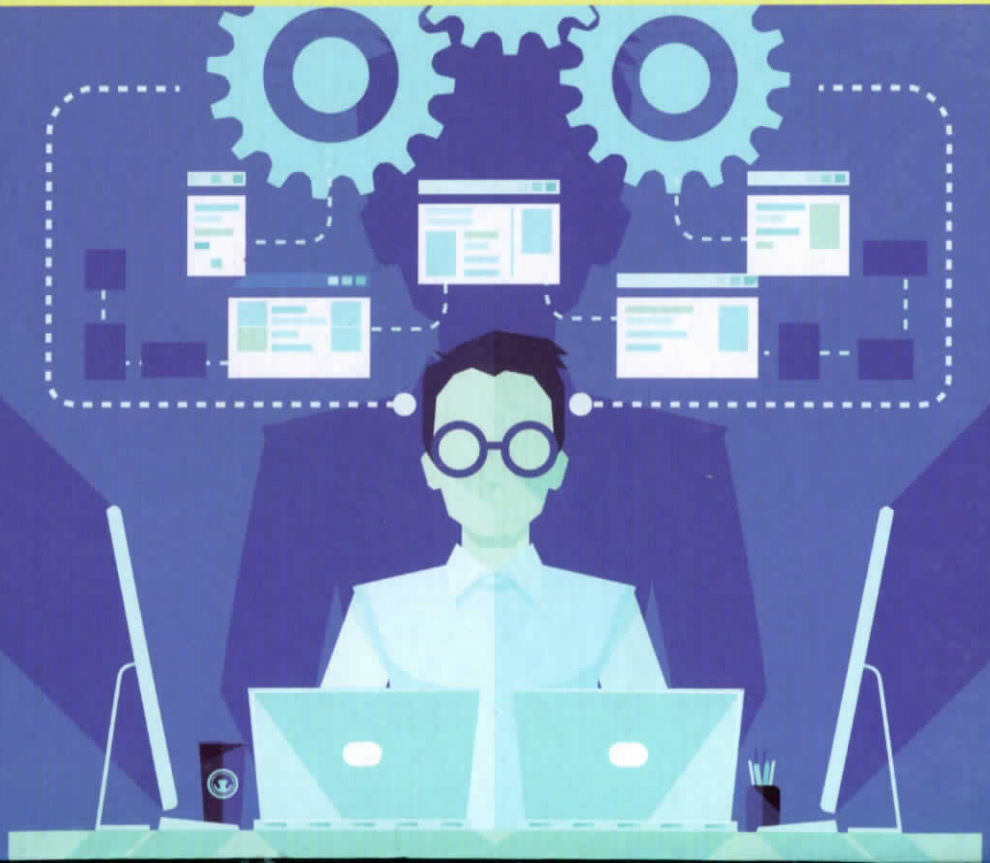
А.Х.Нишанов, О.Рузибаев,
М.Ю.Дощанова, А.М.Матьякубова

ПРОГРАММНЫЙ ИНЖИНИРИНГ



МИНИСТЕРСТВО ПО РАЗВИТИЮ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И КОММУНИКАЦИЙ РЕСПУБЛИКИ УЗБЕКИСТАН

ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ ИМЕНИ МУХАММАДА АЛ-ХОРАЗМИЙ

А.Х.Нишанов, О.Рузибаев,
М.Ю.Дощанова, А.М.Матьякубова.

ПРОГРАММНЫЙ ИНЖИНИРИНГ

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

Ташкент – 2019

УДК: 004.4(075.8)
ББК: 32.973

А.Х.Нишанов, О.Рузибаев, М.Ю.Дощанова, А.М.Матьякубова. Программный инжиниринг. Учебное пособие. – Т.: «Aloqachi», 2019. – 168 с.

ISBN 978–9943–5897–9–7

В учебном пособии рассмотрены понятия программного инжиниринга, программное обеспечение, требования к программному обеспечению, процесс создания программного обеспечения, объектно – ориентированное проектирование.

Темы посвящены проектированию пользовательского интерфейса, принципам проектированию пользовательского интерфейса, представлению различной информации в интерфейсе, оценивание интерфейса. В учебном пособии описаны все понятия и навыки приведенные в Государственном образовательном стандарте.

Учебное пособие предназначено для учащихся по направлению 5330500 – Компьютерный инжиниринг и для тех кто интересуется программным инжинирингом.

Рекомендуется к печати по решению учебно – методического совета ТУИТ имени Мухаммада ал-Хоразмий.

УДК: 004.4(075.8)
ББК: 32.973

Рецензенты:
Ф. Эргашев;
Т.А.Кучкаров.

ISBN 978–9943–5897–9–7

ВВЕДЕНИЕ

Программная инженерия (промышленное программирование) 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.

В конце 60-х – начале 70-х годов прошлого века произошло событие, которое вошло в историю как первый кризис программирования. Событие состояло в том, что стоимость программного обеспечения стала приближаться к стоимости аппаратуры («железа»), а динамика роста этих стоимостей позволяла прогнозировать, что к середине 90-годов все человечество будет заниматься разработкой программ для компьютеров. Тогда и заговорили о программной инженерии или технологии промышленного программирования как о некоторой дисциплине, целью которой является сокращение стоимости программ.

С тех пор программная инженерия прошла достаточно бурное развитие. Этапы развития программной инженерии можно выделять по-разному. Каждый этап связан с появлением (или осознанием) очередной проблемы и нахождением путей и способов решения этой проблемы.

Программная инженерия как некоторое направление возникло и формировалось под давлением роста стоимости создаваемого программного обеспечения.

Программная инженерия прошла несколько этапов развития, в процессе которых были сформулированы фундаментальные принципы и методы разработки программных продуктов. Основной принцип программной инженерии состоит в том, что программы создаются в результате выполнения нескольких взаимосвязанных этапов (анализ требований, проектирование, разработка, внедрение, сопровождение), составляющих жизненный цикл программного продукта. Фундаментальными методами проектирования и разработки являются модульное, структурное и объектно-ориентированное проектирование и программирование.

Главная цель этой области знаний – сокращение стоимости и сроков разработки программ.

1 ГЛАВА. ВВЕДЕНИЕ В ПРОГРАММНЫЙ ИНЖИНИРИНГ

1.1. Понятие программного инжиниринга

На сегодняшний день наш мир все больше зависит от систем, построенных на основе вычислительной техники. Технические устройства включают в себя элементы вычислительной техники и соответствующего управляющего программного обеспечения (ПО) в той или иной форме. В этих системах стоимость ПО порой составляет большую часть общей стоимости. А также, стоимостные показатели отраслей, занимающихся производством ПО, становятся определяющими для экономики – как национальной, так и международной.

Целью инженерии программного обеспечения является эффективное создание программных систем. Программное обеспечение абстрактно и нематериально. Оно не имеет физической природы, отвергает физические законы и не подвергается обработке производственными процессами. Такой упрощенный взгляд на ПО показывает, что не существует физических ограничений на потенциальные возможности программных систем. С другой стороны, отсутствие материального наполнения порой делает ПО чрезвычайно сложным и, следовательно, трудным для понимания "объектом".

Инженерия программного обеспечения – она сравнительно молодая научная дисциплина. Термин *software engineering* был впервые предложен в 1968 году на конференции, посвященной так называемому кризису программного обеспечения. Этот кризис был вызван появлением мощной (по меркам того времени) вычислительной техники третьего поколения. Новая техника позволяла воплотить в жизнь не реализуемые ранее программные приложения. В результате программное обеспечение достигло размеров и уровня сложности, намного превышающих аналогичные показатели у программных систем, реализованных на вычислительной технике предыдущих поколений.

Неформальный подход, применявшийся ранее к построению программных систем, недостаточен для разработки больших систем. На реализацию крупных программных проектов иногда уходили многие годы. Стоимость таких проектов многократно возрастала по сравнению с первоначальными расчетами, сами программные

системы получались ненадежными, сложными в эксплуатации и сопровождении. Разработка программного обеспечения оказалась в кризисе. Стоимость аппаратных средств постепенно снижалась, тогда как стоимость ПО стремительно возрастала. Возникла необходимость в новых технологиях и методах управления комплексными сложными проектами разработки больших программных систем.

Эти методы составили часть инженерии программного обеспечения и в настоящее время широко используются, хотя, конечно же, не являются универсальными. Сохраняется много проблем в процессе разработки сложных программных систем, на решение которых затрачивается много времени и средств.

Программная реализация многих программных проектов сталкивается с подобными проблемами, это дает право некоторым специалистам утверждать, что современная технология создания программного обеспечения находится в состоянии хронического недуга.

Ну а с другой стороны, возрастает как объем производства программного обеспечения, так и его сложность. Объединение вычислительной и коммуникационной техники ставит новые требования перед специалистами по программному обеспечению.

Причина возникновения проблем при разработке программных систем, как и то, что многие компании, занимающиеся производством ПО, не уделяют должного внимания эффективному применению современных методов, разработанных в рамках инженерии программного обеспечения.

По сравнению с 2005 годом сделан огромный поворот и развитие инженерии программного обеспечения значительно улучшило современное ПО. Теперь мы лучше понимаем те процессы, которые определяют развитие программных систем. Разработаны эффективные методы спецификации ПО, его разработки и внедрения. Новые средства и технологии позволяют значительно уменьшить усилия и затраты на создание больших и сложных программных систем.

Многие специалисты по программному обеспечению могут гордиться такими достижениями. Без современного сложного ПО было бы невозможно освоение космического пространства, не существовало бы Internet и современных телекоммуникаций, а все транспортные средства и виды транспортировки были бы более опасными и дорогостоящими. Инженерия программного обеспечения

достигла многого за свою пока еще короткую жизнь, и мы уверены, что ее значение как зрелой научной дисциплины еще более возрастет в XXI столетии.

1.2. Вопросы и ответы об инженерии программного обеспечения

Этот параграф мы построили в виде ответов на основные вопросы инженерии программного обеспечения и отображающие мой собственный взгляд на эту дисциплину. Использовали стиль "списка FAQ" (Frequently Asked Questions – часто задаваемые вопросы). Этот формат обычно применяется в группах новостей Internet, предлагая новичкам ответы на часто задаваемые вопросы. Такой подобный подход будет эффективен в качестве краткого введения в предмет инженерии программного обеспечения.

Вопросы и ответы, часто задаваемые приведены в этой таблице (табл. 1).

Таблица 1. Часто задаваемые вопросы об инженерии программного обеспечения

Вопрос	Ответ
Что такое программное обеспечение (ПО)?	Это компьютерные программы и соответствующая документация. Программные продукты разрабатываются или по частному заказу, или для продажи на рынке программных продуктов
Что такое инженерия программного обеспечения?	Это инженерная дисциплина, охватывающая все аспекты разработки программного обеспечения
В чем различие между инженерией программного обеспечения и компьютерной наукой?	Компьютерная наука – это теоретическая дисциплина, охватывающая все стороны вычислительных систем, включая аппаратные средства и программное обеспечение; инженерия программного обеспечения – практическая дисциплина создания и сопровождения программных систем
В чем различие между инженерией программного обеспечения и системотехникой?	Системотехника охватывает все аспекты разработки вычислительных систем (включая создание аппаратных средств и ПО) и соответствующие технологические процессы. Технологии инженерии программного обеспечения являются частью этих процессов

Что такое технологический процесс создания ПО?	Это совокупность процессов, ведущих к созданию или развитию программного обеспечения
Что такое модель технологического процесса создания ПО?	Формализованное упрощенное представление технологического процесса создания ПО
Какова структура затрат на создание ПО?	Примерно 60% от общих затрат на создание ПО занимают затраты непосредственно на разработку ПО и 40% – на его тестирование и отладку. Для программных продуктов, разрабатываемых по заказу, стоимость тестирования и отладки часто превышает стоимость разработки продукта
Что такое методы инженерии программного обеспечения?	Это структурные решения, предназначенные для разработки ПО и включающие системные модели, формализованные нотацию и правила проектирования, а также способы управления процессом создания ПО
Что такое CASE (Computer-Aided Software Engineering – автоматизированное проектирование и создание ПО)?	Это программные системы, предназначенные для автоматизации процесса создания ПО. CASE-средства часто используются в качестве основы методов инженерии программного обеспечения
Каковы признаки качественного ПО?	Программные продукты должны удовлетворять требованиям функциональности и эффективности (с точки зрения пользователя), а также быть надежными, удобными в эксплуатации и иметь возможности для модернизации
Какие основные проблемы стоят перед специалистами по программному обеспечению?	Проблема наследования ранее созданного ПО, проблема все возрастающей разнородности программных систем и проблема, порожденная требованием уменьшения времени на создание ПО

1.3. Понятие программного обеспечения

Многие сравнивают термин *программное обеспечение* с компьютерными программами. Это весьма ограниченное представление. Программное обеспечение – это не только программы, но и вся документация, а также конфигурационные данные, необходимые для корректной работы программ. Программные системы состоят из совокупности программ, файлов

конфигурации, необходимых для установки этих программ, и документации, которая описывает структуру системы, а также содержит инструкции для пользователей, объясняющие работу с системой, и часто адрес Web узла, где пользователь может найти самую последнюю информацию о данном программном продукте.

