ответный сигнал на полученный системой входной сигнал; процессы управления исполнительными механизмами управляют действиями этих механизмов. Такая модель позволяет быстро собрать данные со всех имеющихся сенсоров (до того, как произойдет следующий ввод данных), обработать их и получить ответный сигнал от соответствующего исполнительного механизма.

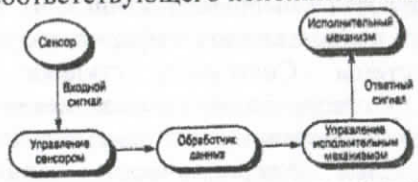

Рис. 7.2. Процессы управления сенсорами и механизмами

7.1. Проектирование систем

Как указывалось в главе 1, в процессе проектирования системы принимаются решения о том, какие свойства будут реализованы программной частью системы, а какие - аппаратными средствами. Временные ограничения и другие требования предполагают, что некоторые функции системы, например, обработку сигналов, необходимо реализовать на специально разработанном оборудовании. Таким образом, процесс проектирования систем реального времени включает в себя проектирование оборудования (аппаратуры) специального назначения и проектирование программного обеспечения.

Аппаратные компоненты обеспечивают более высокую производительность, чем эквивалентное им (по выполняемым функциям) программное обеспечение. Аппаратным средствам можно поручить "узкие" места системной обработки сигналов и, таким образом, избежать дорогостоящей оптимизации ПО. Если за производительность системы отвечают аппаратные компоненты, при

проектировании ПО основное внимание можно уделить его переносимости, а вопросы, связанные с производительностью, отходят на второй план.

Решения о распределении функций по аппаратным и программным компонентам следует принимать как можно позже, поскольку архитектура системы должна состоять из автономных компонентов, которые можно реализовать как аппаратно, так и программно. Именно такая структура соответствует целям разработчика, проектирующего удобную в обслуживании систему. Следовательно, результатом процесса проектирования высококачественной системы должна быть система, которую можно реализовать и аппаратными и программными средствами.

Отличие процесса проектирования систем реального времени от других систем состоит в том, что уже на первых этапах проектирования необходимо учитывать время реакции системы. В центре процесса проектирования системы реального времени – события (входные сигналы), а не объекты или функции. Процесс проектирования таких систем состоит из нескольких этапов.

1. Определение множества входных сигналов, которые будут обрабатываться системой, и соответствующих им системных реакций, т.е. ответных сигналов.
2. Для каждого входного сигнала и соответствующего ему ответного сигнала вычисляются временные ограничения. Они применяются к обработке как входных, так и ответных сигналов.
3. Объединение процессов обработки входных и ответных сигналов в виде совокупности параллельных процессов. В корректной модели системной архитектуры каждый процесс связан с определенным классом входных и ответных сигналов (как показано на рис. 7.2).
4. Разработка алгоритмов, выполняющих необходимые вычисления для всех входных и ответных сигналов. Чтобы получить представление об объемах вычислительных и временных затрат в процессе обработки сигналов, разработка алгоритмов обычно проводится на ранних этапах процесса проектирования.
5. Разработка временного графика работы системы.
6. Сборка системы, работающей под управлением диспетчера – управляющей программы.

Конечно, описанный процесс проектирования является итерационным. Как только определена структура вычислительных процессов и временной график работы, необходимо сделать всесторонний анализ и провести имитацию работы системы, чтобы удостовериться в том, что она удовлетворяет временным ограничениям. В результате анализа может обнаружиться, что система не отвечает временным требованиям. В таком случае для повышения производительности системы необходимо изменить структуру вычислительных процессов, алгоритм управления, управляющую программу или все эти компоненты вместе.

В системах реального времени сложно анализировать временные зависимости. Из-за непредсказуемой природы апериодических входных сигналов разработчики вынуждены делать некоторые предварительные предположения относительно вероятности появлений апериодических сигналов. Сделанные предположения могут оказаться неверными, и после разработки системы ее показатели производительности не будут удовлетворять временным требованиям. В работах обсуждаются общие проблемы проверки временных параметров систем. В книге всесторонне рассмотрены методы, используемые при анализе производительности систем реального времени.

Все процессы в системе реального времени должны быть скоординированы. Механизм координации процессов обеспечивает исключение конфликтов при использовании общих ресурсов. Когда один процесс использует общий ресурс (или объект), другие процессы не должны иметь доступ к этому ресурсу. К механизмам, обеспечивающим взаимное исключение процессов, относятся семафоры, мониторы и метод критических областей. Здесь не будут рассматривать эти механизмы, так как все они хорошо документированы в описаниях операционных систем.

7.2. Моделирование систем реального времени

Системы реального времени должны реагировать на события, происходящие через нерегулярные интервалы времени. Такие события (или входные сигналы) часто приводят к переходу системы из одного состояния в другое. Поэтому одним из способов описания

систем реального времени может быть модель конечного автомата и соответствующая диаграмма состояний, рассмотренные в главе 4.

В модели конечного автомата в каждый момент времени система находится в одном из своих состояний. Получив входной сигнал, она переходит в другое состояние. Например, система управления клапаном может перейти из состояния "Клапан открыт" в состояние "Клапан закрыт" после получения определенной команды оператора (входной сигнал).

Описанный выше подход к моделированию системы проиллюстрируют на рассмотренном в главе 4 примере микроволновой печи. На рис. 7.3 показана модель конечного автомата для обычной микроволновой печи, оборудованной кнопками включения питания, таймера и запуска системы. Состояния системы обозначены скругленными прямоугольниками, входные сигналы, вызывающие переход системы из одного состояния в другое, показаны стрелками. На диаграмме показаны все состояния печи, также названы действия исполнительных механизмов системы или действия по выводу информации.

Просматривать последовательность работы системы нужно слева направо. В начальном состоянии **Ожидание**, пользователь может выбрать режим полной или половинной мощности. Следующее состояние наступает при нажатии на кнопку таймера и установке времени работы печи. Если дверь печи закрыта, система переходит в состояние **Действие**. В этом состоянии идет процесс приготовления пищи, после завершения которого печь возвращается в состояние **Ожидание**.

Модели конечного автомата – хороший способ представления структуры систем реального времени. Поэтому такие модели являются неотъемлемой частью методов проектирования систем реального времени. Метод Харела (Harel), базирующийся на **диаграммах состояний,** направлен на решение проблемы внутренней сложности моделей конечного автомата. Диаграмма состояний структурирует модели таким образом, что группы состояния можно было бы рассматривать как единые сущности. Кроме того, с помощью диаграмм состояний параллельные системы можно представить в виде модели состояний. Модели состояний

поддерживаются также UML. В этой книге используют систему нотации, принятую в UML.

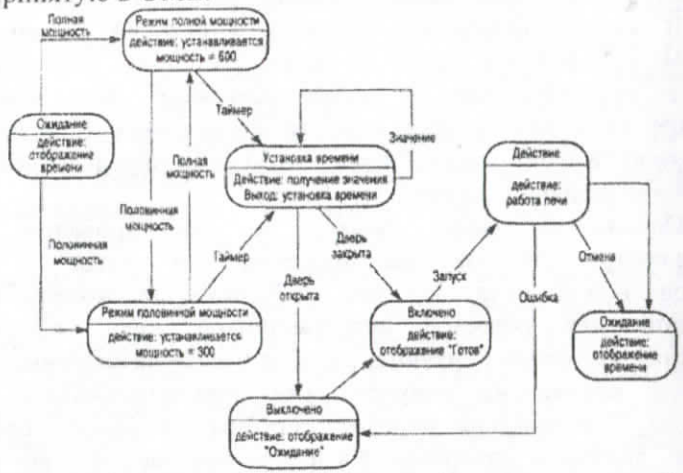

Рис. 7.3. Модель конечного автомата для микроволновой печи

7.3. Программирование систем реального времени

На архитектуру системы реального времени оказывает влияние язык программирования, который используется для реализации системы. До сих пор жесткие системы реального времени часто программируются на ассемблерных языках. Языки более высокого уровня также дают возможность сгенерировать эффективный программный код. Например, язык С позволяет писать весьма эффективные программы. Однако в нем нет конструкций, поддерживающих параллельность процессов и управление совместно используемыми ресурсами. Кроме того, программы на С часто сложны для понимания.

Язык Ada изначально разрабатывался для реализации встроенных систем, а потому располагает такими средствами, как управление процессами, исключения и правила представления. Его средство *рандеву* (rendezvous) - отличный механизм для синхронизации задач (процессов). К сожалению, первая версия языка Ada (Ada 83) оказалась непригодной для реализации жестких систем реального времени. В ней отсутствовали средства, позволяющие установить предельные сроки завершения задач, не было встроенных

исключений для случая превышения предельных сроков и предлагался строгий алгоритм обслуживания очереди "первым пришел - первым вышел". При пересмотре стандартов языка Ada главное внимание уделялось именно этим моментам. В пересмотренной версии языка поддерживаются защищенные типы, что позволило более просто реализовывать защищенные разделяемые структуры данных и обеспечивать более полный контроль при выполнении и синхронизации задач. Однако при программировании систем реального времени улучшенная версия языка Ada все-таки не обеспечивает достаточного контроля над жесткими системами реального времени.

Первые версии языка Java разрабатывались для создания небольших встраиваемых систем, таких, например, как контроллеры устройств и приборов. Разработчики Java включили несколько средств для поддержки параллельных процессов в виде параллельных объектов (потоков) и синхронизированных методов. Но поскольку в подобных системах нет строгих временных ограничений, то и в языке Java не предусмотрены средства, позволяющие управлять планированием потоков или запускать потоки в конкретные моменты времени.

Поэтому Java не подходит для программирования жестких систем реального времени или систем, в которых имеется строгий временной график процессов. Перечислим основные проблемы Java как языка программирования систем реального времени.

1. Нельзя указать время, в течение которого должен выполняться поток.
2. Неконтролируем процесс очистки памяти – он может начаться в любое время. Поэтому невозможно предсказать поведение потоков во времени.
3. Нельзя определить размеры очереди, связанной с разделяемыми ресурсами.
4. Реализация виртуальной машины Java отличается для разных компьютеров.
5. В языке нет средств для детального анализа распределения времени работы процессоров.

В настоящее время ведется работа по решению некоторых из этих проблем и формируется новая версия языка Java для

программирования систем реального времени. Однако не совсем понятно, каким образом эту версию можно отделить от лежащей в ее основе виртуальной машины Java: свойство переносимости языка всегда конфликтовало с характеристиками режима реального времени.

7.4. Управляющие программы

Управляющая программа (диспетчер) системы реального времени является аналогом операционной системы компьютера. Она управляет процессами и распределением ресурсов в системах реального времени, запускает и останавливает соответствующие процессы для обработки входных сигналов и распределяет ресурсы памяти и процессора. Однако обычно в управляющих программах отсутствуют более сложные средства, присущие операционным системам, например средства управления файлами.

В работе представлен полный обзор средств, необходимых управляющим программам систем реального времени. Данная тема обсуждается в монографии, где также кратко рассмотрены коммерческие разработки управляющих программ для систем реального времени. Несмотря на то что на рынке программных продуктов существует несколько управляющих программ систем реального времени, их часто проектируют самостоятельно как части систем из-за специальных требований, предъявляемых к конкретным системам реального времени.

Компоненты управляющей программы (рис. 10.4) зависят от размеров и сложности проектируемой системы реального времени. Обычно управляющие программы, за исключением самых простых, состоят из следующих компонентов:

1. **Часы реального времени** периодически предоставляют информацию для планирования процессов.
2. **Обработчик прерываний управляет** апериодическими запросами к сервисам.
3. **Планировщик** просматривает список процессов, которые назначены на выполнение, и выбирает один из них.
4. **Администратор ресурсов**, получив процесс, запланированный на выполнение, выделяет необходимые ресурсы памяти и процессора.

5. **Диспетчер** запускает на выполнение какой-либо процесс.

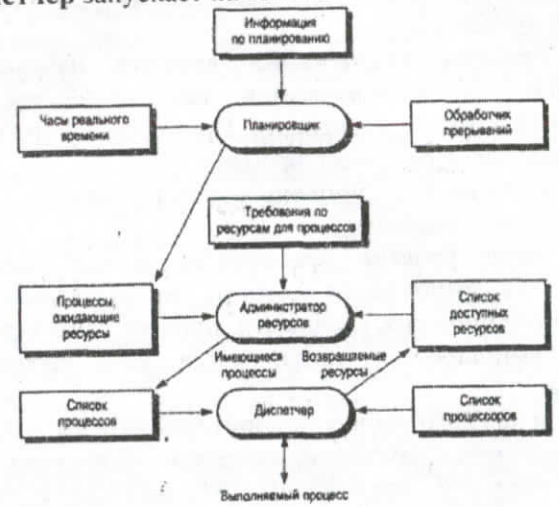

Рис. 7.4. Компоненты управляющей программы реального времени

Управляющие программы систем, предоставляющих сервисы на постоянной основе, например, телекоммуникационных или мониторинговых систем с высокими требованиями к надежности, могут иметь еще несколько компонентов:

- **Конфигуратор** отвечает за динамическое переконфигурирование аппаратных средств. Не прекращая работу системы, из нее можно извлечь аппаратные модули и изменить систему посредством добавления новых аппаратных средств.
- **Менеджер неисправностей** отвечает за обнаружение аппаратных и программных неисправностей и предпринимает соответствующие действия по их исправлению.

Входные сигналы, обрабатываемые системой реального времени, обычно имеют несколько уровней приоритетов. Для одних сигналов, например связанных с исключительными ситуациями, важно, чтобы их обработка завершалась в течение определенного интервала времени. Если процесс с более высоким приоритетом запрашивает сервис, то выполнение других процессов должно быть приостановлено. Вследствие этого администратор системы должен

уметь управлять, по крайней мере, двумя уровнями приоритетов системных процессов.

1. **Уровень прерываний** является наивысшим уровнем приоритетов. Он присваивается тем процессам, на которые необходимо быстро отреагировать. Примером такого процесса может быть процесс часов реального времени.

2. **Тактовый уровень** приоритетов присваивается периодическим процессам.

Еще один уровень приоритетов может быть у фоновых процессов, на выполнение которых не накладываются жесткие временные ограничения, (например, процесс самотестирования). Эти процессы выполняются тогда, когда есть свободные ресурсы процессора.

Внутри каждого уровня приоритетов разным классам процессов можно назначить другие приоритеты. Например, может быть несколько уровней прерываний. Во избежание потери данных прерывание от более быстрого устройства должно вытеснять обработку прерываний от более медленного устройства.

7.5. Управление процессами

Управление процессами – это выбор процесса на выполнение, выделение для него ресурсов памяти и процессора и запуск процесса.

Периодическими называются процессы, которые должны выполняться через фиксированный предопределенный промежуток времени (например, при сборе данных или управлении исполнительными механизмами). Управляющая программа системы реального времени для определения момента запуска процесса использует свои часы реального времени. В большинстве систем реального времени есть несколько классов периодических процессов с разными периодами (интервалами времени между выполнением процессов) и длительностью выполнения. Управляющая программа должна быть способна в любой момент времени выбрать процесс, назначенный на выполнение.

Часы реального времени конфигурируются так, чтобы периодически подавать тактовый сигнал, период между сигналами составляет обычно несколько миллисекунд. Сигнал часов инициирует процесс на уровне прерываний, который запускает планировщик

процессов для управления периодическими процессами. Процесс на уровне прерываний обычно сам не управляет периодическими процессами, поскольку обработка прерываний должна завершаться как можно быстрее.

Действия, выполняемые управляющей программой при управлении периодическими процессами, показаны на рис. 7.5. Планировщик просматривает список периодических процессов и выбирает из него на выполнение один процесс. Выбор зависит от приоритета процесса, периода процесса, предполагаемой длительности выполнения и конечных сроков завершения процесса. Иногда за один период между тактовыми сигналами часов необходимо выполнить два процесса с разными длительностями выполнения. В такой ситуации один процесс необходимо приостановить на время, соответствующее его длительности.

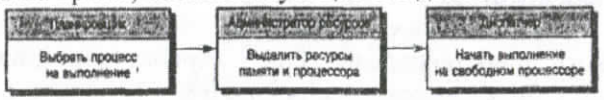

Рис. 7.5. Действия управляющей программы при запуске процесса

Если управляющей программой зарегистрировано прерывание, это означает, что к одному из сервисов сделан запрос. Механизм прерываний передает управление предопределенной ячейке памяти, в которой содержится команда переключения на программу обслуживания прерываний. Эта программа должна быть простой, короткой и быстро выполняться. Во время обслуживания прерываний все другие прерывания системой игнорируются. Чтобы уменьшить вероятность потери данных, время пребывания системы в таком состоянии должно быть минимальным.

Программа, выполняющая сервисную функцию, должна перекрыть доступ следующим прерываниям, чтобы не прервать саму себя. Она должна выявить причину прерывания и инициировать процесс с высоким приоритетом для обработки сигнала, вызвавшего прерывание. В некоторых системах высокоскоростного сбора данных обработчик прерываний сохраняет для последующей обработки данные, которые в момент получения прерывания находились в буфере. После обработки прерывания управление вновь переходит к управляющей программе.

В любой момент времени может быть несколько назначенных на выполнение процессов с разными уровнями приоритетов. Планировщик устанавливает порядок выполнения процессов. Эффективное планирование играет важную роль, если необходимо соответствовать требованиям, которые предъявляются к системе реального времени. Существует две основные стратегии планирования процессов.

1. **Не вытесняющее планирования**. Один процесс планируется на выполнение, он запускается и выполняется до конца или блокируется по каким-либо причинам, например при ожидании ввода данных. При таком планировании могут возникнуть проблемы, связанные с тем, что в случае нескольких процессов с разными приоритетами процесс с высоким приоритетом должен ждать завершения процесса с низким приоритетом.

2. **Вытесняющее планирование**. Выполнение процесса может быть приостановлено, если к сервису поступили запросы от процессов с более высоким приоритетом. Процесс с более высоким приоритетом имеет преимущество перед процессом с более низким уровнем приоритета, и поэтому ему выделяется процессор.

В рамках этих стратегий разработано множество различных алгоритмов планирования. К ним относится циклическое планирование, при котором каждый процесс выполняется по очереди, и планирование по скорости, когда при первом выполнении получают более высокий приоритет процессы с коротким периодом выполнения. Каждый из алгоритмов планирования имеет определенные преимущества и недостатки, однако здесь мы их рассматривать не будем.

Информация о назначенном на выполнение процессе передается администратору ресурсов. Он выделяет для выбранного процесса необходимую память, а в многопроцессорной системе – еще и процессор. Затем процесс помещается в "список назначений", т.е. в список процессов, назначенных на выполнение. Когда процессор завершает выполнение какого-либо процесса и становится свободным, вызывается диспетчер. Он просматривает имеющийся список, выбирает процесс, который можно выполнять на свободном процессоре, и запускает его на выполнение.

7.6. Системы наблюдения и управления

В настоящее время можно выделить несколько классов стандартных систем реального времени: мониторинговые системы (системы наблюдения), системы сбора данных, системы управления и др. Каждому типу систем соответствует особая структура процессов, поэтому при проектировании системы, как правило, архитектуру создают по одному из существующих стандартных типов. Таким образом, вместо обсуждения общих проблем проектирования систем реального времени здесь лучше рассмотреть проектирование с помощью обобщенных моделей.

Системы наблюдения и управления – важный класс систем реального времени. Их основным назначением является проверка сенсоров (датчиков), предоставляющих информацию об окружении системы, и выполнение соответствующих действий в зависимости от поступившей от сенсоров информации. Системы наблюдения выполняют действия после регистрации особого значения сенсора. Системы управления непрерывно управляют аппаратными исполнительными механизмами на основании значений, получаемых от сенсоров.

Рассмотрим следующий пример.

Пусть в здании установлена система охранной сигнализации. В системе используется несколько типов сенсоров: датчики движения, установленные в отдельных комнатах; датчики на окнах первого этажа, которые подают сигнал, если разбивается окно; дверные датчики, фиксирующие открывание дверей. Всего в системе 50 датчиков на окнах, 30 на дверях и 200 датчиков движения.

Когда какой-либо датчик фиксирует присутствие постороннего, система автоматически вызывает местную полицию и, используя звуковой синтезатор, сообщает местоположение датчика (номер комнаты), от которого идет сигнал. В комнатах, расположенных возле активного датчика, включается световая сигнализация и звуковой аварийный сигнал. Система сигнализации обычно включается через сеть, но может работать и от батарей. Проблемы с электропитанием регистрируются специальной программой, контролирующей напряжение электросети. Если в сети регистрируется падение

напряжения, программа переключает систему сигнализации на резервное питание от батарей.

В этом примере описана "мягкая" система реального времени, так как здесь нет жестких временных требований. В такой системе не *нужно* регистрировать события, происходящие с высокой скоростью, поэтому опрос датчиков может проводиться два раза в секунду.

Процесс проектирования начинается с описания апериодических входных сигналов, получаемых системой, и связанных с ними реакций системы. Здесь мы ограничимся упрощенным проектом, в котором не учитываются сигналы, порождаемые процедурами самопроверки, и внешние сигналы, генерируемые при тестировании системы или при ее выключении в случае ложной тревоги. В конечном счете система обрабатывает только два типа входных сигналов.

1. **Отключение электропитания** генерируется программой, контролирующей электрическую цепь. В ответ на этот сигнал система переключает сеть на резервное питание посредством подачи сигнала электронному прибору переключения питания.
2. **Сигнал о вторжении** является входным и генерируется одним из датчиков системы. В ответ на него система определяет номер комнаты, в которой находится активный датчик, вызывает полицию, инициируя звуковой синтезатор, и включает звуковой сигнал тревоги и световую сигнализацию здания в месте нарушения.

На следующем шаге процесса проектирования определяются временные ограничения для каждого входного и ответного сигналов системы. В табл. 7.1 перечислены эти временные ограничения. К разным типам датчиков, генерирующих входные сигналы, предъявлены разные временные требования.

На следующем этапе проектирования распределяются системные функции по параллельным процессам. Периодически нужно опрашивать три типа датчиков, поэтому у каждого типа датчиков имеется связанный с ними процесс. Кроме этого, есть система управления прерываниями, контролирующая электропитание, система связи с полицией, синтезатор речи, система включения звуковой сигнализации и система, включающая световую сигнализацию возле датчика. Каждой системой управляет

независимый процесс. На рис. 7.6 показана архитектура процессов системы.

Таблица 7.1. Временные ограничения на входные и ответные сигналы системы

Сигнал	Временные ограничения
Отключение электропитания	Переключение на питание от батарей должно произойти в течение 50 мс
Сигнализация на двери	Каждый сигнальный датчик на дверях проверяется дважды в секунду
Сигнализация на окнах	Каждый датчик на окне проверяется дважды в секунду
Датчик движения	Каждый датчик движения опрашивается дважды в секунду
Звуковой сигнал	Звуковой сигнал должен прозвучать через полсекунды после сигнала датчика
Включение световой сигнализации	Световая сигнализация должна включиться через полсекунды после сигнала датчика
Связь	Вызов в полицию должен начаться в течение 2 с после сигнала датчика
Синтезатор речи	Синтезированное сообщение должно быть готово через 4 с после сигнала датчика

На схеме, представленной на рис. 7.6, стрелки (с примечаниями), соединяющие отдельные процессы, обозначают потоки данных между процессами с указанием типа данных. Надписи над стрелками справа над каждым процессом указывают систему, управляющую данным процессом. На стрелках вверху указано минимальное значение частоты выполнения процесса.

Частота выполнения процессов определяется количеством датчиков и временными требованиями, предъявляемыми системе. Предположим, что в системе имеется 30 дверных датчиков, которые требуется проверять два раза в секунду. Следовательно, связанный с дверным датчиком процесс должен выполняться 60 раз в секунду (частота 60 Гц). Также 400 раз в секунду выполняется процесс, контролирующий датчик движения.

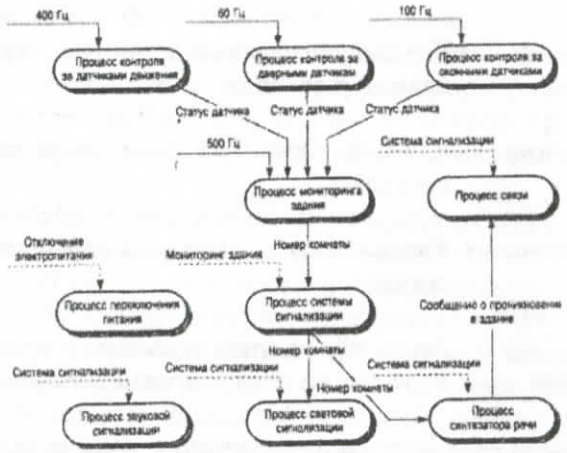

Рис. 7.6. Архитектура процессов системы охранной сигнализации

Апериодические процессы обозначены стрелками с пунктирными линиями. На этих линиях указаны события, которые вызывают данный процесс. Все основные исполнительные процессы (звуковой и световой сигнализации и др.) начинаются командой из процесса **Система безопасности**; им не нужны данные из других процессов. Процессу, управляющему электропитанием, также не нужны данные из других частей системы.

Все представленные процессы можно реализовать на языке Java как потоки. Листинг 7.1 содержит код Java реализации процесса **BuildingMonitor** (мониторинг здания), опрашивающего датчики системы. В случае сигнала тревоги программа активизирует систему сигнализации. Пусть в нашем примере система соответствует временным требованиям. Как уже отмечалось, в языке Java 2.0 нет средств для задания частоты выполнения потоков.

Листинг 7.1. Реализация процесса мониторинга здания

```
class BuildingMonitor extends Thread {
BuildingSensor win, door, move;
Siren siren = new SirenO;
Lights lights = new Lights{};
Synthesizer synthesizer = new Synthesizer();
DoorSensors doors = new DoorSensors (30);
WindowSensors windows = new WindowSensors(50);
MovementSensors movements = new MovementSensors(200);
PowerMonitor pm = new PowerMonitor();

BuildingMonitor()
{
//инициализация датчиков и запуск процессов
siren.start();lights.start();
synthesizer.start();windows.start();
doors.start()/movements.start();pm.start();
}
public void run ()
{
int room = 0;
while (true)
{
//проверка датчиков движения два раза в секунду (400 Гц) move
=
movements.getVal();
//проверка оконных датчиков два раза в секунду (100 Гц) win =
windows.getVal();
//проверка дверных датчиков два раза в секунду (60 Гц) door =
doors.getVal();
if(move.sensorVal==l|door.sensorVal==l|win.sensorVal==l)
{
//датчик зарегистрировал нарушение
if(move.sensorVal == 1) room = move.room;
if(door.sensorVal == 1) room = door.room;
if(win.sensorVal == 1) room = win.room;
```

```
lights.on(room);siren.on();synthesizer.on(room);
break;
}
}
lights.shutdown() ; siren.shutdown();synthesizer.shutdown();
windows.shutdown();doors.shutdown();movements.shutdown();
}//run
)//BuildingMonitor
```

Поскольку данная система не содержит строгих временных требований, ее можно реализовать на языке Java. Конечно, в Java 2.0 нет никакой гарантии соответствия временным спецификациям. В нашей системе все датчики опрашиваются одинаковое количество раз, чего не бывает в реальных системах.

После определения архитектуры процессов системы начинается разработка алгоритмов обработки входных сигналов и генерации ответных сигналов. Этап проектирования необходимо выполнять как можно раньше, чтобы удостовериться, что система будет соответствовать временным требованиям. Если соответствующие алгоритмы оказываются сложными, может возникнуть необходимость в изменении временных ограничений. Обычно алгоритмы систем реального времени сравнительно просты. Они проверяют ячейки памяти, выполняют некоторые простые расчеты и управляют передачей сигнала. В примере системы сигнализации проектирование алгоритмов не рассматривается.

Последним этапом процесса проектирования является составление временного графика выполнения процессов. В нашем примере нет процессов со строгими сроками выполнения. Рассмотрим приоритеты процессов. Все процессы, опрашивающие датчики, имеют один и тот же приоритет. Процессу, управляющему электропитанием, необходимо назначить более высокий приоритет. Приоритеты процессов, управляющих системой сигнализации, и процессов, опрашивающих датчики, должны быть одинаковы.

Систему охранной сигнализации можно отнести скорее к системам наблюдения, чем к системам управления, так как в ней нет исполнительных механизмов, напрямую зависящих от значений датчиков. Примером системы управления может служить система управления отоплением здания. Система наблюдает за

температурными датчиками, установленными в разных комнатах здания и переключает нагревательные приборы в зависимости от реальной температуры и температуры, установленной в термореле. Термореле, в свою очередь, контролирует отопительный котел.

Архитектура процессов такой системы показана на рис. 7.7; в общем виде она выглядит подобно системе сигнализации. Дальнейшее рассмотрение этого примера предлагается читателю в качестве упражнения.

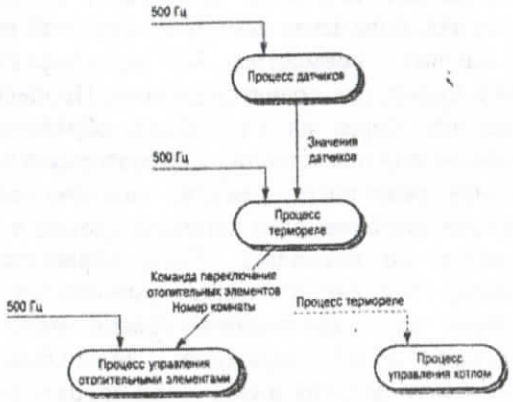

Рис. 7.7. Архитектура процессов системы управления отоплением

7.7.Системы сбора данных

Эти системы представляют другой класс систем реального времени, которые обычно базируются на обобщенной архитектурной модели. Такие системы собирают данные с сенсоров в целях их последующей обработки и анализа.

Для иллюстрации этого класса систем рассмотрим модель, представленную на рис. 7.8. Здесь изображена система, собирающая данные с датчиков, которые измеряют поток нейтронов в ядерном реакторе. Данные, собранные с разных датчиков, помещаются в буфер, из которого затем извлекаются и обрабатываются. На мониторе оператора отображается среднее значение интенсивности потока нейтронов.

Рис. 7.8. Архитектура системы наблюдения за интенсивностью потока нейтронов

Каждый датчик связан с процессом, который преобразует аналоговый сигнал, показывающий интенсивность входного потока, в цифровой. Сигнал совместно с идентификатором датчика записывается в буфер, где хранятся данные. Процесс, отвечающий за обработку данных, берет их из буфера, обрабатывает и передает процессу отображения для вывода на операторную консоль.

В системах реального времени, ведущих сбор и обработку данных, скорости выполнения и периоды процесса сбора и процесса обработки могут не совпадать. Если обрабатываются большие объемы данных, сбор данных может выполнятся быстрее, чем их обработка. Если же выполняются только простые вычисления, быстрее происходит обработка данных, а не их сбор.

Чтобы сгладить разницу в скоростях сбора и обработки данных, в большинстве подобных систем для хранения входных данных используется кольцевой буфер. Процессы, создающие данные (процессы-производители), поставляют информацию в буфер. Процессы, обрабатывающие данные (процессы-потребители), берут данные из буфера (рис. 7.9).

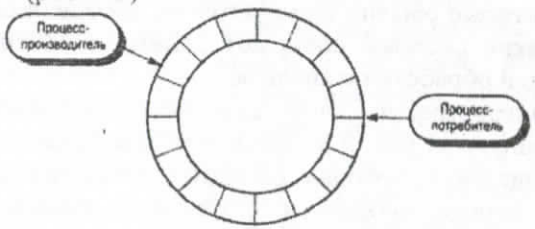

Рис. 7.9. Кольцевой буфер в системе сбора данных

Очевидно, необходимо предотвратить одновременный доступ процесса-производителя и процесса-потребителя к одним и тем же элементам буфера. Кроме того, система должна отслеживать, чтобы процесс-производитель не добавлял данные в полный буфер, а процесс-потребитель не забирал данные из пустого буфера.

В листинге 7.2 показана возможная реализация буфера данных как объекта Java. Значения в буфере имеют тип SensorRecord (запись данных датчика). Определены два метода– get и put: метод get берет элементы из буфера, метод put добавляет элемент в буфер. При объявлении типа CircularBuffer (кольцевой буфер) программный конструктор задает размер буфера.

Листинг 7.2. Реализация кольцевого буфера

```
class CircularBuffer
{
int bufsize;
SensorRecord [] std're;
int numberOfEntries = 0;
int front = 0, back = 0;

CircularBuffer (int n) {
bufsize = n;
store = new SensorRecord [bufsize] ;
}//CircularBuffer

synchronized void put (SensorRecord rec) throws
InterruptedException
{
if(numberOfEntries == bufsize)
waitt) ;
store [back] = new SensorRecord (rec.sensorId,rec.sensorVal);
back = back + 1;
if(back == bufsize)
back = 0;
numberOfEntries = numberOfEntries + 1;
notify();
}//put
synchronized SensorRecord get() throws InterruptedException
{
SensorRecord result = new SensorRecord(-1,-1);
if(numberOfEntries == 0)
wait () ;
```

```
result = store [front];
front = front + 1;
if(front == bufsize)
front = 0;
numberOfEntries = numberOfEntries - 1;
notify();
return result;
}// get
}//CircularBuffer
```

Модификатор **synchronized**, связанный с методами **get** и **put**, указывает на то, что данные методы не должны выполняться параллельно. При вызове одного из этих методов система реального времени блокирует экземпляр объекта, чтобы в это же время не произошел вызов другого метода и соответственно не производились манипуляции на том же участке буфера. Вызовы методов **wait** и **notify** из методов **get** и **put** гарантируют, что входные данные нельзя положить в полный буфер или взять из пустого буфера. Метод **wait** вызывает поток и приостанавливается, пока другой поток с помощью метода **notify** не отправит ему сообщение о снятии ожидания. При вызове метода **wait** блокировка на защищенные данные объекта снимается. Метод **notify** возобновляет выполнение одного из ожидающих потоков.

Контрольные вопросы:

1) Что такое система реального времени?
2) Этапы процесса проектирования системы реального времени.
3) Основные проблемы Java как языка программирования систем реального времени.
4) Из каких компонентов состоят управляющие программы?
5) Укажите основные стратегии планирования процессов.

Ключевые слова: *система реального времени, периодические сигналы, апериодические сигналы, модели конечного автомата, часы реального времени, обработчик прерываний управляет, планировщик, администратор ресурсов, диспетчер, конфигуратор, менеджер неисправностей, уровень прерываний, тактовый уровень, управление процессами, невытесняющее планирования, вытесняющее планирование, система безопасности.*

Key words: *real-time, periodic signals, aperiodic signals, a finite state machine model, real-time clock interrupt handler manages scheduling, resource manager, manager, configurator, manager fault interrupt level, the clock level, process management, non-preemptive scheduling, preemptive planning, security system.*

Kalit so'zlar: real vaqt tizimi, davriy signallar, nodavriy signallar, yakuniy avtomat modellari, real vaqt soatlari, boshqaruv uzulishlariga ishlov berish, rejalashtiruvchi, manbalar administratori, dispetcher, konfigurator, shikastlanganlik menedjeri, uzulishlar darajasi, takt darajasi, jarayonlarni boshqarish, siqib chiqarmaslik rejalashtiruvi, siqib chiqarish rejalashtiruvi, havfsizlik tizimi.

Упражнения

1. Почему системы реального времени обычно реализованы как множество параллельных процессов? Проиллюстрируйте свой ответ примерами.
2. Объясните, почему объектно-ориентированные методы разработки ПО не всегда подходят к системам реального времени.
3. Нарисуйте диаграммы состояний управляющего ПО для следующих систем.
   - Автоматическая стиральная машина с разными программами для разных типов белья.
   - Программное обеспечение для проигрывателя компакт-дисков.
   - Телефонный автоответчик, который записывает входящие сообщения и отображает количество полученных сообщений на жидкокристаллическом экране. Система должна определить телефон звонившего, вывести на экран последовательность чисел (идентифицированных как тоновый набор) и хранить записанные сообщения, которые затем можно прослушать.
   - Автомат по выдаче напитков, который может налить кофе с молоком и сахаром или без них. Пользователь бросает монету и с помощью нажатия кнопок на автомате выбирает нужный режим. Автомат выдает чашку с растворимым кофе. Пользователь затем ставит чашку под кран, нажимает другую кнопку и автомат наливает в чашку горячую воду.