Все специалисты по программному обеспечению разрабатываю программы, т.е. такое ПО, которое можно продать потребителю. Программы делятся на два типа.

1. *Общие программные продукты* - это автономные программные системы, которые созданы компанией по производству ПО и продаются на открытом рынке программных продуктов любому потребителю, способному их купить. Иногда их называют "коробочным ПО". Примерами этого типа программных продуктов могут служить системы управления базами данных, текстовые процессоры, графические пакеты и средства управления проектами.
2. *Программные продукты, созданные на заказ* - это программные системы, которые создаются по заказу определенного потребителя. Такое ПО разрабатывается специально для данного потребителя согласно заключенному контракту. Программные продукты этого типа включают системы управления для электронных устройств, системы поддержки определенных производственных или бизнес-процессов, системы управления воздушным транспортом и т.п.

Различие между этими типами программных продуктов заключается в том, что при создании общих программных продуктов спецификация требований на них разрабатывается компанией производителем. Для заказных программных продуктов спецификация обычно разрабатывается организацией, покупающей данный продукт. Спецификация необходима разработчикам ПО для создания любого программного продукта.

1.4. Понятие инженерия программного обеспечения

Инженерия программного обеспечения – это инженерная дисциплина, которая охватывает все аспекты создания ПО от начальной стадии разработки системных требований через создание ПО до его использования. В этом определении присутствует две ключевые фразы.

1. "Инженерная дисциплина". Инженеры – это те специалисты, которые выполняют практическую работу. Они применяют теоретические построения, методы и средства там, где это необходимо, но делают это выборочно и всегда пытаются найти решение задачи, даже если не существует подходящей теории или методов решения. Специалисты-инженеры также всегда понимают, что они должны работать в организационных и финансовых рамках заключенных контрактов, т.е. ищут решение поставленной перед ними задачи с учетом условий контракта.
2. "Все аспекты создания программного обеспечения". Инженерия программного обеспечения не рассматривает *технические* аспекты создания ПО – в ее ведении такие вопросы, как управление проектом создания ПО и разработка средств, методов и теорий, необходимых для создания программных систем.

Специалисты по программному обеспечению адаптируют существующие методы инженерии ПО к решению своих задач, и зачастую это оказывается наиболее эффективным способом построения высококачественных программных систем.

Инженерия программного обеспечения предоставляет всю необходимую информацию для выбора наиболее подходящего метода для множества практических задач. Вместе с тем творческий неформальный подход в определенных обстоятельствах также может быть эффективным. Например, при разработке программных систем электронной коммерции в Internet требуется неформальный подход в сочетании ПО и графического эскизного проектирования.

Компьютерная наука охватывает теорию и методы построения вычислительных и программных систем, тогда как инженерия программного обеспечения акцентирует внимание на практических проблемах разработки ПО. Знание компьютерной науки необходимо специалистам по программному обеспечению так же, как знание физики – инженерам-электронщикам.

Инженерия программного обеспечения должна основываться на фундаменте компьютерной науки, но это не так. Иногда специалисты по программному обеспечению используют подходы, применимые для решения только конкретной задачи. Элегантные методы компьютерной науки не всегда можно применить к реальным сложным задачам, требующим программного решения.

Контрольные вопросы

1. Что такое программное обеспечение?
2. Что такое инженерия программного обеспечения?
3. В чем различие между инженерией программного обеспечения и компьютерной наукой?
4. В чем различие между инженерией программного обеспечения и системотехникой?
5. Что такое технологический процесс создания ПО?
6. Что такое модель технологического процесса создания ПО?
7. Какова структура затрат на создание ПО?
8. Что такое методы инженерии программного обеспечения?
9. Каковы признаки качественного ПО?

2 ГЛАВА. ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1. Понятие жизненного цикла программного обеспечения

Жизненный цикл программного обеспечения (Software Life Cycle Model) — это период времени, который начинается с момента принятия решения о создании программного продукта и заканчивается в момент его полного изъятия из эксплуатации. Этот цикл — процесс построения и развития ПО.

Жизненный цикл – это модель создания и использования программной системы. Он отражает различные состояния программной системы, начиная с момента возникновения необходимости в этой программной системе и принятия решения о ее создании и заканчивая полным изъятием программной системы из эксплуатации.

Международный стандарт ISOIES 12207 регламентирует структуру жизненного цикла, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания программного обеспечения. По этому стандарту жизненный цикл программного обеспечения базируется на трех группах процессов:

- основные процессы жизненного цикла, то есть приобретение, поставка, разработка, эксплуатация и сопровождение;
- вспомогательные процессы, обеспечивающие выполнение основных процессов, то есть документирование, верификация, аттестация, оценка качества и другие;
- организационные процессы, то есть управление проектами, создание инфраструктуры проекта и обучение.

Разработка включает в себя все работы по созданию программного обеспечения в соответствии с заданными требованиями. Сюда включаются оформление проектной и эксплуатационной документации, подготовка материалов, необходимых для проверки работоспособности и качества программных продуктов.

Основные этапы процесса разработки:

анализ требований заказчика;

проектирование;

реализация (программирование).

Процесс эксплуатации включает в себя работы по внедрению программного обеспечения в эксплуатацию, в том числе конфигурирование рабочих мест, обучение персонала, локализация проблем эксплуатации и устранение причин их возникновения, модификация программного обеспечения в рамках установленного регламента и подготовка предложений по модернизации системы.

Каждый процесс характеризуется определенными задачами и методами их решения, а также исходными данными и результатами.

Жизненный цикл программного обеспечения носит, как правило, итерационный характер, то есть реализуются этапы, начиная с самых ранних, которые циклически повторяются в соответствии с изменением требований внешних условий и введением ограничений.

2.2. Модели Жизненного цикла программного обеспечения

Жизненный цикл можно представить в виде моделей. В настоящее время наиболее распространенными являются: *каскадная*, *инкрементная* (*поэтапная модель с промежуточным контролем*) и *спиральная*модели жизненного цикла.

Каскадная модель (англ. *waterfall model*) — модель процесса разработки программного обеспечения, жизненный цикл которой выглядит как поток, последовательно проходящий фазы анализа требований, проектирования. реализации, тестирования, интеграции и поддержки.

Процесс разработки реализуется с помощью упорядоченной последовательности независимых шагов. Модель предусматривает, что каждый последующий шаг начинается после полного завершения выполнения предыдущего шага. На всех шагах модели выполняются вспомогательные и организационные процессы и работы, включающие управление проектом, оценку и управление качеством, верификацию и аттестацию, менеджмент конфигурации, разработку документации. В результате завершения шагов формируются промежуточные продукты, которые не могут изменяться на последующих шагах (рис. 2.1).

Жизненный цикл традиционно разделяют на следующие основные этапы:

1. Анализ требований,
2. Проектирование,

3. Кодирование (программирование),
4. Тестирование и отладка,
5. Эксплуатация и сопровождение.

Рис. 2.1. Каскадная модель жизненного цикла

Достоинства модели:

- стабильность требований в течение всего жизненного цикла разработки;
- на каждой стадии формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- определенность и понятность шагов модели и простота её применения;
- выполняемые в логической последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие ресурсы (денежные. материальные и людские).

Каскадная модель хорошо зарекомендовала себя при построении относительно простых ПО, когда в самом начале разработки можно достаточно точно и полно сформулировать все требования к продукту.

Недостатки модели:

- сложность чёткого формулирования требований и невозможность их динамического изменения на протяжении пока идет полный жизненный цикл;
- низкая гибкость в управлении проектом;
- последовательность линейной структуры процесса разработки, в результате возврат к предыдущим шагам для решения

возникающих проблем приводит к увеличению затрат и нарушению графика работ;

- непригодность промежуточного продукта для использования;
- невозможность гибкого моделирования уникальных систем;
- позднее обнаружение проблем, связанных со сборкой, в связи с одновременной интеграцией всех результатов в конце разработки;
- недостаточное участие пользователя в создании системы — в самом начале (при разработке требований) и в конце (во время приёмочных испытаний);
- пользователи не могут убедиться в качестве разрабатываемого продукта до окончания всего процесса разработки. Они не имеют возможности оценить качество, т.к.нельзя увидеть готовый продукт разработки;
- у пользователя нет возможности постепенно привыкнуть к системе. Процесс обучения происходит в конце жизненного цикла, когда ПО уже запущено в эксплуатацию;
- каждая фаза является предпосылкой для выполнения последующих действий, что превращает такой метод в рискованный выбор для систем, не имеющих аналогов, т.к. он не поддается гибкому моделированию.

Реализовать Каскадную модель жизненного цикла затруднительно ввиду сложности разработки ПС без возвратов к предыдущим шагам и изменения их результатов для устранения возникающих проблем.

Ограничение области применения каскадной модели определяется её недостатками. Её использование наиболее эффективно в следующих случаях:

1. при разработке проектов с четкими, неизменяемыми в течение *жизненного цикла* требованиями, понятными реализацией и техническими методиками;
2. при разработке проекта, ориентированного на построение системы или продукта такого же типа, как уже разрабатывались разработчиками ранее;
3. при разработке проекта, связанного с созданием и выпуском новой версии уже существующего продукта или системы;

4. при разработке проекта, связанного с переносом уже существующего продукта или системы на новую платформу;
5. при выполнении больших проектов, в которых задействовано несколько больших команд разработчиков.

Инкрементная модель (англ. *increment* — увеличение, приращение) подразумевает разработку программного обеспечения с линейной последовательностью стадий, но в несколько инкрементов (версий), т.е. с запланированным улучшением продукта за все время пока Жизненный цикл разработки ПО не подойдет к окончанию (рис. 2.2).

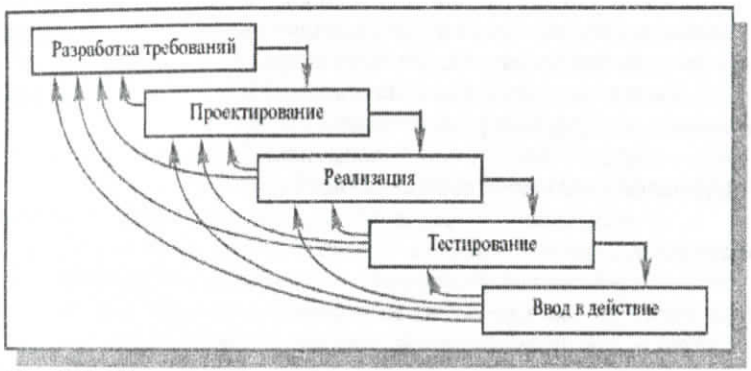


Рис. 2.2. Поэтапная модель с промежуточным контролем

Разработка программного обеспечения ведется итерациями с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных этапах, время жизни каждого из этапов растягивается на весь период разработки.

В начале работы над проектом определяются все основные требования к системе, подразделяются на более и менее важные. После чего выполняется разработка системы по принципу приращений, так, чтобы разработчик мог использовать данные, полученные в ходе разработки ПО. Каждый инкремент должен добавлять системе определенную функциональность. При этом выпуск начинают с компонентов с наивысшим приоритетом. Когда части системы определены, берут первую часть и начинают её детализировать, используя для этого наиболее подходящий процесс. В то же время можно уточнять требования и для других частей,

которые в текущей совокупности требований данной работы были заморожены. Если есть необходимость, можно вернуться позже к этой части. Если часть готова, она поставляется клиенту, который может использовать её в работе. Это позволит клиенту уточнить требования для следующих компонентов. Затем занимаются разработкой следующей части системы. Ключевые этапы этого процесса — простая реализация подмножества требований к программе и совершенствование модели в серии последовательных релизов до тех пор, пока не будет реализовано ПО во всей полноте.