4. Используя методы проектирования систем реального времени спроектируйте заново систему сбора данных от метеостанций, в виде системы "стимул-ответ".
5. Спроектируйте архитектуру процессов для системы наблюдения, собирающей данные с группы датчиков, измеряющих состав воздуха и расположенных вокруг города. В системе 5000 датчиков, организованных в группы по 100 штук. Каждый датчик должен проверяться 4 раза в секунду. Если более 30% датчиков в группе зафиксируют, что качество воздуха ниже допустимого уровня, активизируется предупреждающий световой сигнал. Все датчики передают собранные данные центральному компьютеру, который каждые 15 мин генерирует отчет о составе воздуха в городе.
6. Обсудите сильные и слабые стороны Java как языка программирования для реализации систем реального времени.
7. Система безопасности поезда автоматически закрывает двери, если скорость поезда превышает предельную для данного участка трассы или если при выходе на участок пути горит красный свет (т.е. въезд на участок запрещен). Остальные подробности перечислены во врезке 13.1. Идентифицируйте входные сигналы, которые должна обрабатывать бортовая система управления поездом, и связанные с ними ответные сигналы.
8. Предположите вероятную архитектуру процессов такой системы.
9. Если в бортовой системе безопасности поезда при сборе данных с путевых передатчиков используются периодические процессы, какую частоту сбора данных следует запланировать, чтобы система гарантированно получала информацию от передатчиков? Обоснуйте свой ответ.

ГЛАВА 8. УПРАВЛЕНИЕ КОНФИГУРАЦИЯМИ

Управление конфигурацией – это процесс разработки и применения стандартов и правил по управлению эволюцией программных продуктов. Эволюционирующие системы нуждаются в управлении по той простой причине, что в процессе их эволюции создается несколько версий одних и тех же программ. В эти версии обязательно вносятся некоторые изменения, исправляются ошибки предыдущих версий; кроме того, версии могут адаптироваться к новым аппаратным средствам и операционным системам. При этом в разработке и эксплуатации могут одновременно находиться сразу несколько версий. Поэтому нужно четко отслеживать все вносимые в систему изменения.

Процедуры управления конфигурацией регулируют процессы регистрации и внесения изменений в систему с указанием измененных компонентов, а также способы идентификации различных версий системы. Средства управления конфигурацией применяют для хранения всех версий системных компонентов, для компоновки из этих компонентов системы и для отслеживания поставки заказчикам разных версий системы.

Поэтому иногда одно и то же лицо может отвечать как за управление качеством, так и за управление конфигурацией. Но обычно разрабатываемая программная система сначала контролируется командой по управлению качеством, которая проверяет ПО на соответствие определенным стандартам качества. Далее ПО передается команде по управлению конфигурацией, которая контролирует изменения, вносимые в систему.

Существует много причин, объясняющих наличие разных конфигураций одной и той же системы. Различные версии создаются для разных компьютеров или операционных систем, включающих специальные функции, нужные заказчикам, и т.д. (рис. 8.1). Менеджеры по управлению конфигурацией обязаны следить за различиями между разными версиями, чтобы обеспечить возможность выпуска следующих вариантов системы и своевременную поставку нужных версий соответствующим заказчикам.

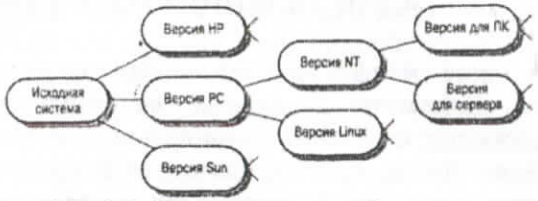

Рис. 8.1. Семейство версий системы

Процесс управления конфигурацией и связанная с ним документация должны подчиняться определенным стандартам. В качестве примера можно привести стандарт IEEE 828-1983, определяющий составление планов управления конфигурацией. Каждая организация должна иметь справочник, в котором указаны эти стандарты, либо они должны входить в общий справочник стандартов качества. Общенациональные или международные стандарты могут быть также использованы как основа для разработки детализированных специальных норм и стандартов для конкретных организаций. За основу можно взять любой тип стандарта, поскольку все они содержат описания однотипных процессов. Для сертификации качества своих программных продуктов организация должна придерживаться официальных стандартов управления конфигурацией, которые приведены в стандартах ISO 9000 и в модели оценки уровня развития SEI.

При традиционной разработке ПО в соответствии с каскадной моделью разрабатываемая система попадает в группу по управлению конфигурацией уже после полного завершения разработки и тестирования ПО. Именно такой подход лежит в основе стандартов управления конфигурацией, которые, в свою очередь, обусловливают необходимость использования для разработки систем моделей, подобных каскадной. Поэтому упомянутые стандарты не в полной мере подходят при использовании таких методов разработки ПО, как эволюционное прототипирование и пошаговая разработка. В этой ситуации некоторые организации изменили подход к управлению конфигурацией, сделав возможным параллельную разработку и тестирование системы. Такой подход основан на регулярной (иногда ежедневной) сборке системы из ее компонентов.

1. Устанавливается время, к которому должна быть завершена поставка компонентов системы (например, к 14.00).

Программисты, работающие над новыми версиями компонентов, должны предоставить их к указанному времени. Работу над компонентами не обязательно завершать, достаточно представить основные рабочие функции для проведения тестирования.

2. Создается новая версия системы с новыми компонентами, которые компилируются и связываются в единую систему.

3. После этого система попадает к группе тестирования. В то же время разработчики продолжают работу над компонентами, добавляя новые функции и исправляя ошибки, обнаруженные в ходе предыдущего тестирования.

4. Дефекты, замеченные при тестировании, регистрируются, соответствующий документ пересылается разработчикам. В следующей версии компонента эти дефекты будут учтены и исправлены.

Основным преимуществом ежедневной сборки системы является возможность выявления ошибок во взаимодействиях между компонентами, которые в противном случае могут накапливаться. Более того, ежедневная сборка системы поощряет тщательную проверку компонентов. Разработчики работают под давлением: нельзя прерывать сборку систем и поставлять неисправные версии компонентов. Поэтому программисты неохотно поставляют новые версии компонентов, если они не были предварительно тщательно проверены. Таким образом, на тестирование и исправление ошибок ПО уходит меньше времени.

Для ежедневных сборок системы требуется достаточно строгое управление процессом изменений, позволяющее отслеживать проблемы, которые выявляются и исправляются в ходе тестирования. Кроме того, в результате возникает множество версий компонентов системы, для управления которыми необходимы средства управления конфигурацией.

8.1. Планирование управления конфигурацией

В плане управления конфигурацией представлены стандарты, процедуры и мероприятия, необходимые для управления. Отправной точкой создания такого плана является набор общих стандартов по управлению конфигурацией, применяемых в организации-разработчике ПО, которые адаптируются к каждому отдельному

проекту. Обычно план управления конфигурацией имеет несколько разделов.

1. Определение контролируемых объектов, подпадающих под управление конфигурацией, а также формальная схема определения этих объектов.
2. Перечень лиц, ответственных за управление конфигурацией и за поставку контролируемых объектов в команду по управлению конфигурацией.
3. Политика ведения управления конфигурацией, т.е. процедуры управления изменениями и версиями.
4. Описание форм записей о самом процессе управления конфигурацией.
5. Описание средств поддержки процесса управления конфигурацией и способов их использования.
6. Определение базы данных конфигураций, применяемой для хранения всей информации о конфигурациях системы.

Распределение обязанностей по конкретным исполнителям является важной частью плана. Необходимо четко определить ответственных за поставку каждого документа или компонента ПО для команд по управлению качеством и конфигурацией. Лицо, отвечающее за поставку какого-либо документа или компонента, должно отвечать и за их разработку. Для упрощения процедур согласования удобно назначать менеджеров проекта или ведущих специалистов команды разработчиков ответственными за все документы, созданные под их руководством.

8.2. Определение конфигурационных объектов

В процессе разработки больших систем создаются тысячи различных документов. Большинство из них – это текущие рабочие документы, связанные с различными этапами разработки ПО. Есть также внутренние записки, протоколы заседания рабочих групп, проекты планов и предложений и т.п. Такие документы представляют разве что исторический интерес и не нужны для дальнейшего сопровождения системы.

Для планирования процесса управления конфигурацией необходимо точно определить, какие проектные элементы (или классы элементов) будут объектами управления. Такие элементы

называются **конфигурационными элементами**. Как правило, они представляют собой официальные документы. Конфигурационными элементами обычно являются планы проектов, спецификации, схемы системной архитектуры, программы и наборы тестовых данных. Кроме того, управлению подлежат все документы, необходимые для будущего сопровождения системы.

В процессе управления конфигурацией каждому документу необходимо присвоить уникальное имя, причем отображающее связи с другими документами. Для этого используется иерархическая система имен, где они имеют, например, такой вид:

PLC-TOOLS/ПРАВКА/ФОРМЫ/ОТОБРАЖЕНИЕ/ИНТЕРФЕЙСЫ/КОД

PLC-TOOLS/ПРАВКА/СПРАВКА/ЗАПРОС/ОКНО_СПРАВКИ/FR-1

Начальная часть имени – это название проекта PLC-TOOLS. В проекте разрабатываются четыре отдельных средства (рис. 8.2). Имя средства используется в следующей части имени. Каждое средство создается из именованных модулей. Такое разбиение продолжается до тех пор, пока не появится ссылка на официальный документ базового уровня. Листья дерева иерархии документов являются официальными документами проекта. На рис. 8.2 показано, что для каждого объекта требуется три формальных документа. Это описание объектов (документ ОБЪЕКТЫ), код компонента (документ КОД) и набор тестов для этого кода (документ ТЕСТЫ).

Рис. 8.2. Иерархия конфигурации

Подобные схемы имен основаны на структуре проекта, когда имена соотносятся с соответствующими проектными компонентами.

Такой подход к именованию документов порождает определенные проблемы. Например, снижается возможность повторного использования компонентов. Обычно в таких случаях из схемы берутся копии компонентов, которые можно повторно использовать, и переименовываются в соответствии с новой областью применения. Другие проблемы могут появиться, если эта схема именования документов используется как основа структуры хранения компонентов. Тогда пользователь должен знать названия документов, чтобы найти нужные компоненты, при этом не все документы одного типа (например, по проектированию) хранятся в одном месте. Также могут встретиться трудности при установлении соответствия между схемой имен и схемой идентификации, используемой в системе управления версиями.

8.3. База данных конфигураций

Такая база данных используется для хранения всей информации о системных конфигурациях. Основными функциями базы данных конфигураций являются поддержка оценивания влияния планируемых изменений в системе и предоставление информации о процессе управления конфигурацией. Задание структуры базы данных конфигураций, определение процедур записи и поиска информации в этой базе данных – все это является частью процесса планирования управления конфигурацией.

Информация, заключенная в базе данных конфигураций, должна помочь ответить на ряд вопросов, среди которых основными и часто запрашиваемыми будут следующие:

- Каким заказчикам поставлена определенная версия системы?
- Какие аппаратные средства и какая операционная система необходимы для работы данной версии системы?
- Сколько было выпущено версий данной системы и когда?
- На какие версии системы повлияют изменения, вносимые в определенный компонент?
- Сколько запросов на изменения было реализовано в данной версии?
- Какое количество ошибок было зарегистрировано в данной версии системы?

В идеале база данных конфигураций должна быть объединена с системой управления версиями, которая создается для хранения и управления формальными проектными документами. Такой подход, к тому же поддерживаемый некоторыми интегрированными CASE-средствами, предоставляет возможность связать изменения, вносимые в систему, и с документами, и с теми компонентами, которые подверглись изменениям. В этом случае упрощается поиск измененных компонентов, *поскольку* установлены связи между документами (например, между документами по системной архитектуре и кодом программ) и этими связями можно управлять.

Однако многие организации вместо использования интегрированных CASE-средств для управления конфигурацией рассматривают базу данных конфигураций как отдельную систему. Конфигурационные элементы могут храниться в отдельных файлах или в системе управления версиями, например RCS (известная система управления версиями для Unix). В этом случае в базе данных конфигураций хранится информация о конфигурационных элементах и ссылки на имена соответствующих файлов в системе управления версиями. Несмотря на относительную дешевизну и гибкость такого подхода, основным недостатком его является то, что конфигурационные элементы могут быть изменены без внесения необходимых записей в базу данных. Поэтому нельзя гарантировать, что в базе данных конфигураций содержится обновленная и корректная информация о состоянии системы.

8.4. Управление изменениями

Изменения в больших программных системах неизбежны. Как отмечалось в предыдущих главах, в течение жизненного цикла системы изменяются пользовательские и системные требования, а также приоритеты и запросы организаций. Процесс управления изменениями и соответствующие CASE-средства предназначены для того, чтобы зарегистрировать изменения и внести их в систему наиболее эффективным способом.

Процесс управления изменениями (листинг 8.1) начинается после того, как программное обеспечение или соответствующая документация передается команде по управлению конфигурацией. Он может начаться во время тестирования системы или даже после ее

поставки заказчику. Процедуры управления изменениями создаются для обеспечения корректного анализа необходимости изменений и их стоимости, а также для контроля за вносимыми изменения.

Листинг 8.1. Процесс управления изменениями

```
Запрос на изменение, заполнение формы запроса
Анализ запроса
if изменение допустимо then
Оценка способа внесения изменения
Оценка стоимости изменения
Запись запроса в базу данных
Передача запроса группе контроля за изменениями
if запрос принят then
repeat
внесение изменений в ПО
регистрация изменений
передача измененного ПО группе управления качеством
until качество ПО соответствует нормам
создание ноной версии системы
else
запрос на изменение отвергнут
else
запрос на изменение отвергнут
```

Первым этапом в процессе управления изменениями является заполнение формы запроса на изменения, в которой указываются те изменения, которые планируется внести в систему. В форме запроса также приводятся рекомендации относительно изменений, предварительная оценка затрат и даты запроса, его утверждения, внедрения и проверки. Также форма может включать раздел, в котором указывается способ выполнения изменения. Запросы на изменения регистрируются в базе данных конфигураций. Таким образом, команда управления конфигурациями может следить за выполнением изменений, а также контролировать изменения определенных программных компонентов.

Во врезке 8.1 приведен пример заполненной формы запроса на изменения. Такая форма обычно определяется на этапе планирования управления конфигурацией. По условиям некоторых контрактов

(например, в контрактах с правительственными органами), такая форма должна соответствовать стандартам заказчика.

Врезка 8.1. Форма запроса на изменения

Проект. Proteus/PCL-Tools **Номер.** 23/94
Лицо, заполняющее запрос. Соммервиль Дата. 1/12/98

Требуемые изменения. После выбора компонента структуры необходимо отображение имени файла, в котором он хранится.

Лицо, осуществляющее анализ запроса. Г. Дин

Дата анализа. 10/12/98

Изменяемые компоненты. Display-Icon.Select, Display-Icon.Display

Связанные компоненты. FileTable

Оценка изменения. Изменение достаточно легко выполнить из-за наличия таблицы имен файлов. Требуется вставить соответствующее поле. Изменений связанных компонентов не требуется.

Уровень приоритета. Низкий

Выполнение изменения.

Оценка затрат. 0,5 дня

Дата передачи в группу контроля за изменениями. 15/12/98

Дата принятия решения группой контроля за изменениями. 1/2/99

Решение группы контроля за изменениями. Принять изменение. Будет реализовано в версии 2.1.

Кто вносит изменения.

Дата внесения изменений.

Дата передачи запроса в группу управления качеством.

Решение группы управления качеством.

Дата передачи запроса в группу по управлению конфигурацией.

Примечание.

Сразу после представления заполненной формы запроса проводится проверка необходимости и допустимости изменения. Это объясняется тем, что некоторые изменения вызваны не ошибками в программе, а неправильным пониманием требований, другие могут

дублировать исправление ранее обнаруженных ошибок. Если в процессе проверки выявляется, что изменение недопустимо, повторяется или уже было рассмотрено, то изменение отклоняется. Лицу, представившему запрос на изменение, объясняется причина отказа.

Для принятых изменений начинается вторая стадия – оценка изменений и предварительное определение стоимости. Сначала следует проверить влияние изменения на всю систему. Для этого делается технический анализ способа внесения изменения. Затем определяется стоимость внесения изменения в определенные компоненты, что регистрируется в форме запроса. В процессе оценивания полезна база данных конфигураций с информацией о взаимосвязях между компонентами, благодаря чему есть возможность оценить влияние изменений на другие компоненты системы.

Все изменения, кроме тех, которые относятся к исправлению мелких недоработок, должны быть переданы в группу контроля за изменениями, где принимается решение о принятии изменения либо отказе. Эта группа оценивает воздействие изменения не с технической, а скорее с организационной или стратегической точек зрения. Во внимание принимаются такие соображения, как экономическая выгодность изменения и организационные факторы, которые оправдывают необходимость изменения.

Группа контроля за изменениями состоит из лиц, на которых возлагается ответственность за решения о внесении изменений. Такие группы со структурой, включающей старшего менеджера компании-заказчика и сотрудников фирмы-разработчика, обязательны при выполнении военных проектов. Для небольших или среднего размера проектов в эту группу может входить только менеджер проекта и один-два инженера, которые не занимались разработкой данного ПО. В отдельных случаях допускается участие аналитика по изменениям, который дает рекомендации относительно того, оправданы эти изменения либо нет.

После принятия решения о внесении изменений программная система для внесения изменений передается разработчикам или команде по сопровождению системы. По окончании этой процедуры система обязательно должна пройти проверку на правильность

внесения изменений. После этого именно команда по управлению конфигурацией, а не разработчики, займется выпуском новой версии.

Изменение каждого компонента системы должно регистрироваться. Таким образом, создается история компонента. Самый лучший способ для этого – создавать стандартизированные комментарии в начале кода компонента, где содержатся ссылки на запросы изменений данного компонента. Для составления отчетов об изменениях компонента и обработки их историй используются специальные средства.

8.5. Управление версиями и выпусками

Управление версиями и выпусками ПО необходимо для идентификации и слежения за всеми версиями и выпусками системы. Менеджеры, отвечающие за управление версиями и выпусками ПО, разрабатывают процедуры поиска нужных версий системы и следят за тем, чтобы изменения не осуществлялись произвольно. Они также работают с заказчиками и планируют время выпуска следующих версий системы. Над новыми версиями системы должна работать команда по управлению конфигурацией, а не разработчики, даже если новые версии предназначены только для внутреннего использования. Только в том случае, если информация об изменениях в версиях вносится исключительно командой по управлению конфигурацией, можно гарантировать согласованность версий.

Версией системы называют экземпляр системы, имеющий определенные отличия от других экземпляров этой же системы. Новые версии могут отличаться функциональными возможностями, эффективностью или исправлениями ошибок. Некоторые версии имеют одинаковую функциональность, однако разработаны под различные конфигурации аппаратного или программного обеспечения. Если отличия между версиями незначительны, они называются **вариантами** одной версии.