Жизненный цикл данной модели характерен при разработке сложных и комплексных систем, для которых имеется четкое видение (как со стороны заказчика, так и со стороны разработчика) того, что собой должен представлять конечный результат. Разработка версиями ведется в силу разного рода причин:

- отсутствия у заказчика возможности сразу профинансировать весь дорогостоящий проект;
- отсутствия у разработчика необходимых ресурсов для реализации сложного проекта в сжатые сроки;
- требований поэтапного внедрения и освоения продукта конечными пользователями. Внедрение всей системы сразу может вызвать у её пользователей неприятие и только "затормозить" процесс перехода на новые технологии. Образно говоря, они могут просто "не переварить большой кусок, поэтому его надо измельчить и давать по частям".

Достоинства и недостатки этой модели (стратегии) такие же, как и у каскадной (классической модели жизненного цикла). Но в отличие от классической стратегии заказчик может раньше увидеть результаты. Уже по результатам разработки и внедрения первой версии он может незначительно изменить требования к разработке, отказаться от нее или предложить разработку более совершенного продукта с заключением нового договора.

Достоинства:

- затраты, которые получаются в связи с изменением требований пользователей, уменьшаются, повторный анализ и совокупность документации значительно сокращаются по сравнению с каскадной моделью;
- легче получить отзывы от клиента о проделанной работе — клиенты могут озвучить свои комментарии в отношении готовых

частей и могут видеть, что уже сделано. Т.к. первые части системы являются прототипом системы в целом.

- у клиента есть возможность быстро получить и освоить программное обеспечение — клиенты могут получить реальные преимущества от системы раньше, чем это было бы возможно с каскадной моделью.

Недостатки модели:

- менеджеры должны постоянно измерять прогресс процесса. в случае быстрой разработки не стоит создавать документы для каждого минимального изменения версии;
- структура системы имеет тенденцию к ухудшению при добавлении новых компонентов — постоянные изменения нарушают структуру системы. Чтобы избежать этого требуется дополнительное время и деньги на рефакторинг. Плохая структура делает программное обеспечение сложным и дорогостоящим для последующих изменений. А прерванный Жизненный цикл ПО приводит еще к большим потерям.

Схема не позволяет оперативно учитывать возникающие изменения и уточнения требований к ПО. Согласование результатов разработки с пользователями производится только в точках, планируемых после завершения каждого этапа работ, а общие требования к ПО зафиксированы в виде технического задания на всё время её создания. Таким образом, пользователи зачастую получаю ПП, не удовлетворяющий их реальным потребностям.

Спиральная модель: Жизненный цикл — на каждом витке спирали выполняется создание очередной версии продукта, уточняются требования проекта, определяется его качество и планируются работы следующего витка. Особое внимание уделяется начальным этапам разработки — анализу и проектированию, где реализуемость тех или иных технических решений проверяется и обосновывается посредством создания прототипов (рис. 2.3).

Данная модель представляет собой процесс разработки программного обеспечения, сочетающий в себе как проектирование, так и постадийное прототипировнаие с целью сочетания преимуществ восходящей и нисходящей концепции, делающая упор на начальные этапы жизненного цикла: анализ и проектирование. Отличительной особенностью этой модели является специальное внимание рискам, влияющим на организацию жизненного цикла.

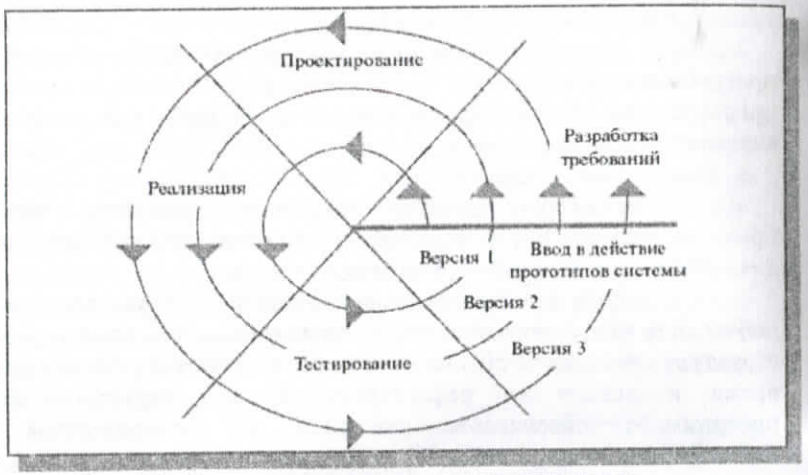


Рис. 2.3. Спиральная модель жизненного цикла

На этапах анализа и проектирования реализуемость технических решений и степень удовлетворения потребностей заказчика проверяется путем создания прототипов. Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы. Это позволяет уточнить требования, цели и характеристики проекта, определить качество разработки, спланировать работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который удовлетворяет действительным требованиям заказчика и доводится до реализации.

Жизненный цикл на каждом витке спирали — могут применяться разные модели процесса разработки ПО. В конечном итоге на выходе получается готовый продукт. Модель сочетает в себе возможности модели прототипирования и водопадной модели. Разработка итерациями отражает объективно существующий спиральный цикл создания системы. Неполное завершение работ на каждом этапе позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. Главная задача — как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Достоинства модели:

- позволяет быстрее показать пользователям системы работоспособный продукт, тем самым, активизируя процесс уточнения и дополнения требований;
- допускает изменение требований при разработке программного обеспечения, что характерно для большинства разработок, в том числе и типовых;
- в модели предусмотрена возможность гибкого проектирования, поскольку в ней воплощены преимущества каскадной модели, и в то же время разрешены итерации по всем фазам этой же модели;
- позволяет получить более надежную и устойчивую систему. По мере развития программного обеспечения ошибки и слабые места обнаруживаются и исправляются на каждой итерации;
- эта модель разрешает пользователям активно принимать участие при планировании, анализе рисков, разработке, а также при выполнении оценочных действий;
- уменьшаются риски заказчика. Заказчик может с минимальными для себя финансовыми потерями завершить развитие неперспективного проекта;
- обратная связь по направлению от пользователей к разработчикам выполняется с высокой частотой и на ранних этапах модели, что обеспечивает создание нужного продукта высокого качества.

Недостатки модели:

- если проект имеет низкую степень риска или небольшие размеры, модель может оказаться дорогостоящей. Оценка рисков после прохождения каждой спирали связана с большими затратами;
- Жизненный цикл модели имеет усложненную структуру, поэтому может быть затруднено её применение разработчиками, менеджерами и заказчиками;
- спираль может продолжаться до бесконечности, поскольку каждая ответная реакция заказчика на созданную версию может порождать новый цикл, что отдаляет окончание работы над проектом;
- большое количество промежуточных циклов может привести к необходимости в обработке дополнительной документации;
- использование модели может оказаться дорогостоящим и даже недопустимым по средствам, т.к. время. затраченное на

Также кроме этих требований двух уровней, применяется еще более детализированное описание системы – *проектная системная спецификация* (software design specification), которая может служить мостом между этапом разработки требований и этапом проектирования системы. Три перечисленных вида требований можно определить следующим образом.

1. *Пользовательские требования* – описание на естественном языке функций, выполняемых системой, и ограничений, накладываемых на нее.
2. *Системные требования* – детализированное описание системных функций и ограничений, которое иногда называют функциональной спецификацией. Она служит основой для заключения контракта между покупателем системы и разработчиками ПО.
3. *Проектная системная спецификация* – обобщенное описание структуры программной системы, которое будет основой для более детализированного проектирования системы и ее последующей реализации. Эта спецификация дополняет и детализирует спецификацию системных требований.

Все пользовательские требования пишутся для заказчика ПО и для лица, заключающего контракт на разработку программной системы, они могут не иметь детальных технических знаний по разрабатываемой системе (рис. 3.1).

Спецификация системных требований предназначена для руководящего технического состава компании-разработчика и для менеджеров проекта. Она также необходима заказчику ПО и субподрядчикам по разработке. Эти оба документа также предназначены для конечных пользователей программной системы. Наконец, проектная системная спецификация является документом, который ориентирован на разработчиков ПО.

Рис. 3.1. Различные типы спецификаций требований и их читатели

3.2. Функциональные и нефункциональные требования

Требования предъявляемые к программной системе часто классифицируются как функциональные, нефункциональные и требования предметной области.

1. *Функциональные требования.* Это перечень сервисов, которые должна выполнять система, система реагирует на те или иные входные данные, поведение в определенных ситуациях и т.д. В некоторых случаях указывается, что система не должна делать.
2. *Нефункциональные требования.* Описывают характеристики системы и ее окружения, а не поведение системы. Здесь также может быть приведен перечень ограничений, накладываемых на действия и функции, выполняемые системой. Они включают *временные* ограничения, ограничения на процесс разработки системы, стандарты и т.д.
3. *Требования предметной области.* Характеризуют ту предметную область, где будет эксплуатироваться система. Эти требования могут быть функциональными и нефункциональными.

В действительности четкой границы между этими типами требований не существует. Например, пользовательские требования,

касающиеся безопасности системы, можно отнести к нефункциональным.

Однако при более детальном рассмотрении такое требование можно отнести к функциональным, поскольку оно порождает необходимость включения в систему средства авторизации пользователя. Поэтому, рассматривая далее эти виды требований, мы должны всегда помнить, что данная классификация в значительной степени искусственна.

3.3. Функциональные требования

Функциональные требования описывают поведение системы и сервисы, которые она выполняет, и зависят от типа разрабатываемой системы и от потребностей пользователей. Если функциональные требования оформлены как пользовательские, они описывают системы в обобщенном виде. В противоположность этому функциональные требования, оформленные как системные, описывают систему максимально подробно, включая ее входные и выходные данные, исключения и т.д.

Функциональные требования для программных систем могут быть описаны разными способами. Рассмотрим для примера функциональные требования к библиотечной системе университета, предназначенной для заказа книг и документов из других библиотек.

1. Пользователь должен иметь возможность проводить поиск необходимых ему книг и документов или по всему множеству доступных каталожных баз данных или по определенному их подмножеству.
2. Система должна предоставлять пользователю подходящее средство просмотра библиотечных документов.
3. Каждый заказ должен быть снабжен уникальным идентификатором (ORDER_ID), который копируется в формуляр пользователя для постоянного хранения.

Функциональные пользовательские требования определяют свойства, которыми должна обладать система. Они взяты из документа, содержащего пользовательские требования, и показывают, что функциональные требования могут быть описаны с разным уровнем детализации.

Проблемы, возникающие при разработке систем, связаны с неточностью и "размытостью" спецификаций требований.

Естественно, разработчики интерпретируют требования, допускающие разное толкование, так, чтобы систему было проще реализовать. Но это толкование может не совпадать с ожиданиями заказчику. Такая ситуация приводит к разработке новых требований и внесению изменений в систему. Это, в свою очередь, ведет к задержке сдачи готовой системы и ее удорожанию.

Рассмотрим второе требование к библиотечной системе. Библиотечная система может предоставлять документы в широком спектре форматов.

В требовании подразумевается, что система должна предоставить средства для просмотра документов в любом формате. Но поскольку это условие четко не выписано, разработчики в случае дефицита времени могут использовать простое средство для просмотра текстовых документов и настаивать на том, что именно такое решение следует из данного требования.

В принципе спецификация функциональных требований должна быть комплексной и непротиворечивой. Комплексность подразумевает описание (определение) всех системных сервисов. Непротиворечивость означает отсутствие несовместимых и взаимоисключающих определений сервисов.

На практике для больших и сложных систем крайне трудно разработать комплексную и непротиворечивую спецификацию функциональных требований. Причина кроется частично в сложности самой разрабатываемой системы, а частично – в несогласованных опорных точках зрения на то, что должна делать система.

Эта несогласованность может не проявиться на этапе первоначального формулирования требований – для ее выявления необходим более глубокий анализ спецификаций. Когда несогласованность системных функций проявится на каком-либо этапе жизненного цикла программы, в системную спецификацию придется внести соответствующие изменения.

3.4. Нефункциональные требования

Нефункциональные требования не связаны непосредственно с функциями, выполняемыми системой. Они связаны с такими интегрированными свойствами системы, как надежность, время ответа или размер системы. Нефункциональные требования могут определять ограничения на систему, например на пропускную

способность устройств ввода-вывода, или форматы данных, используемых в системном интерфейсе.

Нефункциональные требования относятся к системе в целом, а не к отдельным ее средствам. Это означает, что они более значимы и критичны, чем отдельные функциональные требования. Ошибка, допущенная в функциональном требовании, может снизить качество системы, ошибка в нефункциональных требованиях может сделать систему неработоспособной.

Нефункциональные требования могут относиться не только к самой программной системе: одни могут относиться к технологическому процессу создания ПО, другие – содержать перечень стандартов качества, накладываемых на процесс разработки.

Нефункциональные требование может быть указано, что проектирование системы должно выполняться только определенными CASE-средствами, и приведено описание процесса проектирования, которому необходимо следовать.

Нефункциональные требования отображают пользовательские потребности; при этом они основываются на бюджетных ограничениях, учитывают организационные возможности компании разработчика и возможность взаимодействия разрабатываемой системы с другими программными и вычислительными системами, а также такие внешние факторы, как правила техники безопасности, законодательство о защите интеллектуальной собственности и т.п. На рис. 3.2 показана классификация нефункциональных требований.

Все нефункциональные требования, показанные на рис. 3.2, можно разбит на три большие группы.

1. *Требования к продукту.* Описывают эксплуатационные свойства программного продукта. Сюда относятся требования к производительности системы, объему необходимой памяти, надежности (определяет частоту возможных сбоев в системе), переносимости системы на разные компьютерные платформы и удобству эксплуатации.
2. *Организационные требования.* Отображают политику и организационные процедуры заказчика и разработчика ПО. Они включают стандарты разработки программного продукта, требования к реализации ПО (т.е. к языку программирования и методам проектирования), выходные требования, которые определяют сроки изготовления программного продукта, и сопутствующую документацию.

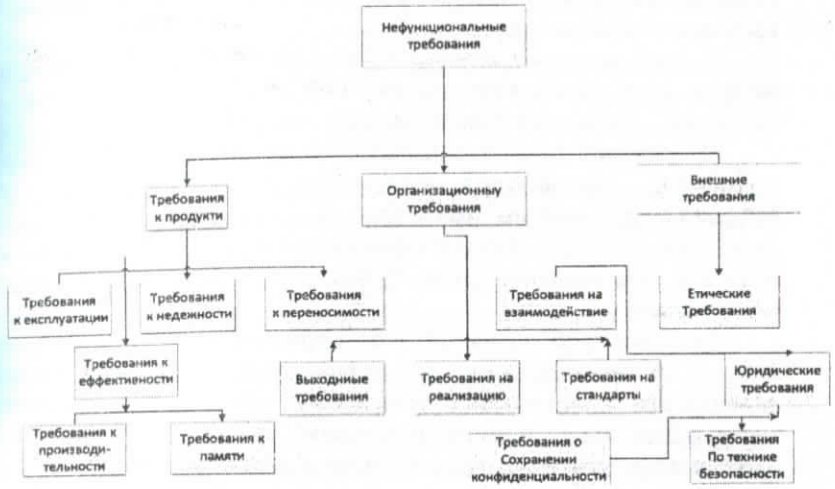


Рис. 3.2. Типы нефункциональных требований

Внешние требования. Учитывают факторы, внешние по отношению к разрабатываемой системе и процессу ее разработки. Они включают требования, определяющие взаимодействие данной системы с другими системами, юридические требования, следование которым гарантирует, что система будет разрабатываться и функционировать в рамках существующего законодательства, а также этические требования. Последние должны гарантировать, что система будет приемлемой для пользователей или заказчика.

Нефункциональные требования часто вступают в конфликт с другими требованиями, предъявляемыми системе. Например, в соответствии с одним из системных требований размер системы не должен превышать 4 Мбайт, поскольку она должна полностью поместиться в постоянное запоминающее устройство ограниченной емкости.

Другое требование обязывает использовать для написания системы язык программирования Ada, который часто применяется для создания критических систем реального времени. Но, допустим, откомпилированная системная программа, написанная на языке Ada,

занимает более 4 Мбайт. Итак, одновременное выполнение этих требований невозможно.

В этой ситуации следует отказаться от одного из требований. Можно или применить другой язык программирования, или увеличить объем памяти, выделяемый для системы.

В принципе функциональные и нефункциональные требования в документе, описывающем требования к системе, должны быть разнесены по разным разделам. Но на практике это условие выполнить непросто. Если нефункциональные требования поместить отдельно от функциональных, будет трудно проследить взаимосвязи между ними.

Если все требования собраны в одном списке, сложно провести анализ функциональных и нефункциональных требований в отдельности и определить требования, относящиеся к системе в целом. Вид представления требований в одном документе также существенно зависит от типа специфицируемой системы.

Но в любом случае должны быть выделены требования, описывающие интеграционные свойства системы. Для этого их можно поместить в отдельный раздел либо каким-нибудь другим способом отделить от остальных требований.

3.5. Требования предметной области

Требования предметной области отображают условия, в которых будет эксплуатироваться программная система. Они могут быть представлены в виде новых функциональных требований, в виде ограничений на уже сформулированные функциональные требований или в виде указаний, как система должна выполнять вычисления.

Требования предметной области очень важны, поскольку отображают ту предметную область, где будет использоваться данная система. Невыполнение требований предметной области может привести к выходу системы из строя. В качестве примера рассмотрим требования к библиотечной системе.

1. Стандартный пользовательский интерфейс, предоставляющий доступ ко всем библиотечным базам данных, должен основываться на стандарте Z39.50.
2. Для обеспечения авторских прав некоторые документы должны быть удалены из системы сразу после получения. Для этого, в зависимости от желания пользователя,

эти документы могут быть распечатаны или на локальном системном сервере, или на сетевом принтере.

Первое требование является ограничением на системное функциональное требование. Оно указывает, что пользовательский интерфейс к базам данных должен быть реализован согласно соответствующему библиотечному стандарту.

Второе требование является внешним и направлено на выполнение закона об авторских правах, применяемого к библиотечным материалам. Из этого требования вытекает, что система должна иметь средство "удалить_на_печать", применяемое автоматически для некоторых типов библиотечных документов.

3.6. Пользовательские требования

Пользовательские требования к системе должны описывать функциональные и нефункциональные системные требования так, чтобы они были понятны даже пользователю, не имеющему специальных технических знаний.

Пользовательские требования должны определять только внешнее поведение системы, избегая по возможности определения структурных характеристик системы. Пользовательские требования должны быть написаны естественным языком с использованием простых таблиц, а также наглядных и понятных диаграмм.

При описании требований на естественном языке могут возникнуть различные проблемы.

1. *Отсутствие четкости изложения.* Иногда нелегко изложить какую-либо мысль естественным языком четко и недвусмысленно, не сделав при этом текст многословным и трудночитаемым.
2. *Смешение требований.* В пользовательских требованиях отсутствует четкое разделение на функциональные и нефункциональные требования, на системные цели и проектную информацию.
3. *Объединение требований.* Несколько различных требований к системе могут описываться как единое пользовательское требование.

занимает более 4 Мбайт. Итак, одновременное выполнение этих требований невозможно.

В этой ситуации следует отказаться от одного из требований. Можно или применить другой язык программирования, или увеличить объем памяти, выделяемый для системы.

В принципе функциональные и нефункциональные требования в документе, описывающем требования к системе, должны быть разнесены по разным разделам. Но на практике это условие выполнить непросто. Если нефункциональные требования поместить отдельно от функциональных, будет трудно проследить взаимосвязи между ними.

Если все требования собраны в одном списке, сложно провести анализ функциональных и нефункциональных требований в отдельности и определить требования, относящиеся к системе в целом. Вид представления требований в одном документе также существенно зависит от типа специфицируемой системы.

Но в любом случае должны быть выделены требования, описывающие интеграционные свойства системы. Для этого их можно поместить в отдельный раздел либо каким-нибудь другим способом отделить от остальных требований.

3.5. Требования предметной области

Требования предметной области отображают условия, в которых будет эксплуатироваться программная система. Они могут быть представлены в виде новых функциональных требований, в виде ограничений на уже сформулированные функциональные требований или в виде указаний, как система должна выполнять вычисления.

Требования предметной области очень важны, поскольку отображают ту предметную область, где будет использоваться данная система. Невыполнение требований предметной области может привести к выходу системы из строя. В качестве примера рассмотрим требования к библиотечной системе.

1. Стандартный пользовательский интерфейс, предоставляющий доступ ко всем библиотечным базам данных, должен основываться на стандарте Z39.50.
2. Для обеспечения авторских прав некоторые документы должны быть удалены из системы сразу после получения. Для этого, в зависимости от желания пользователя,

эти документы могут быть распечатаны или на локальном системном сервере, или на сетевом принтере.

Первое требование является ограничением на системное функциональное требование. Оно указывает, что пользовательский интерфейс к базам данных должен быть реализован согласно соответствующему библиотечному стандарту.

Второе требование является внешним и направлено на выполнение закона об авторских правах, применяемого к библиотечным материалам. Из этого требования вытекает, что система должна иметь средство "удалить_на_печать", применяемое автоматически для некоторых типов библиотечных документов.

3.6. Пользовательские требования

Пользовательские требования к системе должны описывать функциональные и нефункциональные системные требования так, чтобы они были понятны даже пользователю, не имеющему специальных технических знаний.

Пользовательские требования должны определять только внешнее поведение системы, избегая по возможности определения структурных характеристик системы. Пользовательские требования должны быть написаны естественным языком с использованием простых таблиц, а также наглядных и понятных диаграмм.

При описании требований на естественном языке могут возникнуть различные проблемы.

1. *Отсутствие четкости изложения.* Иногда нелегко изложить какую-либо мысль естественным языком четко и недвусмысленно, не сделав при этом текст многословным и трудночитаемым.
2. *Смешение требований.* В пользовательских требованиях отсутствует четкое разделение на функциональные и нефункциональные требования, на системные цели и проектную информацию.
3. *Объединение требований.* Несколько различных требований к системе могут описываться как единое пользовательское требование.

3.7. Системные требования

Системные требования – это более детализированное описание пользовательских требований. Они обычно служат основой для заключения контракта на разработку программной системы и поэтому должны представлять максимально полную спецификацию системы в целом. Системные требования также используются в качестве отправной точки на этапе проектирования системы.

Спецификация системных требований может строиться на основе различных системных моделей, таких, как объектная модель или модель потоков данных.

В принципе системные требования определяют, что должна делать система, не показывая при этом механизма ее реализации. Но, с другой стороны, для полного описания системы требуется детализированная информация о ней, которая по возможности должна включать всю информацию о системной архитектуре. На то существует ряд причин.

1. Первоначальная архитектура системы помогает структурировать спецификацию требований. Системные требования должны описывать подсистемы, из которых состоит разрабатываемая система.
2. В большинстве случаев разрабатываемая система должна взаимодействовать с уже существующими системами. Это накладывает определенные ограничения на архитектуру новой системы.
3. В качестве внешнего системного требования может выступать условие использования для разрабатываемой системы специальной архитектуры.

Спецификации системных требований часто пишутся естественным языком. Но использование естественного языка может породить определенные проблемы при написании детализированной спецификации. Применение естественного языка подразумевает, что те, кто пишет спецификацию, и те, кто ее читает, одни и те же слова и выражения понимают одинаково. Однако на самом деле это не так, поскольку естественному языку присуща определенная размытость понятий. Вследствие этого одно и то же требование может трактоваться разными людьми по-разному.

Чтобы избежать подобных проблем, разработаны методы описания требований, которые структурируют спецификацию и уменьшают размытость определений.

3.8. Структурированный язык спецификаций

Структурированный язык спецификаций - это сокращенная форма естественного языка, предназначенная для написания спецификации требований. Достоинством такого подхода к написанию спецификаций является то, что он сохраняет выразительность и понятность естественного языка и вместе с тем формализует описание требований.

Структурированность языка проявляется в использовании специальной терминологии, а также шаблонов для описания системных требований. Структурированный язык может включать языковые конструкции, взятые из языков программирования.

Для описания системных требований часто разрабатываются специальные формы и шаблоны. Они должны учитывать, на основе чего строится спецификация: на основе объектов, управляемых системой, на основе функций, выполняемых системой, или на основе событий, обрабатываемых системой.