Выходная версия (release) системы – это та версия, которая поставляется заказчику. В каждой выходной версии либо обязательно присутствуют новые функциональные возможности, либо она разработана под новую платформу. Количество версий обычно намного превышает количество выходных версий, поскольку версии

создаются в основном для внутреннего пользования и не поставляются заказчику.

В настоящее время для поддержки управления версиями разработано много разнообразных CASE-средств (см. раздел 8.5). С помощью этих средств осуществляется управление хранением каждой версии и контроль за допуском к компонентам системы. Компоненты могут извлекаться из системы для внесения в них изменений. После введения в систему измененных компонентов получается новая версия, для которой с помощью системы управления версиями создается новое имя.

8.6. Идентификация версий

Любая большая программная система состоит из сотен компонентов, каждый из которых может иметь несколько версий. Процедуры управления версиями должны четко идентифицировать каждую версию компонента. Существует три основных способа идентификации версий.

1. **Нумерация версий**. Каждый компонент имеет уникальный и явный номер версии. Эта схема идентификации используется наиболее широко.
2. **Идентификация, основанная на знамениях атрибутов**. Каждый компонент идентифицируется именем, которое, однако, не является уникальным для разных версий, и набором значений атрибутов, разных для каждой версии компонента. Здесь версия компонента идентифицируется комбинацией имени и набора значений атрибутов.
3. **Идентификация на основе изменений**. Каждая версия системы именуется так же, как в способе идентификации, основанном на значениях атрибутов, плюс ссылки на запросы на изменения, которые реализованы в данной версии системы. Таким образом, версия системы идентифицируется именем и теми изменениями, которые реализованы в системных компонентах.

8.7. Нумерация версий

По самой простой схеме нумерации версий к имени компонента или системы добавляется номер версии. Например, Solaris 2.6 обозначает версию 2.6 системы Solaris. Первая версия обычно

обозначается 1.0, последующими версиями будут 1.1, 1.2 и т.д. На каком-то этапе создается новая выходная версия – версия 2.0, нумерация этой версии начинается заново – 2.1, 2.2 и т.д. Эта линейная схема нумерации основана на предположении о последовательности создания версий. Подобный подход к идентификации версий поддерживается многими программными средствами управления версиями, например RCS (см. раздел 8.5).

На рис. 8.3 графически проиллюстрирован описанный способ нумерации версий. Стрелки на рисунке проведены от исходной версии к новой, которая создается на ее основе. Отметим, что последовательность версий не обязательно линейная – версии с последовательными номерами могут создаваться на основе разных базовых версий. Например, на рис. 8.3 видно, что версия 2.2 создана на основе версии 1.2, а не версии 2.1. В принципе каждая существующая версия может служить основой для создания новой версии системы.

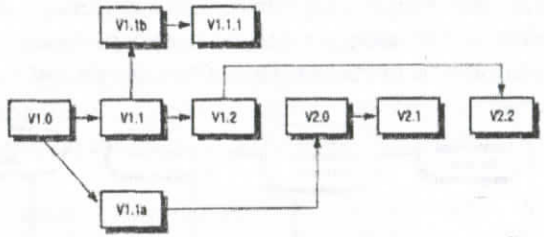

Рис. 8.3. Структура системных версий

Данная схема идентификации версий достаточно проста, однако она требует довольно большого количества информации для сопоставления версий, что позволяло бы отслеживать различия между версиями и связи между запросами на изменения и версиями. Поэтому поиск отдельной версии системы или компонента может быть достаточно трудным, особенно при отсутствии интеграции между базой данных конфигураций и системой хранения версий.

8.8. Сборка системы

Сборкой системы называют процесс компиляции и связывания программных компонентов в единую исполняемую программу. Перед сборкой системы полезно ответить на следующие вопросы.

1. Все ли компоненты, составляющие систему, включены в инструкцию по сборке?
2. Каковы версии компонента, перечисленные в инструкции по сборке?
3. Доступны ли все необходимые файлы данных?
4. Если на файлы данных используются ссылки внутри компонентов, то каковы имена этих файлов в выходной версии?
5. Доступны ли нужные версии компилятора и других необходимых средств? Действующие версии программных средств могут быть несовместимы с более старыми версиями, которые применялись при разработке системы.

В настоящее время существует много средств управления конфигурацией, автоматизирующих процесс сборки системы. Команда управления конфигурацией пишет сценарий, в котором определены зависимости между различными компонентами системы. В нем также указаны средства компилирования и связывания компонентов системы. Средства компоновки интерпретируют сценарий сборки системы и вызывают программы, необходимые для сборки исполняемой системы. Процесс сборки системы представлен на рис. 8.4.

Рис. 8.4. Сборка системы

В сценарии сборки указаны зависимости между компонентами, поэтому компоновщик системы сам принимает решение, когда перекомпилировать компоненты, а когда можно многократно использовать существующий объектный код. Зависимости в сценарии сборки указаны в основном как зависимости между файлами, содержащими исходный код компонентов. Однако, если файлов с исходным кодом разных версий много, возникает проблема выбора нужных файлов. Проблема усугубляется, если файлы исходного и объектного кода имеют одинаковые имена (но, конечно, с разными расширениями).

Чтобы избежать трудностей, связанных с зависимостью физических файлов, было разработано несколько экспериментальных систем, основанных на языках описания модулей. В них используется описание логической структуры ПО и схемы зависимостей между файлами, содержащими компоненты исходного кода. Такой подход снижает количество ошибок и приводит к более понятным описаниям процесса сборки системы.

Контрольные вопросы:

1) Дайте определение управлению конфигурацией.
2) Перечислите разделы плана управления конфигурацией.
3) Что такое начальная часть имени?
4) Что называют версией?
5) Какие три основных способа идентификации версий существует?

Ключевые слова: управление конфигурацией конфигурационные элементы; начальная часть имени; версия; выходная версия; нумерация версий; идентификация, основанная на знамениях атрибутов; идентификация на основе изменений.

Keywords: configuration management configuration items; the initial part of the name; version; output version; numbering versions; identification based on the signs of attributes; identification on the basis of the changes.

Kalit so'zlar: konfiguratsiyani boshqarish, konfiguratsion elementlar, nomning bosh qismi, yangi naql, atributlar nomiga asoslangan identifikatsiya, o'zgarishlarga asoslangan identifikatsiya.

Упражнения

1. Объясните, почему в системе управления конфигурацией для идентификации документов не используются названия документов. Предложите схему идентификации документов, которую можно было бы использовать во всех проектах вашей организации.
2. Используя модель "сущность-связь" или объектно-ориентированный подход, разработайте модель базы данных конфигураций, которая должна содержать информацию о системных компонентах, их версиях, выходных версиях системы и об изменениях, реализованных в системе. База данных должна обладать следующими функциями:

- извлечение всех версий или отдельной указанной версии компонента;
- извлечение последней по времени изменения версии компонента;
- поиск запросов на изменения, которые были реализованы в указанной версии системы;
- определение версий компонентов, включенных в указанную версию системы;
- извлечение выходной версии системы, определяемой по дате выпуска или по имени заказчика, которому она поставлена.

3. С помощью диаграммы потока данных представьте модель управления изменениями, которую можно было бы применить в большой организации, занимающейся разработкой ПО для внешних заказчиков. Запросы на изменения могут поступать как от внешних, так и от внутренних источников.
4. Опишите трудности, которые могут встретиться при сборке системы. В частности, рассмотрите проблемы сборки системы на хост-компьютере.
5. Со ссылкой на систему сборки объясните, почему иногда необходимо сохранять устаревшие компьютеры, на которых разрабатывались большие программные системы.
6. При сборке систем часто возникает сложная проблема, состоящая в том, что имена физических файлов встроены в системный код и используемая файловая структура отличается от файловой структуры на конечной машине, где будет установлена система. Составьте руководство для программистов, которое поможет избежать этой и подобных проблем при сборке систем.
7. Приведите пять факторов, которые необходимо учитывать при сборке выходных версий больших программных систем.
8. Опишите два способа оптимизации процесса сборки системы из ее компонентов с помощью соответствующих CASE-средств компоновки систем.

ГЛАВА 9. ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Тестирования начинается с тестирования отдельных программных модулей, например процедур и объектов. Затем модули компонуются в подсистемы и потом в систему, при этом проводится тестирование взаимодействий между модулями. Наконец, после сборки системы, заказчик может провести серию приемочных тестов, во время которых проверяется соответствие системы ее спецификации.

На рис. 9.1 показана схема двухэтапного процесса тестирования. На этапе покомпонентного тестирования проверяются отдельные компоненты. Это могут быть функции, наборы методов, собранные в один модуль, или объекты. На этапе тестирования сборки эти компоненты интегрируются в подсистемы или законченную систему. На этом этапе основное внимание уделяется тестированию взаимодействий между компонентами, а также показателям функциональности и производительности системы как единого целого. Но, конечно, на этапе тестирования сборки также могут обнаруживаться ошибки в отдельных компонентах, не замеченные на этапе покомпонентного тестирования.

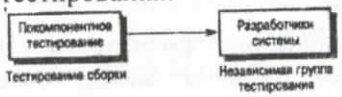

Рис. 9.1. Этапы тестирования ПО

При планировании процесса верификации и аттестации ПО менеджеры проекта должны определить, кто будет отвечать за разные этапы тестирования. Во многих случаях за тестирование своих программ (модулей или объектов) несут ответственность программисты. За следующий этап отвечает группа системной интеграции (сборки), которая интегрирует отдельные программные модули (возможно, полученные от разных разработчиков) в единую систему и тестирует эту систему в целом.

Для критических систем процесс тестирования должен быть более формальным. Такая формализация предполагает, что за все этапы тестирования отвечают независимые испытатели, все тесты разрабатываются отдельно и во время тестирования ведутся подробные записи. Чтобы протестировать критические системы,

независимая группа разрабатывает тесты, исходя из спецификации каждого системного компонента.

При разработке некритических, "обычных" систем подробные спецификации для каждого системного компонента, как правило, не создаются. Определяются только интерфейсы компонентов, причем за проектирование, разработку и тестирование этих компонентов несут ответственность отдельные программисты или группы программистов. Таким образом, тестирование компонентов, как правило, основывается только на понимании разработчиками того, что должен делать компонент.

Тестирование сборки должно основываться на имеющейся спецификации системы. При составлении плана тестирования обычно используется спецификация системных требований или спецификация пользовательских требований. Тестированием сборки всегда занимается независимая группа.

Во многих книгах, посвященных тестированию программного обеспечения, например, описывается процесс тестирования программных систем, реализующих функциональную модель ПО, но не рассматривается отдельно тестирование объектно-ориентированных систем. В контексте тестирования между объектно-ориентированными и функционально-ориентированными системами имеется ряд отличий.

1. В функционально-ориентированных системах существует четко определенное различие между основными программными элементами (функциями) и совокупностью этих элементов (модулями). В объектно-ориентированных системах этого нет. Объекты могут быть простыми элементами, например списком, или сложными, например такими, как объект метеорологической станции из главы 9, состоящий из ряда других объектов.

2. В объектно-ориентированных системах, как правило, нет такой четкой иерархии объектов, как в функционально-ориентированных системах. Поэтому такие методы интеграции систем, как нисходящая или восходящая сборка (см. раздел 9.2), часто не подходят для объектно-ориентированных систем.

Таким образом, в объектно-ориентированных системах между тестированием компонентов и тестированием сборки нет четких границ. В таких системах процесс тестирования является

продолжением процесса разработки, где основной системной структурой являются объекты. Несмотря на то что большая часть методов тестирования подходит для систем любых видов, для тестирования объектно-ориентированных систем необходимы специальные методы. Такие методы рассмотрены в разделе 9.3.

9.1. Тестирование дефектов

Целью тестирования дефектов является выявление в программной системе скрытых дефектов до того, как она будет сдана заказчику. Тестирование дефектов противоположно аттестации, в ходе которой проверяется соответствие системы своей спецификации. Во время аттестации система должна корректно работать со всеми заданными тестовыми данными. При тестировании дефектов запускается такой тест, который вызывает **некорректную** работу программы и, следовательно, выявляет дефект. Обратите внимание на эту важную особенность: тестирование дефектов демонстрирует **наличие**, а не отсутствие дефектов в программе.

Общая модель процесса тестирования дефектов показана на рис. 9.2. **Тестовые сценарии** – это спецификации входных тестовых данных и ожидаемых выходных данных плюс описание процедуры тестирования. Тестовые данные иногда генерируются автоматически. Автоматическая генерация тестовых сценариев невозможна, поскольку результаты проведения теста не всегда можно предсказать заранее.

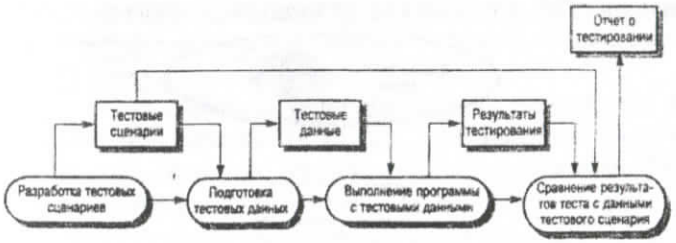

Рис. 9.2. Процесс тестирования дефектов

Полное тестирование, когда проверяются все возможные последовательности выполнения программы, нереально. Поэтому тестирование должно базироваться на некотором подмножестве всевозможных тестовых сценариев. Существуют различные методики выбора этого подмножества. Например, тестовые сценарии могут

предусмотреть выполнение всех операторов в программе, по меньшей мере, один раз. Альтернативная методика отбора тестовых сценариев базируется на опыте использования подобных систем, в этом случае тестированию подвергаются только определенные средства и функции работающей системы, например следующие:

1. Все системные функции, доступные через меню.
2. Комбинации функций, доступные через меню (например, сложное форматирование текста).
3. Если в системе предполагается ввод пользователем каких-либо входных данных, тестируются функции с правильным и неправильным вводом данных.

Из опыта тестирования (и эксплуатации) больших программных продуктов, таких, как текстовые процессоры или электронные таблицы, вытекает, что необычные комбинации функций иногда могут вызывать ошибки, но наиболее часто используемые функции всегда работают правильно.

9.2. Тестирование методом черного ящика

Функциональное тестирование, или тестирование методом черного ящика базируется на том, что все тесты основываются на спецификации системы или ее компонентов. Система представляется как "черный ящик", поведение которого можно определить только посредством изучения ее входных и соответствующих выходных данных. Другое название этого метода – **функциональное тестирование** – связано с тем, что испытатель проверяет не реализацию ПО, а только его выполняемые функции.

Рис. 9.3. Тестирование методом черного ящика

На рис. 9.3 показана модель системы, тестируемая методом черного ящика. Этот метод также применим к системам, организованным в виде набора функций или объектов. Испытатель подставляет в компонент или систему входные данные и исследует соответствующие выходные данные. Если выходные данные не совпадают с предсказанными, значит, во время тестирования ПО **успешно** обнаружена ошибка (дефект).

Основная задача испытателя – подобрать такие входные данные, чтобы среди них с высокой вероятностью присутствовали элементы множества $I_é$. Во многих случаях выбор тестовых данных основывается на предварительном опыте испытателя. Однако дополнительно к этим эвристическим знаниям можно также использовать систематический метод выбора входных данных, обсуждаемый в следующем разделе.

9.3. Области эквивалентности

Входные данные программ часто можно разбить на несколько классов. Входные данные, принадлежащие одному классу, имеют общие свойства, например это положительные числа, отрицательные числа, строки без пробелов и т.п. Обычно для всех данных из какого-либо класса поведение программы одинаково (эквивалентно). Из-за этого такие классы данных иногда называют областями эквивалентности. Один из систематических методов обнаружения дефектов состоит в определении всех областей эквивалентности, обрабатываемых программой. Контрольные тесты разрабатываются так, чтобы входные и выходные данные лежали в пределах этих областей.

На рис. 9.4 каждая область эквивалентности изображена в виде эллипса. Области эквивалентности входных данных – это множества данных, все элементы которых обрабатываются одинаково. Области эквивалентности выходных данных – это данные на выходе программы, имеющие общие свойства, которые позволяют считать их отдельным классом. Корректные и некорректные входные данные также образуют две области эквивалентности.

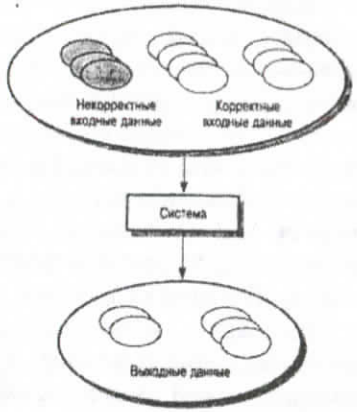

Рис. 9.4. Области эквивалентности

После определения областей эквивалентности для каждой из них подбираются тестовые данные. При выборе тестовых данных можно руководствоваться следующим полезным правилом: для тестов выбираются данные, расположенные на границе области эквивалентности, и отдельно данные, лежащие внутри этой области. Основная причина такого выбора данных заключается в следующем. В процессе разработки системы разработчики и программисты используют для тестов типичные значения входных данных, находящиеся внутри области эквивалентности. Граничные значения часто нетипичны (например, нулевое значение обрабатывается не так, как неотрицательные числа) и потому игнорируются программистами. Хотя чаще всего ошибки в программе возникают именно при обработке подобных нетипичных значений.

Области эквивалентности определяются на основании программной спецификации или документации пользователя и опыта испытателя, выбирающего классы значений входных данных, пригодные для обнаружения дефектов. Пусть, например, в спецификации программы указано, что в программу могут вводиться от 4 до 10 целых пятизначных чисел. Области эквивалентности и возможные значения тестовых входных данных для этого примера показаны на рис. 9.5.

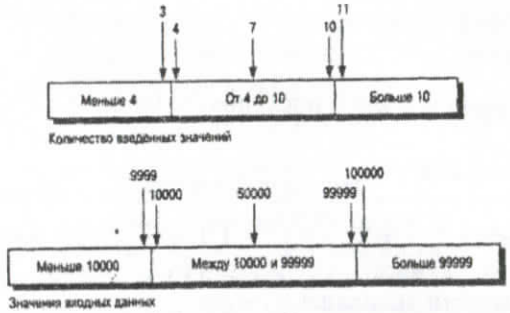

Рис. 9.5. Области эквивалентности

Покажем получение тестовых данных на примере спецификации упрощенной программы **Seach** (поиск), которая выполняет поиск заданного элемента **key** (ключ) в последовательности элементов. Программа возвращает номер позиции этого элемента в последовательности. Спецификация программы, представленная во врезке 9.1, содержит предусловие и постусловие. Предусловие указывает, что программа поиска не работает с пустыми последовательностями. Постусловие определяет, что если элемент, равный ключу, есть в последовательности, то переменная **Found** принимает значение **true** (истина). Индекс **L** обозначает позицию ключевого элемента в последовательности. Если элемент, равный ключу, в последовательности отсутствует, то этот индекс не определен.