Стандартные формы, используемые для специфицирования функциональных требований, должны содержать следующую информацию.

1. Описание функции или объекта.
2. Описание входных данных и их источники.
3. Описание выходных данных с указанием пункта их назначения.
4. Указание, что необходимо для выполнения функции.
5. Если это спецификация функции, необходимо описание предварительных условий, которые должны выполняться перед вызовом функции, и описание заключительного условия, которое должно быть выполнено после завершения выполнения функции.
6. Описание побочных эффектов.

Использование структурированного языка снимает некоторые проблемы, присущие спецификациям, написанным естественным языком, поскольку снижает "вариабельность" спецификации и более

Основные архитектурные стили

- *Клиент-серверная модель*. Разделение системы на два приложения – клиент и сервер. При работе клиент посылает запросы на обслуживание серверу
- *Компонентная архитектура*. Деление системы на компоненты, которые могут быть повторно использованы и не зависят друг от друга. Каждый компонент снабжается известным интерфейсом для коммуникаций
- *Многоуровневая архитектура*. Разделение функций приложения на группы (уровни), которые организованы в виде стекового набора
- *Шина сообщений*. Система, которая может посылать и передавать информационные сообщения в определённом формате по общему коммуникационному каналу. Благодаря этому организуется взаимодействие систем без указаний конкретных получателей сообщений
- *Многозвенная архитектура*. Разделение функций подобно многоуровневой архитектуре, но группировка происходит не только на логическом, и на физическом уровне – отдельным группам соответствует отдельный компьютер (сервер, кластер)
- *Объектно-ориентированная архитектура*. Представление системы в виде набора взаимодействующих объектов
- *Выделенное представление*. Выделение в системе отдельных групп функций для взаимодействия с пользователями и обработки данных
- *Архитектура, ориентированная на сервисы*. Каждый компонент системы представлен в виде независимого сервиса, предоставляющего свои функции по стандартному протоколу

Важно понимать, что стили не исключают совместное применение, особенно при проектировании сложных систем. По сути, архитектурные стили допускают группировку согласно направлению решаемых ими задач. Например, Шина сообщений и Архитектура, ориентированная на сервисы – это коммуникационные стили (т. е. их задача – способ организации коммуникации между отдельными компонентами).

Архитектура ПО обычно содержит несколько видов, которые аналогичны различным типам чертежей в строительстве зданий. Виды являются экземплярами точки зрения, где точка зрения

существует для описания архитектуры с точки зрения заданного множества заинтересованных лиц.

Архитектурный вид состоит из 2 компонентов:

- Элементы
- Отношения между элементами

Архитектурные виды можно поделить на 3 основных типа:

- *Модульные виды* (англ. *module views*) — показывают систему как структуру из различных программных блоков.
- *Компоненты-и-коннекторы* (англ. *component-and-connector views*) — показывают систему как структуру из параллельно запущенных элементов (компонентов) и способов их взаимодействия (коннекторов).
- *Размещение* (англ. *allocation views*) — показывает размещение элементов системы во внешних средах.

Примеры модульных видов:

- *Декомпозиция* (англ. *decomposition view*) — состоит из модулей в контексте отношения «является подмодулем»
- *Использование* (англ. *uses view*) — состоит из модулей в контексте отношения «использует» (т.е. один модуль использует сервисы другого модуля)
- *Вид уровней* (англ. *layered view*) — показывает структуру, в которой связанные по функциональности модули объединены в группы (уровни)
- *Вид классов/обобщений* (англ. *class/generalization view*) — состоит из классов, связанные через отношения «наследуется от» и «является экземпляром»

Примеры видов компонентов-и-коннекторов:

- *Процессный вид* (англ. *process view*) — состоит из процессов, соединённых операциями коммуникации, синхронизации и/или исключения
- *Параллельный вид* (англ. *concurrency view*) — состоит из компонентов и коннекторов, где коннекторы представляют собой «логические потоки»
- *Вид обмена данными* (англ. *shared-data (repository) view*) — состоит из компонентов и коннекторов, которые создают, сохраняют и получают постоянные данные

- *Вид клиент-сервер* (англ. *client-server view*) — состоит из взаимодействующих клиентов и серверов и коннектором между ними (например, протоколов и общих сообщений)

Примеры видов размещения:

- *Развертывание* (англ. *deployment view*) — состоит из программных элементов, их размещения на физических носителях и коммуникационных элементов
- *Внедрение* (англ. *implementation view*) — состоит из программных элементов и их соответствия файловым структурам в различных средах (разработческой, интеграционной и т.д.)
- *Распределение работы* (англ. *work assignment view*) — состоит из модулей и описания того, кто ответственен за внедрение каждого из них

4.2. Диаграммы UML

Унифицированный язык моделирования (UML) является стандартным инструментом для создания "чертежей" программного обеспечения. С помощью UML можно визуализировать, специфицировать, конструировать и документировать артефакты программных систем.

UML пригоден для моделирования любых систем: от информационных систем масштаба предприятия до распределенных Web-приложений и даже встроенных систем реального времени. Это очень выразительный язык, позволяющий рассмотреть систему со всех точек зрения, имеющих отношение к ее разработке и последующему развертыванию. Несмотря на обилие выразительных возможностей, этот язык прост для понимания и использования. Изучение UML удобнее всего начать с его концептуальной модели, которая включает в себя три основных элемента: базовые строительные блоки, правила, определяющие, как эти блоки могут сочетаться между собой, и некоторые общие механизмы языка.

Несмотря на свои достоинства, UML - это всего лишь язык; он является одной из составляющих процесса разработки программного обеспечения, и не более того. Хотя UML не зависит от моделируемой реальности, лучше всего применять его, когда процесс моделирования основан на рассмотрении прецедентов использования,

является итеративным и пошаговым, а сама система имеет четко выраженную архитектуру.

UML - это язык для визуализации, специфицирования, конструирования и документирования артефактов программных систем.

Язык состоит из словаря и правил, позволяющих комбинировать входящие в него слова и получать осмысленные конструкции. В *языке моделирования* словарь и правила ориентированы на концептуальное и физическое представление системы. Язык моделирования, подобный UML, является стандартным средством для составления "чертежей" программного обеспечения.

Моделирование необходимо для понимания системы. При этом единственной модели никогда не бывает достаточно. Напротив, для понимания любой нетривиальной системы приходится разрабатывать большое количество взаимосвязанных моделей. В применении к программным системам это означает, что необходим язык, с помощью которого можно с различных точек зрения описать представления архитектуры системы на протяжении цикла ее разработки.

Словарь и правила такого языка, как UML, объясняют, как создавать и читать хорошо определенные модели, но ничего не сообщают о том, какие модели и в каких *случаях* нужно создавать. Это задача всего процесса разработки программного обеспечения. Хорошо организованный процесс должен подсказать вам, какие требуются артефакты, какие ресурсы необходимы для их создания, как можно использовать эти артефакты, чтобы оценить выполненную работу и управлять проектом в целом.

С точки зрения большинства программистов, размышления по поводу реализации проекта почти эквивалентны написанию для него кода. Вы думаете - значит, вы кодируете. И действительно, некоторые вещи лучше всего выражаются непосредственно в коде на каком-либо языке программирования, поскольку текст программы - это самый простой и короткий путь для записи алгоритмов и выражений.

Но даже в таких случаях программист занимается моделированием, хотя и неформально. Он может, допустим, записать набросок идеи на доске или на салфетке. Однако такой подход чреват неприятностями. Во-первых, обмен мнениями по поводу концептуальной модели возможен только тогда, когда все участники

дискуссии говорят на одном языке. Как правило, при разработке проектов компаниям приходится изобретать собственные языки, и новичку непросто догадаться, о чем идет речь. Во-вторых, нельзя получить представление об определенных аспектах программных систем без модели, выходящей за границы текстового языка программирования. Так, назначение иерархии классов можно, конечно, понять, если внимательно изучить код каждого класса, но воспринять всю структуру сразу и целиком не получится. Аналогично изучение кода системы не позволит составить целостное представление о физическом распределении и возможных миграциях объектов в Web-приложении. В-третьих, если автор кода никогда не воплощал в явной форме задуманные им модели, эта информация будет навсегда утрачена, если он сменит место работы. В лучшем случае ее можно будет лишь частично воссоздать исходя из реализации.

Использование UML позволяет решить третью проблему: явная модель облегчает общение.

Некоторые особенности системы лучше всего моделировать в виде текста, другие - графически. На самом деле во всех интересных системах существуют структуры, которые невозможно представить с помощью одного лишь языка программирования. UML - графический язык, что позволяет решить вторую из обозначенных проблем.

UML - это не просто набор графических символов. За каждым из них стоит хорошо определенная семантика (см. "The Unified Modeling Language Reference Manual"). Это значит, что модель, написанная одним разработчиком, может быть однозначно интерпретирована другим - или даже инструментальной программой. Так решается первая из перечисленных выше проблем.

В данном контексте *специфицирование* означает построение точных, недвусмысленных и полных моделей. UML позволяет специфицировать все существенные решения, касающиеся анализа, проектирования и реализации, которые должны приниматься в процессе разработки и развертывания системы программного обеспечения.

UML не является языком визуального программирования, но модели, созданные с его помощью, могут быть непосредственно переведены на различные языки программирования. Иными словами, UML-модель можно отобразить на такие языки, как Java, C++, Visual Basic, и даже на таблицы реляционной базы данных или устойчивые

объекты объектно-ориентированной базы данных. Те понятия, которые предпочтительно передавать графически, так и представляются в UML; те же, которые лучше описывать в текстовом виде, выражаются с помощью языка программирования.

Такое отображение модели на язык программирования позволяет осуществлять прямое проектирование: генерацию кода из модели UML в какой-то конкретный язык. Можно решить и обратную задачу: реконструировать модель по имеющейся реализации. Обратное проектирование не представляет собой ничего необычного. Если вы не закодировали информацию в реализации, то эта информация теряется при прямом переходе от моделей к коду. Поэтому для обратного проектирования необходимы как инструментальные средства, так и вмешательство человека. Сочетание прямой генерации кода и обратного проектирования позволяет работать как в графическом, так и в текстовом представлении, если инструментальные программы обеспечивают согласованность между обоими представлениями.

Помимо прямого отображения в языки программирования UML в силу своей выразительности и однозначности позволяет непосредственно исполнять модели, имитировать поведение систем и контролировать действующие системы.

Компания, выпускающая программные средства, помимо исполняемого кода производит и другие артефакты, в том числе следующие:

- требования к системе;
- архитектуру;
- проект;
- исходный код;
- проектные планы;
- тесты;
- прототипы;
- версии, и др.

В зависимости от принятой методики разработки выполнение одних работ производится более формально, чем других. Упомянутые артефакты - это не просто поставляемые составные части проекта; они необходимы для управления, для оценки результата, а также в качестве средства общения между членами коллектива во время разработки системы и после ее развертывания.

UML позволяет решить проблему документирования системной архитектуры и всех ее деталей, предлагает язык для формулирования требований к системе и определения тестов и, наконец, предоставляет средства для моделирования работ на этапе планирования проекта и управления версиями.

Язык UML предназначен прежде всего для разработки программных систем. Его использование особенно эффективно в следующих областях:

- информационные системы масштаба предприятия;
- банковские и финансовые услуги;
- телекоммуникации;
- транспорт;
- оборонная промышленность, авиация и космонавтика;
- розничная торговля;
- медицинская электроника;
- наука;
- распределенные Web-системы.

Сфера применения UML не ограничивается моделированием программного обеспечения. Его выразительность позволяет моделировать, скажем, документооборот в юридических системах, структуру и функционирование системы обслуживания пациентов в больницах, осуществлять проектирование аппаратных средств.

Для понимания UML необходимо усвоить его концептуальную модель, которая включает в себя три составные части: основные строительные блоки языка, правила их сочетания и некоторые общие для всего языка механизмы. Усвоив эти элементы, вы сумеете читать модели на UML и самостоятельно создавать их -вначале, конечно, не очень сложные. По мере приобретения опыта в работе с языком вы научитесь пользоваться и более развитыми его возможностями.

Словарь языка UML включает три вида строительных блоков:

- сущности;
- отношения;
- диаграммы.

Сущности - это абстракции, являющиеся основными элементами модели. Отношения связывают различные сущности; диаграммы группируют представляющие интерес совокупности сущностей.

В UML имеется четыре типа сущностей:

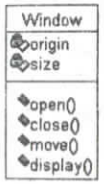
- структурные;

- поведенческие;
- группирующие;
- аннотационные.

Сущности являются основными объектно-ориентированными блоками языка. С их помощью можно создавать корректные модели.

Структурные сущности - это имена существительные в моделях на языке UML. Как правило, они представляют собой статические части модели, соответствующие концептуальным или физическим элементам системы. Существует несколько разновидностей структурных сущностей.

Класс (Class) - это описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. Класс реализует один или несколько интерфейсов. Графически класс изображается в виде прямоугольника, в котором обычно записаны его имя, атрибуты и операции, как показано на рисунке.



Классы

Интерфейс (Interface) - это совокупность операций, которые определяют сервис (набор услуг), предоставляемый классом или компонентом. Таким образом, интерфейс описывает видимое извне поведение элемента. Интерфейс может представлять поведение класса или компонента полностью или частично; он определяет только спецификации операций (сигнатуры), но никогда - их реализации. Графически интерфейс изображается в виде круга, под которым пишется его имя, как показано на рисунке. Интерфейс редко существует сам по себе - обычно он присоединяется к реализующему его классу или компоненту.

Интерфейсы

Кооперация (Collaboration) определяет взаимодействие; она представляет собой совокупность ролей и других элементов, которые,

работая совместно, производят некоторый кооперативный эффект, не сводящийся к простой сумме слагаемых. Кооперация, следовательно, имеет как структурный, так и поведенческий аспект. Один и тот же класс может принимать участие в нескольких кооперациях; таким образом, они являются реализацией образцов поведения, формирующих систему. Графически кооперация изображается в виде эллипса, ограниченного пунктирной линией, в который обычно заключено только имя, как показано на рисунке.

Start Race

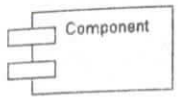
Кооперации

Прецедент (Use case) - это описание последовательности выполняемых системой действий, которая производит наблюдаемый результат,значимый для какого-то определенного *актера* (Actor). Прецедент применяется для структурирования поведенческих сущностей модели. Прецеденты реализуются посредством кооперации. Графически прецедент изображается в виде ограниченного непрерывной линией эллипса, обычно содержащего только его имя, как показано на рисунке.

Start Race

Прецеденты

Компонент (Component) - это физическая заменяемая часть системы, которая соответствует некоторому набору интерфейсов и обеспечивает его реализацию. В системе можно встретить различные виды устанавливаемых компонентов, такие как COM+ или Java Beans, а также компоненты, являющиеся артефактами процесса разработки, например файлы исходного кода. Компонент, как правило, представляет собой физическую упаковку логических элементов, таких как классы, интерфейсы и кооперации. Графически компонент изображается в виде прямоугольника с вкладками, содержащего обычно только имя, как показано на рисунке. Компонент подобен классу: он описывает совокупность объектов с общими атрибутами, операциями, отношениями и семантикой.

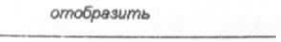


Компоненты

Эти базовые элементы - классы, интерфейсы, кооперации, прецеденты и компоненты - являются основными структурными сущностями, которые могут быть включены в модель UML. Существуют также разновидности этих сущностей: актеры, сигналы, утилиты (виды классов), процессы и нити (виды активных классов), приложения, документы, файлы, библиотеки, страницы и таблицы (виды компонентов).

Поведенческие сущности (Behavioral things) являются динамическими составляющими модели UML. Это глаголы языка: они описывают поведение модели во времени и пространстве. Существует всего два основных типа поведенческих сущностей.

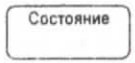
Взаимодействие (Interaction) - это поведение, суть которого заключается в обмене сообщениями (Messages) между объектами в рамках конкретного контекста для достижения определенной цели. С помощью взаимодействия можно описать как отдельную операцию, так и поведение совокупности объектов. Взаимодействие предполагает ряд других элементов, таких как сообщения, последовательности действий (поведение, инициированное сообщением) и связи (между объектами). Графически сообщения изображаются в виде стрелки, над которой почти всегда пишется имя соответствующей операции, как показано на рисунке.



Сообщения

Автомат (State machine) - это алгоритм поведения, определяющий последовательность состояний, через которые объект или взаимодействие проходят на протяжении своего жизненного цикла в ответ на различные события, а также реакции на эти события. С помощью автомата можно описать поведение отдельного класса или кооперации классов. С автоматом связан ряд других элементов: состояния, переходы (из одного состояния в другое), события (сущности, инициирующие переходы) и виды действий (реакция на переход). Графически состояние изображается в виде

прямоугольника с закругленными углами, содержащего имя и, возможно, подсостояния.



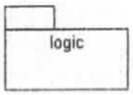
Состояния

Эти два элемента - взаимодействия и автоматы - являются основными поведенческими сущностями, входящими в модель UML. Семантически они часто бывают связаны с различными структурными элементами, в первую очередь - классами, кооперациями и объектами.

Группирующие сущности являются организующими частями модели UML. Это блоки, на которые можно разложить модель. Есть только одна первичная группирующая сущность, а именно пакет.

Пакеты (Packages) представляют собой универсальный механизм организации элементов в группы. В пакет можно поместить структурные, поведенческие и даже другие группирующие сущности. В отличие от компонентов, существующих во время работы программы, пакеты носят чисто концептуальный характер, то есть существуют только во время разработки.

Изображается пакет в виде папки с закладкой, содержащей, как правило, только имя и иногда - содержимое.



Пакеты

Пакеты - это основные группирующие сущности, с помощью которых можно организовать модель UML. Существуют также вариации пакетов, например каркасы (Frameworks), модели и подсистемы.

Аннотационные сущности - пояснительные части модели UML. Это комментарии для дополнительного описания, разъяснения или замечания к любому элементу модели. Имеется только один базовый тип аннотационных элементов - примечание (Note). Примечание - это просто символ для изображения комментариев или ограничений, присоединенных к элементу или группе элементов. Графически примечание изображается в виде прямоугольника с загнутым краем, содержащим текстовый или графический комментарий, как показано на рисунке.

Примечания

Этот элемент является основной аннотационной сущностью, которую можно включать в модель UML. Чаще всего примечания используются, чтобы снабдить диаграммы комментариями или ограничениями, которые можно выразить в виде неформального или формального текста. Существуют вариации этого элемента, например требования, где описывают некое желательное поведение с точки зрения внешней по отношению к модели.

В языке UML определены четыре типа отношений:

- зависимость;
- ассоциация;
- обобщение;
- реализация.

Эти отношения являются основными связующими строительными блоками в UML и применяются для создания корректных моделей.

Зависимость (Dependency) - это семантическое отношение между двумя сущностями, при котором изменение одной из них, независимой, может повлиять на семантику другой, зависимой. Графически зависимость изображается в виде прямой пунктирной линии, часто со стрелкой, которая может содержать метку.

Зависимости

Ассоциация (Association) - структурное отношение, описывающее совокупность связей; связь - это соединение между объектами. Разновидностью ассоциации является агрегирование (Aggregation) - так называют структурное отношение между целым и его частями. Графически ассоциация изображается в виде прямой линии (иногда завершающейся стрелкой или содержащей метку), рядом с которой могут присутствовать дополнительные обозначения, например кратность и имена ролей. На рисунке показан пример отношений этого типа.

Ассоциации

Обобщение (Generalization) - это отношение "специализация/обобщение", при котором объект специализированного элемента (потомок) может быть подставлен вместо объекта обобщенного элемента (родителя или предка). Таким образом, потомок (Child) наследует структуру и поведение своего родителя (Parent). Графически отношение обобщения изображается в виде линии с незакрашенной стрелкой, указывающей на родителя, как показано на рисунке.

Обобщения

Наконец, реализация (Realization) - это семантическое отношение между классификаторами, при котором один классификатор определяет "контракт", а другой гарантирует его выполнение. Отношения реализации встречаются в двух случаях: во-первых, между интерфейсами и реализующими их классами или компонентами, а во-вторых, между прецедентами и реализующими их кооперациями. Отношение реализации изображается в виде пунктирной линии с незакрашенной стрелкой, как нечто среднее между отношениями обобщения и зависимости.

Реализации

Четыре описанных элемента являются основными типами отношений, которые можно включать в модели UML. Существуют также их вариации, например уточнение (Refinement), трассировка (Trace), включение и расширение (для зависимостей).

Диаграмма в UML - это графическое представление набора элементов, изображаемое чаще всего в виде связанного графа с вершинами (сущностями) и ребрами (отношениями). Диаграммы рисуют для визуализации системы с разных точек зрения. Диаграмма - в некотором смысле одна из проекций системы. Как правило, за исключением наиболее тривиальных случаев, диаграммы дают свернутое представление элементов, из которых составлена система. Один и тот же элемент может присутствовать во всех диаграммах, или только в нескольких (самый распространенный вариант), или не присутствовать ни в одной (очень редко). Теоретически диаграммы могут содержать любые комбинации сущностей и отношений. На практике, однако, применяется сравнительно небольшое количество типовых комбинаций, соответствующих пяти наиболее

употребительным видам, которые составляют архитектуру программной системы.

Для *представления архитектуры*, а, точнее, различных входящих в нее структур, удобно использовать графические языки. На настоящий момент наиболее проработанным и наиболее широко используемым из них является унифицированный язык моделирования (Unified Modeling Language, UML), хотя достаточно часто архитектуру системы описывают просто набором именованных прямоугольников, соединенных линиями и стрелками, которые представляют возможные связи.

UML предлагает использовать для описания архитектуры 8 видов диаграмм. 9-й вид *UML* диаграмм, *диаграммы вариантов использования* (см. лекцию 4), не относится к *архитектурным представлениям*. Кроме того, и другие виды диаграмм можно использовать для описания внутренней структуры компонентов или сценариев действий пользователей и прочих элементов, к архитектуре часто не относящихся. В этом курсе мы не будем разбирать диаграммы *UML* в деталях, а ограничимся обзором их основных элементов, необходимым для общего понимания смысла того, что изображено на таких диаграммах.

Диаграммы *UML* делятся на две группы — статические и динамические диаграммы.

Статические диаграммы

Статические диаграммы представляют либо постоянно присутствующие в системе сущности и связи между ними, либо суммарную информацию о сущностях и связях, либо сущности и связи, существующие в какой-то определенный момент времени. Они не показывают способов поведения этих сущностей. К этому типу относятся диаграммы классов, объектов, компонентов и диаграммы развертывания.

- Диаграммы классов (class diagrams) показывают классы или типы сущностей системы, характеристики классов (поля и операции) и возможные связи между ними. Пример диаграммы классов изображен на рис.5.

Классы представляются прямоугольниками, поделенными на три части. В верхней части показывают имя класса, в средней — набор его полей, с именами, типами, модификаторами доступа (public '+', protected '#', private '-') и начальными значениями, в

нижней — набор операций класса. Для каждой операции показывается ее модификатор доступа и сигнатура.

На рис. 5.1 изображены классы Account, Person, Organization, Address, CreditAccount и абстрактный класс Client.

Класс CreditAccount имеет private поле maximumCredit типа dou ble, а также public метод getCredit() и protected метод setCredit().

Интерфейсы, т.е. типы, имеющие только набор операций и не определяющие способов их реализации, часто показываются в виде небольших кружков, хотя могут изображаться и как обычные классы. На рис. 4.1 представлен интерфейс AccountInterface.

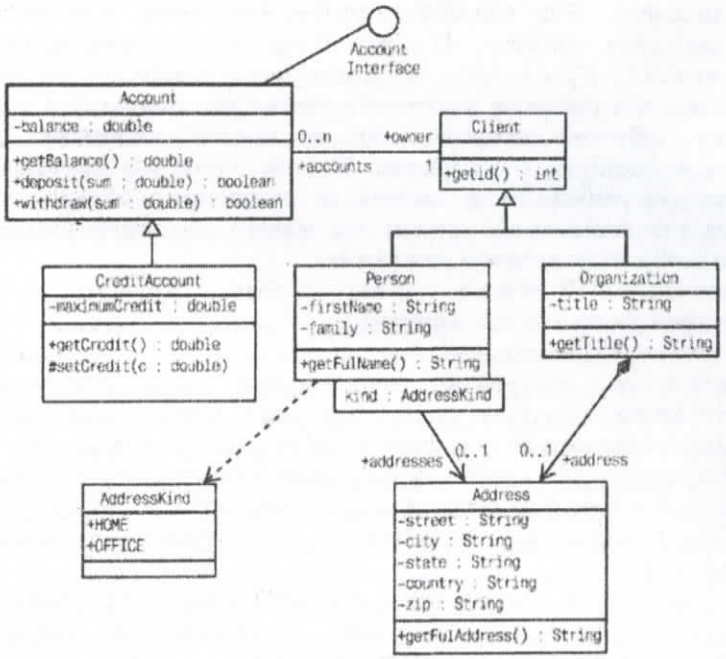


Рис. 4.1. Диаграмма классов

Наиболее часто используется три вида связей между классами — связи по композиции, ссылки, связи по наследованию и реализации.