Врезка 9.1. Спецификация программы поиска

Процедура: Seach (Key: ELEM; T: SEQ of ELEM;
Found: **in out** BOOLEAN; L: **in out** ELEM_INDEX);

Предусловие

-- в последовательности должен быть хотя бы один элемент

T'FIRST<= T'LAST

Постусловие

-- если элемент обнаружен под номером L

(Found **and** T(L) = Key)

или

-- если элемента нет в последовательности

(**not** Found **and**

not (exists i, T'FIRST >= i <= T'LAST, T(i) = Key))

Согласно данной спецификации, можно определить две очевидные области эквивалентности:

• последовательности входных данных, содержащие ключевой элемент (**Found = true**);

• последовательности входных данных, не содержащие ключевого элемента (**Found = false**).

При определении областей эквивалентности руководствуются различными правилами. Вот несколько правил выбора тестирующих последовательностей.

1. Тестирующая последовательность может состоять из одного элемента. Обычно считается, что последовательности состоят из нескольких элементов и программисты иногда закладывают такое представление в свои программы. Следовательно, если ввести последовательность из одного элемента, программа может сработать неправильно.

2. Следует использовать в разных тестах различные последовательности, содержащие разное количество элементов. Это уменьшает вероятность того, что программа имеющая дефекты, случайно выдаст правильные результаты в силу некоторых случайных свойств входных данных.

3. Следует использовать тестирующие последовательности, в которых ключевой элемент является первым, средним и последним элементом последовательности. Такой метод помогает выявить проблемы на границах областей эквивалентности.

Исходя из этих правил, можно определить еще две области эквивалентности входных данных для программы **Seach**.

• Входная последовательность состоит из одного элемента.

• Во входной последовательности больше одного элемента.

Эти области комбинируются с определенными ранее областями эквивалентности в результате будут получены области эквивалентности, представленные в табл. 9.1.

Таблица 9.1. Области эквивалентности для программы поиска

Последовательность	Ключевой элемент
Один элемент	Есть в последовательности
Один элемент	Нет в последовательности
Несколько элементов	Первый элемент последовательности
Несколько элементов	Последний элемент последовательности
Несколько элементов	Средний элемент последовательности
Несколько элементов	Нет в последовательности

Входная последовательность (T)	Key	Выходные данные (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

В табл. 9.1 также представлен набор возможных тестовых данных, взятых из этих областей. Если ключевого элемента нет в последовательности, значение L не определено. При подборе тестовых данных применялось правило выбора последовательностей, согласно которому в разных тестах следует использовать последовательности разных размеров.

Множество вводимых значений, используемых для тестирования программы поиска, не является полным. Например, в работе программы может произойти сбой, если входная последовательность содержит элементы 1, 2, 3 или 4. Однако разумно предположить, что если не обнаружены дефекты при обработке одного элемента какого-либо класса эквивалентности, то тесты с любыми другими элементами этого класса также не выявят дефектов. Конечно, это не означает, что в программе отсутствуют дефекты. Возможно, не все области эквивалентности определены или определены неверно, или неправильно подобраны тестовые данные.

Здесь намеренно не рассматриваются тесты, которые проверяют порядок и тип используемых параметров. Возможные ошибки в использовании параметров лучше всего может выявить инспектирование программ или автоматический статический анализ. По этой же причине при тестировании не проверяется непредвиденное искажение данных на выходе программного компонента.

9.4. Структурное тестирование

Метод структурного тестирования (рис. 9.6) предполагает создание тестов на основе структуры системы и ее реализации. Такой подход иногда называют тестированием методом "белого ящика", "стеклянного ящика" или "прозрачного ящика", чтобы отличать его от тестирования методом черного ящика.

Рис.9.6. Структурное тестирование

Как правило, структурное тестирование применяется к относительно небольшим программным элементам, например к подпрограммам или методам, ассоциированным с объектами. При таком подходе испытатель анализирует программный код и для получения тестовых данных использует знания о структуре компонента. Например, из анализа кода можно определить, сколько контрольных тестов нужно выполнить для того, чтобы в процессе тестирования все операторы выполнились, по крайней мере один раз.

Знание алгоритма, используемого при реализации некоторой функции, можно применять для определения областей эквивалентности. В качестве примера возьмем спецификацию программы поиска (см. врезку 9.1), реализованную на языке Java в виде процедуры бинарного поиска (листинг 9.1). Здесь реализованы более строгие предусловия. Последовательность представлена в виде массива, массив должен быть упорядоченным, значение нижней границы массива должно быть меньше значения верхней границы.

Листинг 9.1. Процедура бинарного поиска


```
class BinSearch {
//Реализация функции бинарного поиска;
//на входе: упорядоченный массив объектов и ключевой элемент
key
//Возвращает объект с двумя атрибутами:
//index - значение индекса массива
//found - логическая переменная,
//показывает, есть или нет ключевой элемент в массиве
//Если в массиве нет элемента, совпадающего с key, key = -1
public static void search (int key, int [] elemArray, Result r)
{
int bottom = 0;
int top = elemArray.length - 1;
int mid;
r.found = false; r.index = -1;
while ( bottom <= top )
{
mid = (top + bottom) / 2;
if (elemArray [mid] == key)
{
r.index = mid;
r.found = true;
return;
} //часть if
else
{
if (elemArray[mid] < key)
bottom = mid + 1;
else
top = mid - 1 ;
}
} //цикл while
} // поиск
} //BinSearch
```

Из текста программы видно, что во время ее выполнения область поиска разделяется на три части, каждая из которых является областью эквивалентности (рис. 9.7). При проверке программы в

качестве тестовых данных необходимо взять последовательности с ключевыми элементами, расположенными на границах этих областей.

Рис. 9.7. Классы эквивалентности для бинарного поиска

Тестовые данные, представленные в табл. 9.7, необходимо изменить, поскольку элементы входного массива должны быть отсортированы в возрастающем порядке. Кроме того, следует добавить тестовые данные, где ключевой элемент расположен возле среднего элемента массива. Полученное множество тестовых данных для программы бинарного поиска представлено в табл. 9.2.

Таблица 9.2. Тестовые данные для программы бинарного поиска

Входной массив (T)	Ключ (Key)	Результат (Found, L)
17	17	true, 1
17	0;	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Тестирование ветвей

Это метод структурного тестирования, при котором проверяются все независимо выполняемые ветви компонента или программы. Если выполняются все независимые ветви, то и все операторы должны выполняться, по крайней мере, один раз. Более того, все условные операторы тестируются как с истинными, так и с ложными значениями условий. В объектно-ориентированных системах тестирование ветвей используется для тестирования методов, ассоциированных с объектами.

Количество ветвей в программе обычно пропорционально ее размеру. После интеграции программных модулей в систему, методы структурного тестирования оказываются невыполнимыми. Поэтому

методы тестирования ветвей, как правило, используются при тестировании отдельных программных элементов и модулей.

При тестировании ветвей не проверяются все возможные комбинации ветвей программы. Не считая самых тривиальных программных компонентов без циклов, подобная полная проверка компонента оказывается нереальной, так как в программах с циклами существует бесконечное число возможных комбинаций ветвей. В программе могут быть дефекты, которые проявляются только при определенных комбинациях ветвей, даже если все операторы программы протестированы (т.е. выполнились) хотя бы один раз.

Метод тестирования ветвей основывается на графе потоков управления программы. Этот граф представляет собой скелетную модель всех ветвей программы. Граф потоков управления состоит из узлов, соответствующих ветвлениям решений, и дуг, показывающих поток управления. Если в программе нет операторов безусловного перехода, то создание графа – достаточно простой процесс. При построении графа потоков все последовательные операторы (операторы присвоения, вызова процедур и ввода-вывода) можно проигнорировать. Каждое ветвление операторов условного перехода (if-then-else или case) представлено отдельной ветвью, а циклы обозначаются стрелками, концы которых замкнуты на узле с условием цикла. На рис. 9.8 показаны циклы и ветвления в графе потоков управления программы бинарного поиска.

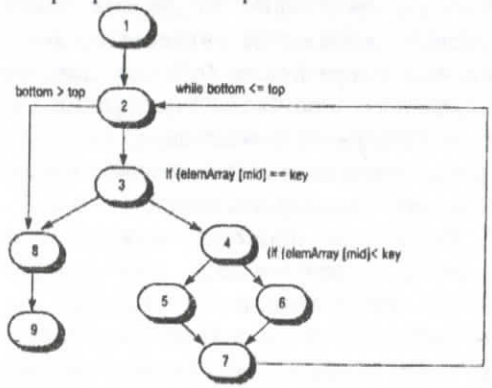

Рис. 9.8. Граф потоков управления программы бинарного поиска

Цель структурного тестирования – удостовериться, что каждая независимая ветвь программы выполняется хотя бы один раз.

Независимая ветвь программы – это ветвь, которая проходит, по крайней мере, по одной новой дуге графа потоков. В терминах программы это означает ее выполнение при новых условиях. С помощью трассировки в графе потоков управления программы бинарного поиска можно выделить следующие независимых ветвей.

1, 2, 3, 8, 9
1, 2, 3, 4, 6, 7, 2
1, 2, 3, 4, 5, 7, 2
1, 2, 3, 4, 6, 7, 2, 8, 9

Если все эти ветви выполняются, можно быть уверенным в том, что, во-первых, каждый оператор выполняется, по крайней мере один раз и, во-вторых, каждая ветвь выполняется при условиях, принимающих как истинные, так и ложные значения.

Количество независимых ветвей в программе можно определить, вычислив цикломатическое число графа потоков управления программы. Цикломатическое число С любого связанного графа G вычисляется по формуле

C (G) = количество дуг – количество узлов + 2.

Для программ, не содержащих операторов безусловного перехода, значение цикломатического числа всегда больше количества проверяемых условий. В составных условиях, содержащих более одного логического оператора, следует учитывать каждый логический оператор. Например, если в программе шесть операторов **if** и один цикл **while**, то цикломатическое число равно 8. Если одно условное выражение является составным выражением с двумя логическими операторами (объединенными операторами and или or), то цикломатическое число будет равно 10. Цикломатическое число программы бинарного поиска равно 4.

После определения количества независимых ветвей в программе путем вычисления цикломатического числа разрабатываются контрольные тесты для проверки каждой ветви. Минимальное количество тестов, требующееся для проверки всех ветвей программы, равно цикломатическому числу.

Проектирование контрольных тестов для программы бинарного поиска не вызывает затруднений. Однако, если программы имеют сложную структуру ветвлений, трудно предсказать, как будет выполняться какой-либо отдельный контрольный тест. В таких

случаях используется динамический анализатор программ для составления рабочего профиля программы.

Динамические анализаторы программ – это инструментальные средства, которые работают совместно с компиляторами. Во время компилирования в сгенерированный код добавляются дополнительные инструкции, подсчитывающие, сколько раз выполняется каждый оператор программы. Чтобы при выполнении отдельных контрольных тестов увидеть, какие ветви в программе выполнялись, а какие нет, распечатывается рабочий профиль программы, где видны непроверенные участки.

9.5. Тестирование сборки

После того как протестированы все отдельные программные компоненты, выполняется сборка системы, в результате чего создается частичная или полная система. Процесс интеграции системы включает сборку и тестирования полученной системы, в ходе которого выявляются проблемы, возникающие при взаимодействии компонентов. Тесты, проверяющие сборку системы, должны разрабатываться на основе системной спецификации, причем тестирование сборки следует начинать сразу после создания работоспособных версий компонентов системы.

Во время тестирования сборки возникает проблема локализации выявленных ошибок. Между компонентами системы существуют сложные взаимоотношения, и при обнаружении аномальных выходных данных бывает трудно установить источник ошибки. Чтобы облегчить локализацию ошибок, следует использовать пошаговый метод сборки и тестирования системы. Сначала следует создать минимальную конфигурацию системы и ее протестировать. Затем в минимальную конфигурацию нужно добавить новые компоненты и снова протестировать, и так далее до полной сборки системы.

В примере на рис. 9.9 последовательность тестов Tl, T2 и T3 сначала выполняется в системе, состоящей из модулей А и В (минимальная конфигурация системы). Если во время тестирования обнаружены дефекты, они исправляются. Затем в систему добавляется модуль С. Тесты Tl, T2 и T3 повторяются, чтобы убедиться, что в новой системе нет никаких неожиданных

взаимодействий между модулями A и B. Если в ходе тестирования появились какие-то проблемы, то, вероятно, они возникли во взаимодействиях с новым модулем C. Источник проблемы локализован, таким образом упрощается определение дефекта и его исправление. Затем система запускается с тестами T4. На последнем шаге добавляется модуль D и система тестируется еще раз выполняемыми ранее тестами, а затем новыми тестами T5.

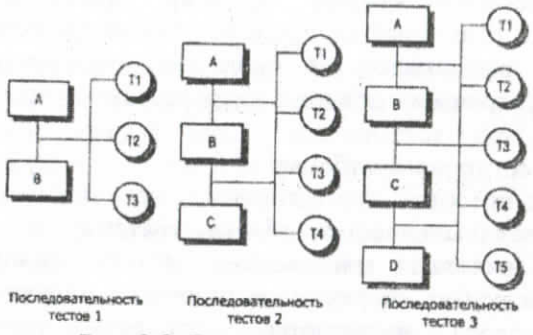

Рис. 9.9. Тестирование сборки

Конечно, на практике редко встречаются такие простые модели. Функции системы могут быть реализованы в нескольких компонентах. Тестирование новой функции, таким образом, требует интеграции сразу нескольких компонентов. В этом случае тестирование может выявить ошибки во взаимодействиях между этими компонентами и другими частями системы. Исправление ошибок может оказаться сложным, так как в данном случае ошибки влияют на целую группу компонентов, реализующих конкретную функцию. Более того, при интеграции нового компонента может измениться структура взаимосвязей между уже протестированными компонентами. Вследствие этого могут выявиться ошибки, которые не были выявлены при тестировании более простой конфигурации.

9.6. Нисходящее и восходящее тестирование

Методики нисходящего и восходящего тестирования (рис. 9.10) отражают разные подходы к системной интеграции. При нисходящей интеграции компоненты высокого уровня интегрируются и тестируются еще до окончания их проектирования и реализации. При восходящей интеграции перед разработкой компонентов более

высокого уровня сначала интегрируются и тестируются компоненты нижнего уровня.

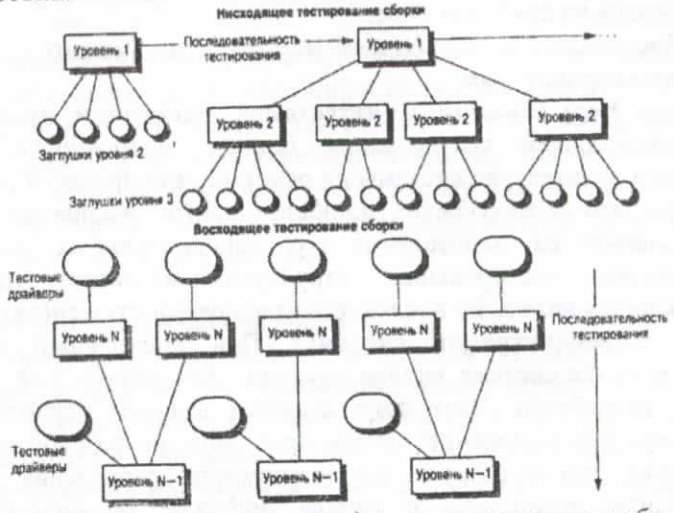

Рис. 9.10. Нисходящее и восходящее тестирование сборки

Нисходящее тестирование является неотъемлемой частью процесса нисходящей разработки систем, при котором сначала разрабатываются компоненты верхнего уровня, а затем компоненты, находящиеся на нижних уровнях иерархии. Программу можно представить в виде одного абстрактного компонента с субкомпонентами, являющимися заглушками. Заглушки имеют такой же интерфейс, что и компонент, но с ограниченной функциональностью. После того как компонент верхнего уровня запрограммирован и протестирован, таким же образом реализуются и тестируются его субкомпоненты. Процесс продолжается до тех пор, пока не будут реализованы компоненты самого нижнего уровня. Затем вся система тестируется целиком.

При восходящем тестировании, наоборот, сначала интегрируются и тестируются модули, расположенные на более низких уровнях иерархии. Затем выполняется сборка и тестирование модулей, расположенных на верхнем уровне иерархии, и так до тех пор, пока не будет протестирован последний модуль. При таком подходе не требуется наличие законченного архитектурного проекта системы, и поэтому он может начинаться на раннем этапе процесса

разработки. Обычно такой подход применяется тогда, когда в системе есть повторно используемые компоненты или модифицированные компоненты из других систем.

Нисходящее и восходящее тестирование можно сравнить по четырем направлениям:

1. **Верификация и аттестация системной архитектуры**. При нисходящем тестировании больше возможностей выявить ошибки в архитектуре системы на раннем этапе процесса разработки. Обычно это структурные ошибки, раннее выявление которых предполагает их исправление без дополнительных затрат. При восходящем тестировании структура высокого уровня не утверждается вплоть до последнего этапа разработки системы.
2. **Демонстрация системы**. При нисходящей разработке незаконченная система вполне пригодна для работы уже на ранних этапах разработки. Этот факт является важным психологическим стимулом использования нисходящей модели разработки систем, поскольку демонстрирует осуществимость управления системой. Аттестация проводится в начале процесса тестирования путем создания демонстрационной версии системы. Но если система создается из повторно используемых компонентов, то и при восходящей разработке также можно создать ее демонстрационную версию.
3. **Реализация тестов**. Нисходящее тестирование сложно реализовать, так как необходимо моделировать программы-заглушки нижних уровней. Программы-заглушки могут быть упрощенными версиями представляемых компонентов. При восходящем тестировании для того, чтобы использовать компоненты нижних уровней, необходимо разработать тестовые драйверы, которые эмулируют окружение компонента в процессе тестирования.
4. **Наблюдение за ходом испытаний**. При нисходящем и восходящем тестировании могут возникать проблемы, связанные с наблюдениями за ходом тестирования. В большинстве систем, разрабатываемых сверху вниз, более верхние уровни системы, которые реализованы первыми, не генерируют выходные данные, однако для проверки этих уровней нужны какие-либо выходные результаты. Испытатель должен создать искусственную среду для генерации результатов теста. При восходящем тестировании также

может возникнуть необходимость в создании искусственной среды (тестовых драйверов) для исследования компонентов нижних уровней.