Композиция описывает ситуацию, в которой объекты класса A включают в себя объекты класса B, причем последние не могут разделяться (объект класса B, являющийся частью объекта класса A, не может являться частью другого объекта класса A) и существуют только в рамках объемлющих объектов (уничтожаются при уничтожении объемлющего объекта).

Композицией на рис. 4.1 является связь между классами Organization и Address.

Ссылочная связь (или **слабая агрегация**) обозначает, что объект некоторого класса A имеет в качестве поля ссылку на объект другого (или того же самого) класса B, причем ссылки на один и тот же объект класса B могут иметься в нескольких объектах класса A.

И композиция, и ссылочная связь изображаются стрелками, ведущими от класса A к классу B. Композиция дополнительно имеет закрашенный ромбик у начала этой стрелки. Двусторонние ссылочные связи, обозначающие, что объекты могут иметь ссылки друг на друга, показываются линиями без стрелок. Такая связь показана на рис. 4.1 между классами Account и Client.

Эти связи могут иметь описание **множественности**, показывающее, сколько объектов класса B может быть связано с одним объектом класса A. Оно изображается в виде текстовой метки около конца стрелки, содержащей точное число или нижние и верхние границы, причем бесконечность изображается звездочкой или буквой n. Для двусторонних связей множественности могут показываться с обеих сторон. На рис. 4.1 множественности, изображенные для связи между классами Account и Client, обозначают, что один клиент может иметь много счетов, а может и не иметь ни одного, и счет всегда привязан ровно к одному клиенту.

Наследование классов изображается стрелкой с пустым наконечником, ведущей от наследника к предку. На рис. 5.1 класс CreditAccount наследует классу Account, а классы Person и Organization — классу Client.

Реализация интерфейсов показывается в виде пунктирной стрелки с пустым наконечником, ведущей от класса к реализуемому им интерфейсу, если тот показан в виде прямоугольника. Если же интерфейс изображен в виде кружка, то связь по реализации показывается обычной сплошной линией (в этом случае неоднозначности в ее толковании не возникает). Такая связь

изображена на рис. 4.1 между классом Account и интерфейсом AccountInterface.

Один класс **использует** другой, если этот другой класс является типом параметра или результата операции первого класса. Иногда связи по использованию показываются в виде пунктирных стрелок. Пример такой связи между классом Person и перечислимым типом AddressKind можно видеть на рис. 4.1.

Ссылочные связи, реализованные в виде ассоциативных массивов или отображений (map) — такая связь в зависимости от некоторого набора ключей определяет набор ссылок-значений — показываются при помощи стрелок, имеющих прямоугольник с перечислением типов и имен ключей, примыкающий к изображению класса, от которого идет стрелка. Множественность на конце стрелки при этом обозначает количество ссылок, соответствующее одному набору значений ключей.

На рисунке 4.1 такая связь ведет от класса Person к классу Address, показывая, что объект класса Person может иметь один адрес для каждого значения ключа kind, т.е. один домашний и один рабочий адреса.

Диаграммы классов используются чаще других видов диаграмм.

- **Диаграммы объектов (object diagrams)** показывают часть объектов системы и связи между ними в некотором конкретном состоянии или суммарно, за некоторый интервал времени.

Объекты изображаются прямоугольниками с идентификаторами ролей объектов (в контексте тех состояний, которые изображены на диаграмме) и типами. Однородные коллекции объектов могут изображаться накладывающимися друг на друга прямоугольниками.

Такие диаграммы используются довольно редко.

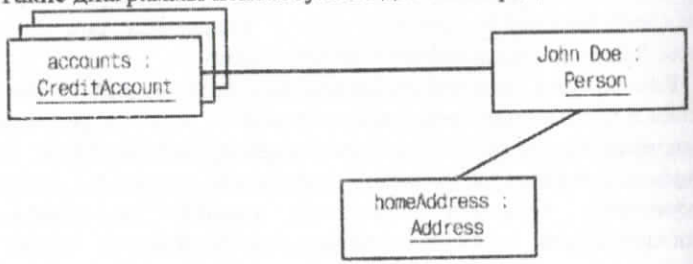


Рис. 4.2. Диаграмма объектов

- Диаграммы компонентов (component diagrams) представляют компоненты в нескольких смыслах — атомарные составляющие системы с точки зрения ее сборки, конфигурационного управления и развертывания. Компоненты сборки и конфигурационного управления обычно представляют собой файлы с исходным кодом, динамически подгружаемые библиотеки, HTML-странички и пр., компоненты развертывания — это компоненты JavaBeans, CORBA, COM и т.д.

Компонент изображается в виде прямоугольника с несколькими прямоугольными или другой формы "зубами" на левой стороне.

Связи, показывающие зависимости между компонентами, изображаются пунктирными стрелками. Один компонент зависит от другого, если он не может быть использован в отсутствии этого другого компонента в конфигурации системы. Компоненты могут также реализовывать интерфейсы.

Диаграммы этого вида используются редко.

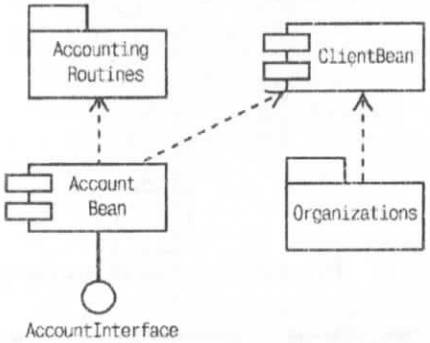


Рис. 4.3. Диаграмма компонентов

На диаграмме компонентов, изображенной на рис. 4.3, можно также увидеть пакеты, изображаемые в виде "папок", точнее — прямоугольников с прямоугольными "наростами" над левым верхним углом. Пакеты являются пространствами имен и средством группировки диаграмм и других модельных элементов UML — классов, компонентов и пр. Они могут появляться на диаграммах классов и компонентов для указания зависимостей между ними и отдельными классами и компонентами. Иногда на такой диаграмме могут присутствовать только пакеты с зависимостями между ними.

• Диаграммы развертывания (deployment diagrams) показывают декомпозицию системы на физические устройства различных видов — серверы, рабочие станции, терминалы, принтеры, маршрутизаторы и пр. — и связи между ними, представленные различного рода сетевыми и индивидуальными соединениями.

Физические устройства, называемые узлами системы (nodes), изображаются в виде кубов или параллелепипедов, а физические соединения между ними — в виде линий.

На диаграммах развертывания может быть показана привязка (в некоторый момент времени или постоянная) компонентов развертывания системы к физическим устройствам — например, для указания того, что компонент *EJB* AccountEJB исполняется на сервере приложений, а аплет AccountInfoEditor — на рабочей станции оператора банка.

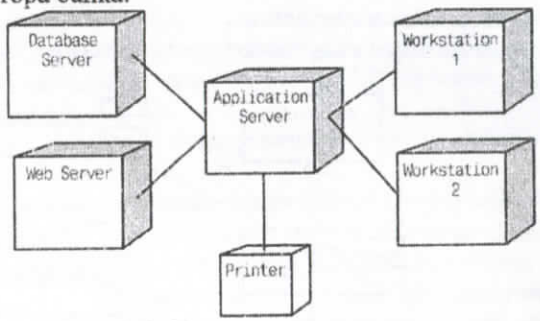


Рис. 4.4. Диаграмма развертывания

Эти диаграммы используются достаточно редко. Пример диаграммы развертывания изображен на рис. 4.4.

Динамические диаграммы

Динамические диаграммы описывают происходящие в системе процессы. К ним относятся диаграммы деятельности, сценариев, диаграммы взаимодействия и диаграммы состояний.

• Диаграммы деятельности (activity diagrams) иллюстрируют набор процессов-деятельностей и потоки данных между ними, а также возможные их синхронизации друг с другом.

Деятельность изображается в виде прямоугольника с закругленными сторонами, слева и справа, помеченного именем деятельности.

Потоки данных показываются в виде стрелок. Синхронизации двух видов — развилки (forks) и слияния (joins) — показываются жирными короткими линиями (кто-то может посчитать их и тонкими закрашенными прямоугольниками), к которым сходятся или от которых расходятся потоки данных. Кроме синхронизаций, на диаграммах деятельности могут быть показаны разветвления потоков данных, связанных с выбором того или иного направления в зависимости от некоторого условия. Такие разветвления показываются в виде небольших ромбов.

Диаграмма может быть поделена на несколько горизонтальных или вертикальных областей, называемых дорожками (swimlanes). Дорожки служат для группировки деятельностей в соответствии с выполняющими их подразделением организации, ролью, приложением, подсистемой и пр.

Диаграммы деятельности могут заменять часто используемые диаграммы потоков данных , поэтому применяются достаточно широко. Пример такой диаграммы показан на рис. 4.5.

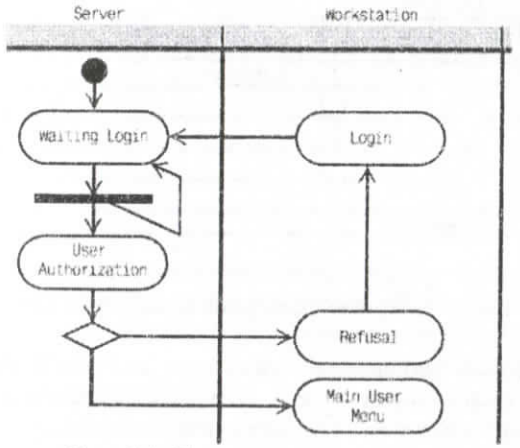


Рис. 4.5. Диаграмма активности

• Диаграммы сценариев (или диаграммы последовательности, sequence diagrams) показывают возможные сценарии обмена сообщениями или вызовами во времени между различными компонентами системы (здесь имеются в виду архитектурные компоненты, компоненты в широком смысле — это

могут быть компоненты развертывания, обычные объекты, подсистемы и пр.). Эти диаграммы являются подмножеством специального графического языка — языка диаграмм последовательностей сообщений (Message Sequence Charts, MSC), который был придуман раньше UML и достаточно долго развивается параллельно ему.

Компоненты, участвующие во взаимодействии, изображаются прямоугольниками вверху диаграммы. От каждого компонента вниз идет вертикальная линия, называемая его линией жизни. Считается, что ось времени направлена вертикально вниз. Интервалы времени, в которые компонент активен, т.е. управление находится в одной из его операций, представлены тонким прямоугольником, для которого линия жизни компонента является осью симметрии.

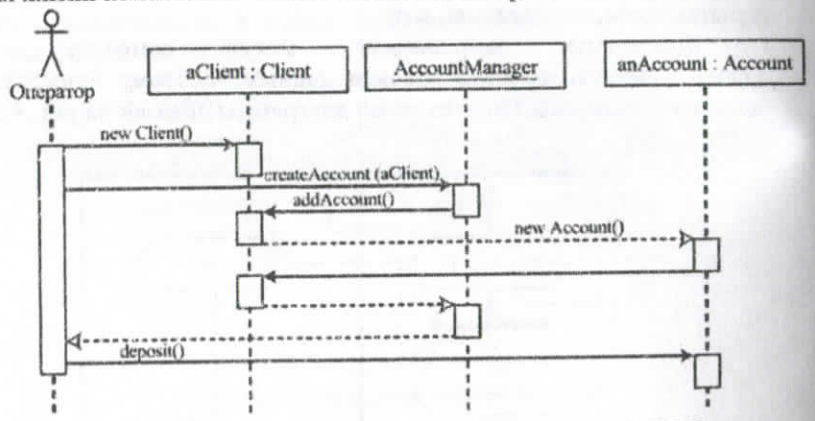


Рис. 4.6. Пример диаграммы сценария открытия счета

Передача сообщения или вызов изображаются стрелкой от компонента-источника к компоненту-приемнику. Возврат управления показан пунктирной стрелкой, обратной к соответствующему вызову.