На практике при разработке и тестировании систем чаще всего используется композиция восходящих и нисходящих методов. Разные сроки разработки для разных частей системы предполагают, что группа, проводящая тестирование и интеграцию, должна работать с каким-либо готовым компонентом. Поэтому во время процесса тестирования сборки в любом случае необходимо разрабатывать как заглушки, так и тестовые драйверы.

9.7. Тестирование интерфейсов

Как правило, тестирование интерфейса выполняется в тех случаях, когда модули или подсистемы интегрируются в большие системы. Каждый **модуль** или подсистема имеет заданный интерфейс, который вызывается другими компонентами системы. **Цель тестирования интерфейса** – выявить ошибки, возникающие в системе вследствие ошибок в интерфейсах или неправильных предположений об интерфейсах.

Схема тестирования интерфейса показана на рис. 9.11. Стрелки в верхней части схемы означают, что контрольные тесты применяются не к отдельным компонентам, а к подсистемам, полученным в результате комбинирования этих компонентов.

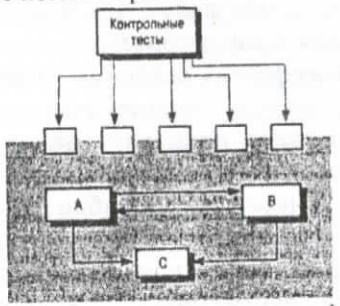

Рис. 9.11. Тестирование интерфейсов

Данный тип тестирования особенно важен в объектно-ориентированном проектировании, в частности при повторном использовании объектов и классов объектов. Объекты в значительной

степени определяются с помощью интерфейсов и могут повторно использоваться в различных комбинациях с разными объектами и в разных системах. Во время тестирования отдельных объектов невозможно выявить ошибки интерфейса, так как они являются скорее результатом взаимодействия между объектами, чем изолированного поведения одного объекта.

Между компонентами программы могут быть разные типы интерфейсов и соответственно разные типы ошибок интерфейсов.

1. **Параметрические интерфейсы**. Интерфейсы, в которых ссылки на данные и иногда функции передаются в виде параметров от одного компонента к другому.

2. **Интерфейсы разделяемой памяти**. Интерфейсы, в которых какой-либо блок памяти совместно используется разными подсистемами. Одна подсистема помещает данные в память, а другие подсистемы используют эти данные.

3. **Процедурные интерфейсы.** Интерфейсы, в которых одна подсистема инкапсулирует набор процедур, вызываемых из других подсистем. Такой тип интерфейса имеют объекты и абстрактные типы данных.

4. **Интерфейсы передачи сообщений**. Интерфейсы, в которых одна подсистема запрашивает сервис у другой подсистемы посредством передачи ей сообщения. Ответное сообщение содержит, результаты выполнения сервиса. Некоторые объектно-ориентированные системы имеют такой тип интерфейсов; например, так работают системы клиент/сервер.

Ошибки в интерфейсах являются наиболее распространенными типами ошибок в сложных системах и делятся на три класса.

- **Неправильное использование интерфейсов**. Компонент вызывает другой компонент и совершает ошибку при использовании его интерфейса. Данный тип ошибки особенно распространен в параметрических интерфейсах; например, параметры могут иметь неправильный тип, следовать в неправильном порядке или же иметь неверное количество параметров.

- **Неправильное понимание интерфейсов.** Вызывающий компонент, в который заложена неправильная интерпретация спецификации интерфейса вызываемого компонента, предполагает определенное поведение этого компонента. Если поведение

вызываемого компонента не совпадает с ожидаемым, поведение вызывающего компонента становится непредсказуемым. Например, если программа бинарного поиска вызывается для поиска заданного элемента в неупорядоченном массиве, то в работе программы произойдет сбой.

- **Ошибки синхронизации**. Такие ошибки встречаются в системах реального времени, где используются интерфейсы разделяемой памяти или передачи сообщений. Подсистема – производитель данных и подсистема – потребитель данных могут работать с разной скоростью. Если при проектировании интерфейса не учитывать этот фактор, потребитель может, например, получить доступ к устаревшим данным, потому что производитель к тому моменту еще не успел обновить совместно используемые данные.

Тестирование дефектов интерфейсов сложно, поскольку некоторые ошибки могут проявиться только в необычных условиях. Например, пусть некий объект реализует очередь в виде структуры списка фиксированного размера. Вызывающий его объект при вводе очередного элемента не проверяет переполнение очереди, так как предполагает, что очередь реализована как структура неограниченного размера. Такую ситуацию можно обнаружить только во время выполнения специальных тестов: специально вызывается переполнение очереди, которое приводит к непредсказуемому поведению объекта.

Другая проблема может возникнуть из-за взаимодействий между ошибками в разных программных модулях или объектах. Ошибки в одном объекте можно выявить только тогда, когда поведение другого объекта становится непредсказуемым. Например, для получения сервиса один объект вызывает другой объект и полагает, что полученный ответ правильный. Если объект неправильно понимает вычисленные значения, возвращаемое значение может быть достоверным, но неправильным. Такие ошибки можно выявить только тогда, когда оказываются неправильными дальнейшие вычисления.

Вот несколько общих правил тестирования интерфейсов.

1. Просмотрите тестируемый код и составьте список всех вызовов, направленных к внешним компонентам. Разработайте такие наборы тестовых данных, при которых параметры, передаваемые

внешним компонентам, принимают крайние значения из диапазонов их допустимых значений. Использование экстремальных значений параметров с высокой вероятностью обнаруживает несоответствия в интерфейсах.

2. Если между интерфейсами передаются указатели, всегда тестируйте интерфейс с нулевыми параметрами указателя.

3. При вызове компонента через процедурный интерфейс используйте тесты, вызывающие сбой в работе компонента. Одна из наиболее распространенных причин ошибок в интерфейсе – неправильное понимание спецификации компонентов.

4. В системах передачи сообщений используйте тесты с нагрузкой, которые рассматриваются в следующем разделе. Разрабатывайте тесты, генерирующие в несколько раз большее количество сообщений, чем будет в обычной работе системы. Эти же тесты позволяют обнаружить проблемы синхронизации.

5. При взаимодействии нескольких компонентов через разделяемую память разрабатывайте тесты, которые изменяют порядок активизации компонентов. С помощью таких тестов можно выявить сделанные программистом неявные предположения о порядке использования компонентами разделяемых данных.

Обычно статические методы тестирования более рентабельны, чем специальное тестирование интерфейсов. В языках со строгим контролем типов, например Java, многие ошибки интерфейсов помогает обнаружить компилятор. В языках со слабым контролем типов (например, C) ошибки интерфейса может выявить статический анализатор, такой как LINT. Кроме того, при инспектировании программ можно сосредоточиться именно на проверке интерфейсов компонентов.

9.8. Инструментальные средства тестирования

Тестирование – дорогой и трудоемкий этап разработки программных систем. Поэтому создан широкий спектр инструментальных средств для поддержки процесса тестирования, которые значительно сокращают расходы на него.

На рис. 9.14 показаны возможные инструментальные средства тестирования и отношения между ними. Перечислим их.

1. **Организатор тестов**. Управляет выполнением тестов. Он отслеживает тестовые данные, ожидаемые результаты и тестируемые функции программы.

2. **Генератор тестовых данных**. Генерирует тестовые данные для тестируемой программы. Он может выбирать тестовые данные из базы данных или использовать специальные шаблоны для генерации случайных данных необходимого вида.

3. **Оракул.** Генерирует ожидаемые результаты тестов. В качестве оракулов могут выступать предыдущие версии программы или исследуемого объекта. При тестировании параллельно запускаются оракул и тестируемая программа и сравниваются результаты их выполнения.

4. **Компаратор файлов.** Сравнивает результаты тестирования с результатами предыдущего тестирования и составляет отчет об обнаруженных различиях. Компараторы особенно важны при сравнении различных версий программы. Различия в результатах указывают на возможные проблемы, существующие в новой версии системы.

5. **Генератор отчетов**. Формирует отчеты по результатам проведения тестов.

6. **Динамический анализатор**. Добавляет в программу код, который подсчитывает, сколько раз выполняется каждый оператор. После запуска теста создает исполняемый профиль, в котором показано, сколько раз в программе выполняется каждый оператор.

7. **Имитатор**. Существует несколько типов имитаторов. Целевые имитаторы моделируют машину, на которой будет выполняться программа. Имитатор пользовательского интерфейса – это программа, управляемая сценариями, которая моделирует взаимодействия с интерфейсом пользователя. Имитатор ввода-вывода генерирует последовательности повторяющихся транзакций.

Требования, предъявляемые к процессу тестирования больших систем, зависят от типа разрабатываемого приложения. Поэтому инструментальные средства тестирования неизменно приходится адаптировать к процессу тестирования конкретной системы.

Для создания полного комплекса инструментального средства тестирования, как правило, требуется много сил и времени. Весь набор инструментальных средств, показанных на рис. 9.14,

используется только при тестировании больших систем. Для таких систем полная стоимость тестирования может достигать 50% от всей стоимости разработки системы. Вот почему выгодно инвестировать разработку высококачественных и производительных CASE-средств тестирования.

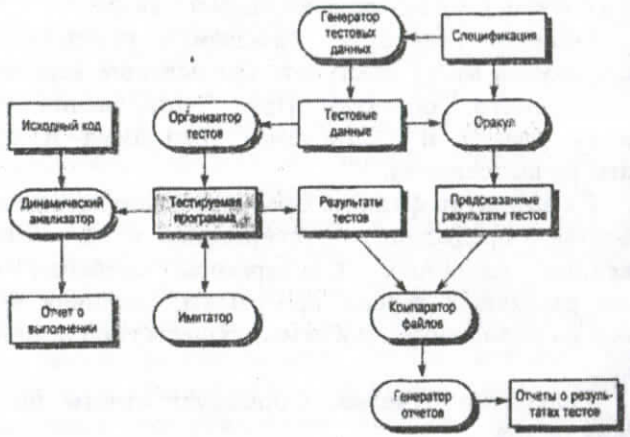

Рис. 9.14. Инструментальные средства тестирования

Контрольные вопросы:

1) Что такое тестовые сценарии?
2) Перечислите правила выбора тестирующих последовательностей?
3) Что такое динамические анализаторы программ?
4) По каким четырем направлениям можно сравнить нисходящее и восходящее тестирование?
5) Перечислите общие правила тестирования интерфейсов.

Ключевые слова: тестовые сценарии функциональное тестирование, основная задача испытателя, динамические анализаторы программ, верификация и аттестация системной архитектуры, демонстрация системы, реализация тестов, наблюдение за ходом испытаний, параметрические интерфейсы, интерфейсы разделяемой памяти, процедурные интерфейсы, интерфейсы передачи сообщений, неправильное использование интерфейсов, неправильное понимание интерфейсов, ошибки синхронизации, тестирование, организатор тестов, генератор тестовых данных,

оракул, компаратор файлов, генератор отчетов, динамический анализатор, имитатор.

Keywords: functional testing test cases, the main objective test, dynamic analyzers programs, verification and certification of the system architecture, the demonstration system, the implementation of test, monitoring the progress of the test, parametric interfaces, shared memory, procedural interfaces, messaging misuse interfaces , misunderstanding interfaces, synchronization errors, testing, organizer test generator test data, oracle comparator files, report generator, dynamic analyzer, simulator.

Kalit so'zlar: test ssenariylari, funksional test, sinovchining asosiy masalasi, dinamik analizatorlar, tizim demonstratsiyasi, testlar realizatsiyasi, sinovlar kuzatish, parametrik interfeyslar, bo'limli xotira interfeyslari, protsedura interfeyslari, xabarlar yuborish interfeyslari, sinxronizatsiya hatoliklari, test ma'lumotlar generatori, orakul, dinamik analizator, simulyator.

Упражнения

1. Обсудите различия между тестированием методом черного ящика и структурным тестированием. Подумайте, каким образом можно совместно использовать эти методы в процессе тестирования дефектов.
2. Какие проблемы тестирования могут возникнуть в программах, которые обрабатывают как очень большие, так и очень малые числа?
3. Разработайте набор тестовых данных для следующих компонентов:
   - программа сортировки массивов целых чисел;
   - программа, которая вычисляет количество символов (отличных от пробелов) в текстовых строках;
   - программа, которая проверяет текстовые строки и заменяет последовательности пробелов одним пробелом;
   - объект, реализующий символьные строки разной длины. Среди операций, ассоциированных с этим объектом, должны быть операция конкатенации, операция определения длины строки и операция выбора подстроки.

4. Напишите код для первых трех перечисленных выше программ, используя любой язык программирования. Для каждой программы рассчитайте цикломатическое число.
5. На примере небольшой программы покажите, почему практически невозможно полностью протестировать программу.
6. Для проверки созданных вами программ, разработайте контрольные тесты в дополнение к тем, которые уже были рассмотрены. Выявил ли анализ кода упущения в первоначальных наборах тестовых данных?
7. Реализуйте (на языке Java или C++) класс объектов SYMBOL_TABLE, который будет использоваться как часть системы компиляции. Этот класс должен иметь следующие методы: добавление имени и типа данных в таблицу идентификаторов, удаление имени, изменение информации, связанной с именем, и поиск по таблице. Организуйте инспектирование кода этого объекта и сделайте подсчет обнаруженных ошибок. Протестируйте объект методом черного ящика и сравните ошибки, выявленные при тестировании и при инспектировании.
8. Объясните, почему методы нисходящего и восходящего тестирования не подходят для объектно-ориентированных систем.
9. Создайте сценарии тестирования состояний микроволновой печи, модель состояний.

ГЛАВА 10. МОДЕРНИЗАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Невозможно создать систему, которая не потребует изменений в будущем. Как только программное обеспечение вводится в эксплуатацию, возникают новые требования к системе, обусловленные непрерывным развитием бизнес-процессов и все возрастающими общими требованиями к программным системам. Иногда в системе следует изменить некоторые составляющие в целях повышения производительности или улучшения других характеристик, а также для исправления обнаруженных ошибок. Все это требует дальнейшего развития системы после ее ввода в эксплуатацию.

Полная зависимость организаций от программного обеспечения, которое к тому же обходится в достаточно круглую сумму, объясняет исключительную важность серьезного отношения к ПО. Это предусматривает дополнительные вложения в эволюцию уже эксплуатируемой системы с тем, чтобы обеспечить прежний уровень ее производительности.

Существует несколько стратегических подходов к процессу модернизации ПО.

1. **Сопровождение программного обеспечения**. Это наиболее часто используемый подход, который заключается в изменении отдельных частей ПО в ответ на растущие требования, но с сохранением основной системной структуры. Подробнее этот вопрос освещен в разделе 10.2.

2. **Эволюция системной архитектуры**. Этот подход более радикальный, чем сопровождение ПО, так как предполагает существенные изменения в программной системе. Эта стратегия модернизации ПО подробно раскрыта в разделе 10.3.

3. **Реинжиниринг программного обеспечения.** Кардинально отличается от других подходов, так как модернизация предусматривает не внесение каких-то новых компонентов, а наоборот, упрощение системы и удаление из нее всего лишнего. При этом возможны изменения в архитектуре, но без серьезных переделок. Этому вопросу посвящена глава 10.

Приведенные стратегии не исключают одна другую. Иногда для упрощения системы перед изменением архитектуры или для переделки некоторых ее компонентов применяется реинжиниринг. Некоторые части системы заменяются серийными, а более стабильные системные компоненты продолжают функционирование. Выбор стратегии модернизации системы основывается не только на технических характеристиках, но и на том, насколько хорошо система поддерживает деловую активность компании.

Разные стратегии могут также применяться к отдельным частям системы или к отдельным программам наследуемой системы. Сопровождение приемлемо для программ со стабильной и четкой структурой, не требующей особого внимания. Для других программ, которые постоянно контактируют со многими пользователями, можно изменить архитектуру так, чтобы интерфейс пользователя запускался на машине клиента. Еще один компонент в этой же системе можно заменить аналогичной программой стороннего производителя. Однако при реинжиниринге обычно необходимо изменять все компоненты системы.

Изменения в ПО служат причиной появления многочисленных версий системы и ее компонентов. Поэтому особенно важно внимательно следить за всеми этими изменениями, а также за тем, чтобы версия компонента соответствовала той версии системы, в которой он применяется.

10.1. Динамика развития программ

Под динамикой развития программ подразумевается исследование изменений в программной системе. Основной работой в этой области является. Результатом этих исследований стало появление ряда "законов" Лемана (Lehman), относящихся к модернизации систем. Считается, что эти законы неизменны и применимы практически во всех случаях. Они сформулированы после исследования процесса создания и эволюции ряда больших программных систем. Эти законы (в сущности, не законы, а гипотезы) приведены в табл. 10.1.

Таблица 10.1. Законы Лемана

Закон	Описание
Непрерывность модернизации	Для программ, эксплуатируемых в реальных условиях, модернизация – это необходимость, иначе их полезность снижается
Возрастающая сложность	По мере развития программы становятся все более сложными. Для упрощения или сохранения их структуры необходимы дополнительные затраты
Эволюция больших систем	Процесс развития систем саморегулируемый. Такие характеристики системы, как размер, время между выпусками очередных версий и количество регистрируемых ошибок, для каждой версии программы остаются практически неизменными
Организационная стабильность	Жизненный цикл системы относительно стабилен, независимо от средств, выделяемых (или не выделяемых) на ее развитие
Стабильность количества изменений	За весь жизненный цикл системы количество изменений в каждой версии остается приблизительно одинаковым

Из первого закона вытекает необходимость постоянного сопровождения системы. При изменении окружения, в котором работает система, появляются новые требования, и система должна неизбежно изменяться с тем, чтобы им соответствовать. Изменения системы носят циклический характер, когда новые требования порождают появление новой версии системы, что, в свою очередь, вызывает изменения системного окружения; это находит отражение в формировании новых требований к системе и т.д.

Второй закон констатирует нарушение структуры системы после каждой модификации. Единственным способом избежать этого, по всей видимости, является только профилактическое обслуживание, которое, однако, требует средств и времени. При этом

совершенствуется структура программы без изменения ее функциональности. Поэтому в бюджете, предусмотренном на содержание системы, следует также учесть и эти дополнительные затраты.

Самым спорным и, пожалуй, самым интересным законом Лемана является третий. Согласно этому закону, все большие системы имеют собственную динамику изменений, которая устанавливается на начальном этапе разработки системы. Этим определяются возможности сопровождения системы и ограничивается количество модификаций. Предполагается, что этот закон является результатом действия фундаментальных структурных и организационных факторов. Как только система превышает определенный размер, она начинает действовать подобно некой инерционной массе. Размер становится препятствием для новых изменений, поскольку эти изменения с большой вероятностью станут причиной ошибок в системе, которые снизят эффективность нововведений в новой версии системы.

Четвертый закон Лемана утверждает, что крупные проекты по разработке программного обеспечения действуют в режиме "насыщения". Это означает, что изменения ресурсов или персонала оказывает незначительное влияние на долгосрочное развитие системы. Это, правда, уже указано в третьем законе, который утверждает, что развитие программы не зависит от решений менеджмента. Этим законом также утверждается, что крупные команды программистов неэффективны, так как время, потраченное на общение и внутри командные связи, превышает время непосредственной работы над системой.

Пятый закон затрагивает проблему увеличения количества изменений с каждой новой версией программы. Расширение функциональных возможностей системы каждый раз сопровождается новыми ошибками в системе. Таким образом, масштабное расширение функциональных возможностей в одной версии означает необходимость последующих доработок и исправления ошибок. Поэтому в следующей версии уже будут проведены незначительные модификации. Таким образом, менеджер, формируя бюджет для внесения крупных изменений в версию системы, не должен забывать

о необходимости разработки следующей версии с исправленными ошибками предыдущей версии.

В основном законы Лемана выглядят весьма разумными и убедительными. При планировании сопровождения они обязательно должны учитываться. Случается, что по коммерческим соображениям законами следует пренебречь. Например, это может быть обусловлено маркетингом, если существует необходимость провести ряд модификаций системы в одной версии. В результате все равно получится так, что одна или несколько следующих версий будут связаны с исправлением ошибок.

Может показаться, что большие различия между последовательными версиями одной и той же программы опровергнут законы Лемана. Например, Microsoft Word превратилась из простой программы текстовой обработки, требующей 25 Кбайт памяти, в огромную систему с множеством функций. Теперь, для того чтобы работать с этой программой, нужно много памяти и быстродействующий процессор. Эволюция этой программы противоречит четвертому и пятому законам Лемана. Однако здесь подозревают, что это все-таки не одна и та же программа, которая просто подверглась ряду изменений. Думаю, программа была существенно переработана; по сути, была разработана новая программа, но в рекламных целях был сохранен единый логотип.

10.2. Сопровождение программного обеспечения

Сопровождение – это обычный процесс изменения системы после ее поставки заказчику. Эти изменения могут быть как элементарно простыми (исправление ошибок программирования), так и более серьезными, связанными с корректировкой отдельных недоработок либо приведением в соответствие с новыми требованиями. Как упоминалось в вводной части главы, сопровождение не связано со значительным изменением архитектуры системы. При сопровождении тактика простая: изменение существующих компонентов системы либо добавление новых.

Существует три вида сопровождения системы.

1. **Сопровождение с целью исправления ошибок**. Обычно ошибки в программировании достаточно легко устранимы, однако ошибки проектирования стоят дорого и требуют корректировки или

перепрограммирования некоторых компонентов. Самые дорогие исправления связаны с ошибками в системных требованиях, так как здесь может понадобиться перепроектирование системы.

2. **Сопровождение с целью адаптации ПО к специфическим условиям эксплуатации**. Это может потребоваться при изменении определенных составляющих рабочего окружения системы, например аппаратных средств, операционной системы или программных средств поддержки. Чтобы адаптироваться к этим изменениям, система должна быть подвергнута определенным модификациям.

3. **Сопровождение с целью изменения функциональных возможностей системы**. В ответ на организационные или деловые изменения в организации могут измениться требования к программным средствам. В таких случаях применяется данный тип сопровождения. Наиболее существенные изменения при этом претерпевает именно программное обеспечение.

На практике однозначно четкое разграничение между различными видами сопровождения провести достаточно сложно. Ошибки в системе могут быть выявлены в том случае, если, например, система использовалась непредсказуемым способом. Поэтому наилучший способ исправления ошибок – расширение функциональных возможностей программы с тем, чтобы сделать работу с ней как можно проще. При адаптации программного обеспечения к новому рабочему окружению расширение функциональных возможностей системы будет способствовать улучшению ее работы. Также добавление определенных функций в программу может оказаться полезным, если в случае ошибок был изменен шаблон использования системы и побочным действием при расширении функциональных возможностей будет удаление ошибок.

Перечисленные типы сопровождения широко используются, хотя им подчас дают разные названия. Сопровождение с целью исправления ошибок обычно называют корректирующим. Название "адаптивное сопровождение" может относиться как к адаптации к новому рабочему окружению, так и к новым требованиям. Усовершенствование программного обеспечения может означать улучшение путем соответствия новым требованиям, а также усовершенствование структуры и производительности с сохранением

функциональных возможностей. Здесь намеренно не употребляют здесь все эти названия, чтобы избежать излишней путаницы.

Рис. 10.1. Распределение типов сопровождения

Найти современные данные относительно того, как часто используется тот или иной тип сопровождения, будет нелегко. Согласно исследованиям, которые уже несколько устарели, 65% сопровождения связано с выполнением новых требований, 18% отводится на изменения системы с целью адаптации к новому окружению и 17% связано с исправлением ошибок (рис. 10.1.). Десятилетие спустя в работе определены похожие соотношения.

Из этого можно определить, что исправление ошибок не является самым распространенным видом сопровождения. Модернизация системы в соответствии с новым рабочим окружением либо в соответствии с новыми требованиями более эффективна. Поэтому сопровождение само по себе является естественным процессом продолжения разработки системы со своими процессами проектирования, реализации и тестирования. Таким образом, спиральная модель, показанная на рис. 10.2, лучше представляет процесс развития ПО, чем каскадная модель, где сопровождение рассматривается как отдельный процесс.

Значительная часть бюджета большинства организаций уходит на сопровождение ПО, а не на само использование программных систем. В 1980-х годах было обнаружено, что во многих организациях по меньшей мере 50% всех средств, потраченных на программирование, идет на развитие уже существующих систем. В работе определено похожее соотношение затрат на различные виды сопровождения, при этом от 65 до 75% средств общего бюджета расходуется на сопровождение. Так как предприятия заменяют

старые системы коммерческим ПО, например программами планирования ресурсов, эти цифры никак не будут уменьшаться. Поэтому можно утверждать, что изменение ПО все еще остается доминирующим в статье затрат организаций на программное обеспечение.

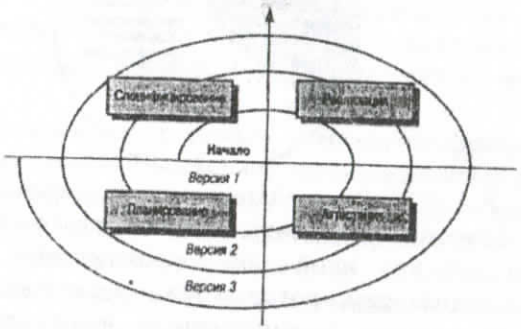

Рис. 10.2. Спиральная модель развития ПО

Соотношение между величинами средств на сопровождение и на разработку может быть разным в зависимости от предметной области, где эксплуатируется система. Для прикладных систем, работающих в деловой сфере, соотношение затрат на сопровождение в основном сравнимо со средствами, потраченными на разработку. Для встроенных систем реального времени затраты на сопровождение могут в четыре раза превышать стоимость самой разработки. Высокие требования в отношении производительности и надежности таких систем предполагают их жесткую структуру, которая труднее поддается модификации.

Можно получить значительную общую экономию средств, если заранее потратить финансы и усилия на создание системы, не требующей дорогостоящего сопровождения. Весьма затратно проводить изменения в системе после ее поставки заказчику, поскольку для этого требуется хорошо знать систему и провести анализ реализации этих изменений. Поэтому усилия, потраченные во время разработки программы на снижение стоимости такого анализа, автоматически снизят и затраты на сопровождение. Такие технологии разработки ПО, как формирование четких требований, объектно-

ориентированное программирование и управление конфигурацией, способствуют снижению стоимости сопровождения.

На рис. 10.3 показано, как снижается стоимость сопровождения хорошо разработанных систем. Здесь при разработке системы 1 выделено дополнительно $25 000 для облегчения процесса сопровождения. В результате за время эксплуатации системы это помогло сэкономить около $100 000. Из сказанного следует, что увеличение средств на разработку системы пропорционально снизит затраты на ее сопровождение.

Рис. 10.3. Расходы на разработку и сопровождение систем

Причиной высоких затрат на сопровождение является сложность модернизации системы после ее внедрения, поскольку расширить функциональные возможности намного легче в процессе создания системы. Ниже приведены ключевые факторы, которые определяют стоимость разработки и сопровождения и могут привести к подорожанию сопровождения.

1. **Стабильность команды разработчиков**. Вполне естественно, что после внедрения системы команда разработчиков распадается, специалисты будут работать над другими проектами. Новым членам команды или же отдельным специалистам, которые возьмут на себя дальнейшее сопровождение системы, будет трудно понять все ее особенности. Поэтому на понимание системы перед внесением в нее изменений уходит много времени и средств.

2. **Ответственность согласно контракту**. Контракт на сопровождение обычно заключается отдельно от договора на разработку программы. Более того, часто контракт на сопровождение может получить фирма, сама не занимающаяся разработкой. Вместе с фактором нестабильности команды это может стать причиной отсутствия в команде стимула создать легко изменяемую, удобную в сопровождении систему. Если членам команды выгодно пойти кратчайшим путем с минимальными затратами усилий, то вряд ли

они откажутся от этого даже с риском повышения последующих затрат на сопровождение.

3. **Квалификация специалистов**. Специалисты, занимающиеся сопровождением, часто не знакомы с предметной областью, где эксплуатируется система. Сопровождение не пользуется популярностью среди разработчиков. Это считается менее квалифицированной разработкой и часто поручается младшему персоналу. Более того, старые системы могут быть написаны на устаревших языках программирования, не знакомых молодым специалистам и требующих дополнительного изучения.

4. **Возраст и структура программы**. *С* возрастом структура программ нарушается вследствие частых изменений, поэтому их становится сложнее понимать и изменять. Кроме того, многие наследуемые системы были созданы без использования современных технологий. Они никогда не отличались хорошей качественной структурой; изменения, сделанные в них, были направлены скорее на повышение эффективности функционирования, чем на повышение удобства сопровождения. Документация на старые системы часто бывает неполной либо вообще отсутствует.

Первые три проблемы объясняются тем, что многие организации все еще делают различие между разработкой системы и ее сопровождением. Сопровождение считается делом второстепенным, поэтому нет никакого желания инвестировать средства для снижения затрат на будущее сопровождение. Руководству организаций необходимо помнить, что у систем редко бывает четко определенный срок функционирования, наоборот, они могут находиться в эксплуатации в той либо иной форме неограниченное время.

Дилемма заключается в следующем: или создавать системы и поддерживать их до тех пор, пока это возможно, и затем заменять их новыми, или разрабатывать постоянно эволюционирующие системы, которые могут изменяться в соответствии с новыми требованиями. Их можно создавать на основе наследуемых систем, улучшая структуру последних с помощью реинжиниринга, либо путем изменения архитектуры этих систем, что подробно рассматривается в разделе 10.3.

Последняя проблема, а именно нарушение структуры, является самой простой из них. В подходящих случаях адаптировать систему к новым аппаратным средствам может и преобразование архитектуры (см. далее). Профилактические меры при сопровождении будут полезны, если возникнет необходимость усовершенствовать систему и сделать ее более удобной для изменений.

10.3. Процесс сопровождения

Процессы сопровождения могут быть самыми разными, что зависит от типа программного обеспечения, технологии его разработки, а также от специалистов, которые непосредственно занимались созданием системы. Во многих организациях сопровождение носит неформальный характер. В большинстве случаев разработчики получают информацию о проблемах системы от самих пользователей в устной форме. Другие же компании имеют формальный процесс сопровождения со структурированной документацией на каждый его этап. Но на самом общем уровне любые процессы сопровождения имеют общие этапы, а именно: анализ изменений, планирование версий, реализация новой версии системы и поставка системы заказчику.

Рис. 10.4. Схема процесса модернизации

Процесс сопровождения начинается при наличии достаточного количества запросов на изменения от пользователей, менеджеров или покупателей. Далее оцениваются возможные изменения с тем, чтобы определить уровень модернизации системы, а также стоимость внедрения этих изменений. Если принимается решение о модернизации системы, начинается этап планирования новой версии системы. Во время планирования анализируется возможность реализации всех необходимых изменений, будь то исправление ошибок, адаптация или расширение функциональных возможностей системы. Только после этого принимается окончательное решение о том, какие именно изменения будут внесены в систему. Когда

изменения реализованы, выходит очередная версия системы. На рис. 10.4, заимствованном из книги, представлен весь процесс модернизации.

В идеале процесс модернизации должен привести к изменению системной спецификации, архитектуры и программной реализации (рис. 10.5). Новые требования должны отражать изменения, вносимые в систему. Ввод в систему новых компонентов требует ее перепроектирования, после чего необходимо повторное тестирование системы. Для анализа вносимых изменений при необходимости можно создать прототип системы. На этой стадии проводится подробный анализ изменений, при котором могут выявиться те последствия модернизации, которые не были замечены при начальном анализе изменений.

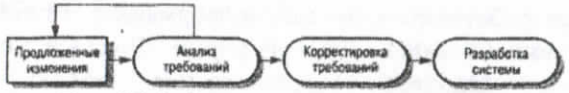

Рис. 10.5. Выполнение модернизации

Иногда в экстренных случаях требуется быстрое внесение изменений, например по следующим причинам.

- Сбой в системе, вследствие чего возникла чрезвычайная ситуация, требующая экстренного вмешательства для продолжения нормальной работы системы.
- Изменение рабочего окружения системы с непредусмотренным влиянием на систему.
- Неожиданные изменения в деловой сфере организации (из-за действий конкурентов или из-за введения нового законодательства).

В таких случаях быстрая реализация изменений имеет большую важность, чем четкое следование формальностям процесса модернизации системы. Вместо того чтобы изменять требования или структуру системы, лучше быстро внести коррективы в программный код (рис. 10.6). Однако этот подход опасен тем, что требования, системная архитектура и программный код постепенно теряют целостность. Этого трудно избежать, если принять во внимание необходимость быстрого выполнения задания, когда тщательная доработка системы откладывается на потом. Если разработчик, который изменил код, внезапно уходит из команды, то его коллеге

будет сложно привести в соответствие сделанным изменениям спецификацию и структуру системы.

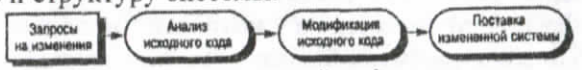

Рис. 10.6. Процесс экстренной модернизации системы

Еще одна проблема срочных изменений системы состоит в том, что из-за дефицита времени из двух возможных решений будет принято не самое лучшее (в аспекте сохранения структуры системы), а то, которое можно быстрее и эффективнее реализовать. В идеале после срочной коррекции кода системы запрос на изменения должен все еще оставаться в силе. Поэтому после тщательного анализа изменений можно отменить внесенные изменения и принять более оптимальное решение для модернизации системы. Однако на практике такая возможность используется крайне редко, чаще всего в силу субъективных причин.

10.4. Прогнозирование сопровождения

Менеджеры терпеть не могут сюрпризов, особенно если они выливаются в непредсказуемо высокие затраты. Поэтому лучше предусмотреть заранее, какие изменения возможны в системе, с какими компонентами системы будет больше всего проблем при сопровождении, а также рассчитать общие затраты на сопровождение в течение определенного периода времени. На рис. 10.7 представлены различные типы прогнозов, связанные с сопровождением, и показано, на какие вопросы они должны ответить.

Прогнозирование количества запросов на изменения системы зависит от понимания взаимосвязей между системой и ее окружением. Некоторые системы находятся в достаточно сложной взаимозависимости с внешним окружением, и изменение окружения обязательно повлияет на систему. Для того чтобы правильно судить об этих взаимоотношениях, необходимо оценить следующие показатели:

1. **Количество и сложность системных интерфейсов**. Чем больше системных интерфейсов и чем более сложными они являются, тем выше вероятность изменений в будущем.
2. **Количество изменяемых системных требований.** Как упоминалось, требования, отражающие деловую сферу или

стандарты организации, чаще изменяются, чем требования, описывающие предметную область.

3. **Бизнес-процессы, в которых используется данная система**. По мере развития бизнес-процессы приводят к появлению новых требований к системе.

Чтобы корректно спрогнозировать процесс сопровождения, нужно знать количество и типы взаимосвязей между разными компонентами системы, а также учитывать сложность этих компонентов. Различные виды сложности систем изучались в работах. Другие исследования посвящены взаимосвязям между сложностью систем и процессом сопровождения. Все эти исследования показали, что, чем выше сложность системы и ее компонентов, тем более дорогостоящим окажется сопровождение, чего и следовало ожидать.

Рис. 10.7. Прогнозирование сопровождения

В работе проведено исследование ряда коммерческих программ, написанных на языке COBOL, с использованием разных методик измерения сложности, включая размер процедур, размер модулей и количество ветвлений, что определяет сложность системы. Сравнивая сложность отдельных системных компонентов с отчетами по сопровождению, исследователи обнаружили, что снижение сложности программирования значительно сокращает расходы на сопровождение системы.

Измерение уровня сложности систем оказалось весьма полезным для выявления тех компонентов систем, которые будут особенно сложны для сопровождения. Результаты анализа ряда

системных компонентов показали, что сопровождение часто сосредоточено на обслуживании небольшого количества частей системы, которые отличаются особой сложностью. Поэтому экономически выгодно заменить сложные системные компоненты более простыми их версиями.

После введения системы в эксплуатацию появляются данные, позволяющие прогнозировать дальнейшее сопровождение системы. Перечисленные ниже показатели полезны для оценивания удобства сопровождения.

1. **Количество запросов на корректировку системы**. Возрастание количества отчетов о сбоях в системе означает увеличение количества ошибок, подлежащих исправлению при сопровождении. Это говорит об ухудшении удобства сопровождения.

2. **Среднее время, потраченное на анализ причин системных сбоев и отказов.** Этот показатель пропорционален количеству системных компонентов, в которые требуется внести изменения. Если этот показатель возрастает, система требует многочисленных изменений.

3. **Среднее время, необходимое на реализацию изменений**. Не следует путать этот показатель с предыдущим, хотя они тесно связаны. Здесь учитывается не время анализа системы по выявлению причин сбоев, а время реализации изменений и их документирования, которое зависит от сложности программного кода. Увеличение этого показателя означает сложность сопровождения.

4. **Количество незавершенных запросов на изменения**. С возрастанием количества таких запросов затрудняется сопровождение системы.

Для определения стоимости сопровождения используется предварительная информация о запросах на изменения и прогнозирование относительно удобства сопровождения системы. В решении этого вопроса большинству менеджеров поможет также интуиция и опыт. В модели определения стоимости COCOMO 2, предполагается, что для оценки стоимости сопровождения понадобятся сведения об усилиях, потраченных на понимание существующего кода системы и на разработку нового.

10.5. Реинжиниринг программного обеспечения

Реинжиниринг – это повторная реализация наследуемой системы в целях повышения удобства ее эксплуатации и сопровождения. В это понятие входят разные процессы, среди которых назовем повторное документирование системы, ее реорганизацию и реструктуризацию, перевод системы на один из более современных языков программирования, модификацию и модернизацию структуры и системных данных. При этом функциональность системы и ее архитектура остаются неизменными.

С технической точки зрения реинжиниринг – это решение "второго сорта" проблемы системной эволюции. Если учесть, что архитектура системы не изменяется, то сделать централизованную систему распределенной представляется делом довольно сложным. Обычно нельзя изменить язык программирования старых систем на объектно-ориентированные языки (например, Java или C++). Эти ограничения вводятся для сохранения архитектуры системы.

Однако с коммерческой точки зрения реинжиниринг часто принимается за единственный способ сохранения наследуемых систем в эксплуатации. Другие подходы к эволюции системы либо слишком дорогостоящие, либо рискованные. Чтобы понять причины такой позиции, следует рассмотреть проблемы, связанные с наследуемыми системами.

Код эксплуатируемых в настоящее время программных систем чрезвычайно огромен. В 1990 году Улрич (Ulrich) насчитал 120 млрд. строк исходного кода эксплуатируемых в то время программ. При этом большинство программ были написаны на языке COBOL, который лучше всего подходит для обработки данных в деловой сфере, и на языке FORTRAN. У этих языков достаточно ограниченные возможности в плане структуризации программ, а FORTRAN к тому же отличается ограниченной поддержкой структурирования данных.

Несмотря на постоянную замену подобных систем, многие из них все еще используются. С 1990 года отмечается резкое возрастание использования вычислительной техники в деловой сфере. При грубом подсчете можно говорить о 250 млрд. строк исходного кода, которые нуждаются в сопровождении. Большинство создано отнюдь не с помощью объектно-ориентированных языков

программирования, многие из них функционируют все еще на мэйнфреймах.

Программных систем настолько много, что говорить о полной замене или радикальной реструктуризации их в большинстве организаций не приходится. Сопровождение старых систем действительно стоит дорого, однако реинжиниринг может продлить время их существования. Реинжиниринг систем выгоден в том случае, если система обладает определенной коммерческой ценностью, но дорога в сопровождении. С помощью реинжиниринга совершенствуется системная структура, создается новая документация и облегчается сопровождение системы.

По сравнению с более радикальными подходами к совершенствованию систем реинжиниринг имеет два преимущества.

1. **Снижение рисков.** При повторной разработке ПО существуют большие риски – высока вероятность ошибок в системной спецификации и возникновения проблем во время разработки системы. Реинжиниринг снижает эти риски.

2. **Снижение затрат**. Себестоимость реинжиниринга значительно ниже, чем разработка нового программного обеспечения. В статье приводится пример системы, эксплуатируемой в коммерческой структуре, повторная разработка которой оценивалась в 50 млн. долларов. Для этой системы был успешно выполнен реинжиниринг стоимостью всего 12 млн. долларов. Приведенные цифры типичны: считается, что реинжиниринг в четыре раза дешевле, чем повторная разработка системы.

Реинжиниринг ПО тесно связан с реинжинирингом деловых процессов. Последний означает преобразование бизнес-процессов для снижения количества излишних видов деятельности и повышения эффективности делового процесса. Обычно реинжиниринг бизнес-процессов предполагает внедрение новых программ для поддержки деловых процессов или модификацию существующих программ, при этом наследуемые системы существенно зависят от делового процесса. Такую зависимость следует выявлять как можно раньше и устранять, прежде чем начнется планирование каких-либо изменений в самом бизнес-процессе. Поэтому решение о реинжиниринге ПО может возникнуть, если наследуемую систему не удается

адаптировать к новым деловым процессам путем изменений в обычном сопровождении системы.

Основное различие между реинжинирингом и новой разработкой системы связано со стартовой точкой начала работы над системой. При реинжиниринге вместо написания системной спецификации "с нуля" старая система служит основой для разработки спецификации новой системы. В статье традиционная разработка ПО названа *разработкой вперед* (forward engineering), чтобы подчеркнуть различие между ней и реинжинирингом. Это различие проиллюстрировано на рис. 10.1. Традиционная разработка начинается с этапа создания системной спецификации, за которой следует проектирование и реализация новой системы. Реинжиниринг основывается на существующей системе, которая разработчиками изучается и преобразуется в новую.

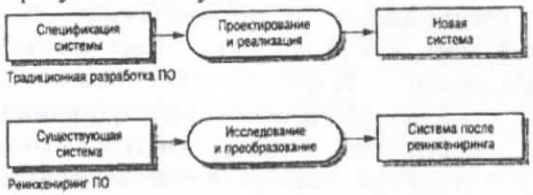

Рис. 10.1. Традиционная разработка и реинжиниринг ПО

На рис. 10.1 показан возможный процесс реинжиниринга. В начале этого процесса имеем наследуемую систему, а в результате – структурированную и заново скомпонованную версию той же системы. Перечислим основные этапы этого процесса.

1. **Перевод исходного кода**. Конвертирование программы со старого языка программирования на современную версию того же языка либо на другой язык.
2. **Анализ программ**. Документирование структуры и функциональных возможностей программ на основе их анализа.
3. **Модификация структуры программ**. Анализируется и модифицируется управляющая структура программ с целью сделать их более простыми и понятными.
4. **Разбиение на модули.** Взаимосвязанные части программ группируются в модули; там, где возможно, устраняется избыточность. В некоторых случаях изменяется структура системы.

5. **Изменение системных данных**. Данные, с которыми работает программа, изменяются с тем, чтобы соответствовать нововведениям.

Рис. 10.2. Процесс реинжиниринга

При реинжиниринге программ необязательно проходить все стадии, показанные на рис. 10.2. Например, не всегда нужно переводить исходный код, если язык программирования, на котором написана программа, все еще поддерживается разработчиком компилятора. Если реинжиниринг проводится с помощью автоматизированных средств, то не обязательно восстанавливать документацию на программу. Изменение системных данных необходимо, если в результате реинжиниринга изменяется их структура. Однако реструктуризация данных в процессе реинжиниринга требуется всегда.

Стоимость реинжиниринга обычно определяется объемом выполненных работ. На рис. 10.3 показано несколько различных подходов к процессу реинжиниринга и динамика изменения стоимости работ для этих подходов.

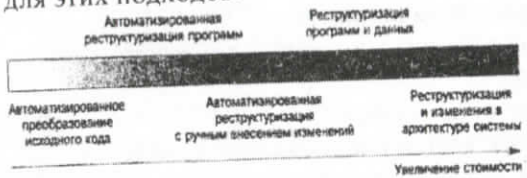

Рис. 10.3. Стоимость реинжиниринга

Кроме объема выполняемых работ, есть и другие факторы, обусловливающие стоимость реинжиниринга.

1. **Качество программного обеспечения, которое подвергается реинжинирингу.** Чем ниже качество программ и их документации (если она есть в наличии), тем выше стоимость реинжиниринга.
2. **Наличие средств поддержки процесса реинжиниринга**. Обычно реинжиниринг экономически выгоден, если применяются CASE-средства для автоматизированного внесения изменений в программы.
3. **Объем необходимого преобразования данных**. Стоимость процесса реинжиниринга возрастет при увеличении объема преобразуемых данных.
4. **Наличие необходимых специалистов**. Если персонал, который занимается сопровождением системы, не может выполнить реинжиниринг, это также может стать причиной повышения стоимости процесса. Вновь привлеченные специалисты потратят много времени на изучение системы.

Основным недостатком реинжиниринга принято считать то, что с его помощью систему можно улучшить только до определенной степени. Например, с помощью реинжиниринга невозможно функционально-ориентированную систему сделать объектно-ориентированной. Основные архитектурные изменения или полную реструктуризацию программ невозможно выполнить автоматически, что также увеличивает стоимость реинжиниринга. И, несмотря на то что реинжиниринг поможет улучшить сопровождение системы, все равно она будет намного хуже в сопровождении, чем новая, созданная с помощью современных методов инженерии ПО.

Контрольные вопросы:

1) Какие стратегические подходы к процессу модернизации ПО существует?
2) Что такое сопровождение?
3) Перечистите виды сопровождения системы.
4) Какие ключевые факторы определяют стоимость разработки и сопровождения и могут привести к подорожанию сопровождения?
5) Что такое реинжиниринг?

Ключевые слова: сопровождение программного обеспечения, эволюция системной архитектуры, реинжиниринг программного обеспечения, стабильность команды разработчиков, ответственность согласно контракту, квалификация специалистов, возраст и структура программы, реинжиниринг, снижение рисков, снижение затрат, перевод исходного кода, анализ программ, модификация структуры программ, разбиение на модули, изменение системных данных.

Keywords: *software support, the evolution of the system architecture, software reengineering, the stability of the development team, responsible under the contract, qualified specialists, age and structure of the program, re-engineering, risk reduction, cost reduction, the translation of the source code, program analysis, modification of the structure of programs modularization, the change of system data.*

Kalit so'zlar: *dasturiy ta'minot kuzatuvi, tizim arhitekturasi evolyutsiyasi, dasturiy ta'minot reinjiniringi, ishlabchiqaruvchilar guruhining barqarorligi, shartnomaga muvofiq javobgarlik, mutahassislarning malakasi, dasturning yoshi va strukturasi, chiqim va havf kamayishi, dasturlar tahlili, modullarga bo'linish, tizim ma'lumotlarini o'zgartirish.*

Упражнения

1. В каких случаях необходимо заменить старое ПО новым вместо его реинжиниринга?
2. Сравните управляющие структуры (циклы и условные операторы) в двух любых известных вам языках программирования. Кратко опишите процесс перевода управляющих структур с одного языка на другой.
3. Напишите ряд указаний по определению модулей в неструктурированной программе.
4. Какие трудности возникнут при переносе данных из СУБД одного типа в СУБД другого типа (например, из иерархической базы данных в реляционную или из реляционной в объектно-ориентированную)?
6. Объясните, почему невозможно восстановить системную спецификацию путем автоматического анализа исходного кода системы.

7. Приведите примеры для описания проблем, связанных с нарушением данных при их чистке.
8. Проблема 2000 года (когда даты представлены с помощью двух цифр) стала одной из основных причин для многих организаций внести коррективы в сопровождение программ. Как это повлияло на процесс изменения данных?

ЛИТЕРАТУРА

1. Sh.Mirziyoyev "Erkin va farovon, demokratik O'zbekiston davlatini birgalikda barpo etamiz".Tosh.O 'zbekiston-2016.
2. O'zbekiston Respublikasining "Ta'lim to'g'risida"gi qonuni// Barkamol avlod O'zbekiston taraqqiyotining poydevori.-T.: "Sharq" nashriyot-matbaa konserni, 1997. 22-32b.
3. O'zbekiston Respublikasi "Kadrlar tayyorlash Milliy dasturi"//Barkamol avlod O'zbekiston taraqqiyotining poydevori.-T.: "Sharq" nashriyot-matbaa kontserni, 1997.-V.31-61.
4. O'zbekiston Respublikasi Prezidentining 2017-yil 7-fevraldagi PF-4947 sonli "O'zbekiston Respublikasini yanada rivojlantirishning *Harakatlar* strategiyasi to'g'risida"gi farmoni// Xalq so'zi gazetasi. 2017 yil 8-fevral,№ 28 .
5. O'zbekiston Respublikasi Vazirlar Mahkamasining 2002 yil 6 iyundagi 200-sonli "Kompyuterlashtirishni yanada rivojlantirish va axborot kommunikatsiya texnologiyalarini joriy etish chor-tadbirlari to'g'risida"gi qarori.
6. Соколов, А.Г. Природа экранного творчества: психологические закономерности. М.: Изд.А.Дворников, 2011. 638 с.
7. Гурвист М. Использование Macромедиа Флэш MX / М. Гурвист, Л. Мак-Кейб ; спест. изд-е. – М. : Вилъямс, 2013. – 704 с.
8. Г. С. Иванова. Технология программирования, 2012, М.:МГТУ Баумана
9. Чепмен Н. цифровые технологии мулътимедиа / Н. Чепмен, Д. Чепмен; 2-е изд. – М. : Вилъямс, 2013. – 624 с.
10. Черткова Е.А. Введение в программную инженерию. Учебное пособие, Московский государственный машиностроительный университет, 2013
11. «Программная инженерия. Учебник для вузов. 5-е издание обновленное и дополненное» Профессиональная литература, Блог компании Издательский дом «Питер», 2012.
12. Элиенс А. Принципы объектно-ориентированной разработки программ, 2-е изд. – М.: Издат. дом "Вильямс", 2002.
13. Бен-Ари М. Языки программирования. Практический сравнительный анализ. – М.: Мир, 2000.

14.Абдикеев Н.М. Когнитивная бизнес-аналитика: учебник. - М.: Инфра-М, 2011.
15. Агальцов В. П. Базы данных/Кн. 1: Локальные базы данных: учебник для вузов. – М.: Форум: ИНФРА-М, 2009. - 349 с.
16. Агальцов В. П. Базы данных / Кн. 2: Распределенные и удаленные базы данных: учебник для вузов. – М.: Форум : ИНФРА-М, 2009. - 270 с.
17. Гуйдо А. В. Технологии программирования [Электронный ресурс]: учеб. пособие/ Под ред. Б. М. Суховилова; Южно-Урал. гос. ун-т, каф. Информатика; ЮУрГУ, Челябинск: изд-во ЮУрГУ, 2010. Электрон. текстовые данные http://www.lib.susu.ac.ru/ftd?base=SUSU_METHOD &key=000428010.
18. Ekaterina M.Lavrischeva. Assemblling Paradigms of Programming in Software Engineering. – 2016, 9, 2016. – p. 296-317, http://www.scrip.org/journal/jsea, http://dx.do.org/ 10.4236/jsea.96021
19. Lavrischeva E. Generative and composition programming: aspects of developing software system families.- Cybernetics and Systems Analysis, Springer Volume 49, Issue 1 (2013), Page 110-123.
20. Лаврищева Е.М. Теория и практика фабрик программных продуктов .- Кибернетика и системный анализ .- № 6, 2011.- с.145-158.
21. Павловская Т. А. С/С++. Структурное и объектно-ориентированное программирование: практикум /Т. А. Павловская, Ю. А. Щупак. – СПб. и др.: Питер, 2010. - 347 с.
22. Симонович С.В. Информатика. Базовый курс: учебник для вузов, 2-е изд., – СПб: Питер, 2010.
23. Лаврищева Е.М. Программная инженерия. Парадигмы, технологии, CASE- средства.- 2 издание книги в п.10.- Юрайт, 2016.-280 с.
24. Лаврищева К.М. Розвиток идей академіка В.М.Глушкова з питань технологии програмування.–Київ, Вісник НАН України, 2013, №9.– с. 66–83.

25. Островський А. И. Подход к обеспечению взаимодействия программных сред JAVA и Ms.Net. – Проблеми програмування, 2011.–№ 2.–с. 37–44.
26. Радецький І. О. Один з підходів до забезпечення взаємодії середовищ MS.Net і Eclipse // Проблеми програмування, № 2, 2011.– с. 45–52.
27. Лавріщева К. М. Онтологічне подання ЖЦ ПС для загальної лінії виробництва програмних продуктів. – Праці конференції TAAPSD'2013, "Теоретичні і приклані аспекти побудови програмних систем", 25 травня – 2 червня 2013. – с. 81–90.
28. Лавріщева К.М. Підхід до формального подання онтології життєвого циклу програмних систем. – Вісник КГУ, серія фіз.-мат.наук. – 2013. – №4. – С. 94 – 100
29. Бабенко Л.П. Онтологический подход к спецификации свойств программных систем и их компонент//Кибернетика и системный анализ.–2009.–№1.–с.30–37.
30. Зінькович В.М. Онтологічне моделювання предметної області з проблематикою e–science' // Проблеми програмування. – 2011 – № 3. – С. 91–99
31. Островський А. И. Подход к обеспечению взаимодействия программных сред JAVA и Ms.Net. – Проблеми програмування, 2011.–№ 2.–с. 37–44.
32. Радецький І. О. Один з підходів до забезпечення взаємодії середовищ MS.Net і Eclipse // Проблеми програмування, № 2, 2011.– с. 45–52.
33. Лаврищева Е.М. Теория объектно-компонентного моделирования программных систем. препринт ИСП РАН, 2016, 48 с. www.ispras.ru/preprints/docs/prep_29_ 2016_pdf
34. Е.М..Лаврищева. Компонентная теория и коллекция технологий для разработки индустриальных приложений из готовых ресурсов, Труды 4-научно-практической конференции «Актуальные проблемы системной и программной инженерии», АПСПИ-2015, 20-21мая 2015, с .101-119.
35. Романов Е.Л. Архитектура и прикладные протоколы клиент-серверных приложений. Электронный учебно-методический комплекс. № ОФЭРНиО: 21514 [Электронный

ресурс]: режим доступа – http://dispace.edu.nstu.ru/didesk/course/show/5379 (дата обращения: 01.07.2016).

36. Разработка приложений на C# с использованием СУБД PostgreSQL: учебное пособие / И.А. Васюткина, Г.В. Трошина, М.И. Бычков, С.А. Менжулин. – Новосибирск: Изд-во НГТУ, 2015. – 143 с. ISBN 978-5-7782-2699-9.

37. Романов Е.Л. Теория языков программирования и методы трансляции. Электронное учебное пособие для студентов 3 курса дневного и заочного отделений направления 230100. Новосибирский государственный технический университет, № гос. регистрации 0321100595., 2011, 150 с.

ОГЛАВЛЕНИЕ