Эти диаграммы используются достаточно часто, например, при детализации сценариев, входящих в варианты использования. Пример такой диаграммы изображен на рис. 4.6.

- Диаграммы взаимодействия (collaboration diagrams) показывают ту же информацию, что и диаграммы сценариев, но привязывают обмен сообщениями/вызовами не к

времени, а к связям между компонентами. Пример такой диаграммы представлен на рис. 4.7.

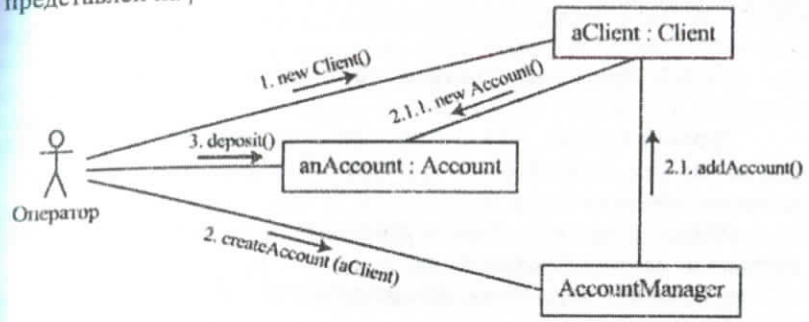


Рис. 4.7. Диаграмма взаимодействия, соответствующая диаграмме сценария на рис. 4.6

На диаграмме изображаются компоненты в виде прямоугольников и связи между ними. Вдоль связей могут передаваться сообщения, показываемые в виде небольших стрелок, параллельных связи. Стрелки нумеруются в соответствии с порядком происходящих событий. Нумерация может быть иерархической, чтобы показать вложенность действий друг в друга (т.е. если вызов некоторой операции имеет номер 1, то вызовы, осуществляемые при выполнении этой операции, будут нумероваться как 1.1, 1.2, и т.д.).

Диаграммы взаимодействия используются довольно редко.

- Диаграммы состояний (statechart diagrams) показывают возможные состояния отдельных компонентов или системы в целом, переходы между ними в ответ на какие-либо события и выполняемые при этом действия.

Состояния показываются в виде прямоугольников с закругленными углами, переходы — в виде стрелок. Начальное состояние представляется как небольшой темный кружок, конечное — как пустой кружок с концентрически вложенным темным кружком.

Состояния могут быть устроены иерархически: они могут включать в себя другие состояния, даже целые отдельные диаграммы вложенных состояний и переходов между ними.

Пребывая в таком состоянии, система находится ровно в одном из его подсостояний.

4.3. Проектирование программного обеспечения

Проектирование ПО – процесс определения архитектуры, компонентов, интерфейсов, других характеристик системы и конечного результата.

Область знаний «Проектирование ПО (Software Design)» состоит из следующих разделов:

– базовые концепции проектирования ПО (Software Design Basic Concepts),

– ключевые вопросы проектирования ПО (Key Issue in Software Design),

– структура и архитектура ПО (Software Structure and Architecture),

– анализ и оценка качества проектирования ПО (Software Design Quality Analysis and Evaluation),

– нотации проектирования ПО (Software Design Notations),

– стратегия и методы проектирования ПО (Software Design Strategies and Methods).

К базовым концепциям проектирования ПО относятся процессы ЖЦ, процесс проектирования архитектуры с использованием разных принципов (объектного, компонентного и др.) и техник: абстракции, декомпозиции, инкапсуляции и др. Автоматизируемая система декомпозируется на отдельные компоненты, выбираются необходимые артефакты (нотации, методы и др.) программной инженерии и строится архитектура ПО.

К ключевым вопросам проектирования ПОотносятся: декомпозиция на функциональные компоненты для независимого и параллельного их выполнения, принципы распределения компонентов в среде выполнения и их взаимодействие между собой, механизмы обеспечения качества и живучести системы и др.

При проектировании структуры ПО используется архитектурный стиль проектирования, основанный на определении основных элементов структуры – подсистем, компонентов и связей между ними.

Архитектура проекта – высокоуровневое представление структуры, задаваемое с помощью паттернов, компонентов и их идентификация. Описание архитектуры содержит описание логики отдельных компонентов системы, достаточное для проведения работ по кодированию, и связей между ними. Существуют и другие виды структур, основанные на проектировании образцов, шаблонов, семействе программ и их каркасов.

Паттерн – это конструктивный элемент ПО, который задает взаимодействие объектов (компонентов) проектируемой системы, определение ролей и ответственности исполнителей. Основным языком задания этого элемента является UML.

Паттерн может быть: структурным, в котором определяются типовые композиции структур из объектов и классов диаграммами классов, объектов, связей и др.; поведенческим, определяющим схемы взаимодействия классов объектов и их поведение диаграммами активностей, взаимодействия, потоков управления и др.; креативным, отображающим типовые схемы распределения ролей экземпляров объектов диаграммами взаимодействия, кооперации и др.

Анализ и оценка качества проектирования ПО включает мероприятия по анализу сформулированных в требованиях атрибутов качества, оценки различных аспектов ПО – размера и структуры ПО, функций и качества проектирования с помощью формальных метрик (функционально-ориентированных, структурных и объектно-ориентированных), а также проведения качественного анализа результатов проектирования путем статического анализа, моделирования и прототипирования.

Нотации проектирования позволяют представить артефакты ПО и его структуру, а также поведение системы. Существует два типа нотаций: структурные, поведенческие и множество различных их представлений.

Структурные нотации являются графическими, они используются для представления структурных аспектов проектирования, компонентов и их взаимосвязей, элементов архитектуры и их интерфейсов. К ним относятся формальные языки спецификаций и проектирования: ADL (Architecture Description Language), UML (Unified Modeling Language), ERD (Entity–Relation Diagrams), IDL (Interface Description Language), классы и объекты, компоненты и классы (CRC Cards), Use Case Driven и др. Нотации

включают языки описания архитектуры и интерфейса, диаграммы классов и объектов, диаграммы сущность-связь, компонентов, развертывания, а также структурные диаграммы и схемы.

Поведенческие нотации отражают динамический аспект поведения систем и их компонентов. Таким нотациям соответствуют диаграммы: Data Flow, Decision Tables, Activity, Colloboration, Pre-Post Conditions, Sequence, таблицы принятия решений, формальные языки спецификации, языки проектирования PDL и др.

Стратегия и методы проектирования ПО представляет различные стратегии и методы, которые используются при проектировании. К общим стратегиям относятся: снизу-вверх, сверху–вниз, абстракции, паттерны и др. Функционально–ориентированные (структурные) методы базируются на структурном анализе, структурных картах, Dataflow-диаграммах и др. Они ориентированы на идентификацию функций и их уточнение сверху–вниз, после чего проводится разработка диаграмм потоков данных и описание процессов. В обьектно–ориентированном проектировании ключевую роль играет наследование, полиморфизм и инкапсуляция, а также абстрактные структуры данных и отображение объектов. Подходы, ориентированные на структуры данных, базируются на методе Джексона (Jackson) и используются для задания входных и выходных данных структурными диаграммами.

Компонентное проектирование ориентировано на использование и интеграцию компонентов (особенно компонентов повторного использования) и на их интерфейс, обеспечивающий взаимодействие компонентов; является базисом других видов программирования, в том числе сервисно-ориентированного, в котором группы компонентов обеспечивают функциональный сервис. К другим методам относятся: формальные, точные и трансформационные методы, а также UML для моделирования архитектурных решений с помощью диаграмм.

Таким образом,предложенные в данной области знаний подходы, стратегии и методы проектирования ПО, средства распределения и взаимодействия компонентов в разных средах являются основными при разработке проекта с применением разных элементов (шаблонов, сценариев, диаграмм и др.) и стилей проектирования структуры ПО, а также мероприятий по проведению анализа полученных на этапе проектирования атрибутов качества ПО.

5 ГЛАВА. ПРОЦЕСС СОЗДАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И ЕГО ДОКУМЕНТИРОВАНИЕ

5.1. Процесс создания программного обеспечения

Процесс создания программного обеспечения – это множество взаимосвязанных процессов и результатов их выполнения, которые ведут к созданию программного продукта. Процесс создания ПО может начинаться с разработки программной системы "с нуля", но чаще новое ПО разрабатывается на основе существующих программных систем путем их модификации.

Процесс создания программного обеспечения, как и любая другая интеллектуальная деятельность, основан на человеческих суждениях и умозаключениях, т.е. является творческим. Вследствие этого все попытки автоматизировать процесс создания ПО имеют лишь ограниченный успех.

CASE-средства могут помочь в реализации некоторых этапов процесса разработки ПО, но по крайней мере в ближайшие несколько лет не стоит ожидать от них существенного продвижения в автоматизации тех этапов создания ПО, где существенен фактор творческого подхода к разработке ПО.

Одна из причин ограниченного применения автоматизированных средств к процессу создания ПО – огромное многообразие видов деятельности, связанных с разработкой программных продуктов. Кроме того, организации-разработчики используют разные подходы к разработке ПО.

Также различаются характеристики и возможности создаваемых систем, что требует особого внимания к определенным сторонам процесса разработки. Поэтому даже в одной организации при создании разных программных систем могут использоваться различные подходы и технологии.

Несмотря на то что наблюдается огромное многообразие подходов, методов и технологий создания ПО, существуют фундаментальные базовые процессы, без реализации которых не может обойтись ни одна технология разработки программных продуктов. Перечислим эти процессы.

1. *Разработка спецификации ПО.* Это фундамент любой программной системы. Спецификация определяет все функции и действия, которые будет выполнять разрабатываемая система.

2. *Проектирование и реализация (производство) ПО.* Это процесс непосредственного создания ПО на основе спецификации.

3. *Аттестация ПО.* Разработанное программное обеспечение должно быть аттестовано на соответствие требованиям заказчика.

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

Хотя не существует "идеального" процесса создания ПО, во многих организациях-разработчиках пытаются его усовершенствовать, поскольку он может опираться на устаревшие технологии и не включать лучших методов современной инженерии программного обеспечения. Кроме того, многие организации постоянно используют одни и те же технологии и им также необходимы методы современной инженерии ПО.

Совершенствовать процесс создания программных систем можно разными путями. Например, путем стандартизации, которая уменьшит разнородность используемых в данной организации технологий. Это, в свою очередь, приведет к совершенствованию внутренних коммуникаций в организации, уменьшению времени обучения персонала и сделает экономически выгодным процесс автоматизации разработок. Стандартизация обычно является первым шагом к внедрению новых методов и технологий инженерии ПО.

5.2. Создание программного обеспечения

Создание программного обеспечения – это множество процессов, приводящих к созданию программного продукта. Эти процессы основываются главным образом на технологиях инженерии программного обеспечения. Существует четыре фундаментальных процесса, которые присущи любому проекту создания ПО.

1. *Разработка спецификации требований на программное обеспечение.* Требования определяют функциональные характеристики системы и обязательны для выполнения.

2. *Создание программного обеспечения.* Разработка и создание ПО согласно спецификации на него.

3. *Аттестация программного обеспечения.* Созданное ПО должно пройти аттестацию для подтверждения соответствия требованиям заказчика.

4. *Совершенствование (модернизация) программного обеспечения.* ПО должно быть таким, чтобы его можно было модернизировать согласно измененным требованиям потребителя.

При выполнении разнообразных программных проектов эти процессы могут быть организованы различными способами и описаны на разных уровнях детализации. Длительность реализации этих процессов также далеко не всегда одинакова. И вообще, различные организации, занимающиеся производством ПО, зачастую используют разные процессы для создания программных продуктов даже одного типа.

Определенные процессы более подходят для создания программных продуктов одного типа и менее – для другого типа программных приложений. Если использовать неподходящий процесс, это может привести к снижению качества и функциональности разрабатываемого программного продукта.

5.3. Модель процесса создания ПО

Такая модель представляет собой упрощенное описание процесса создания ПО – последовательность практических этапов, необходимых для разработки создаваемого программного продукта. Подобные модели, несмотря на их разнообразие, служат абстрактным представлением реального процесса создания ПО. Модели могут отображать процессы, которые являются частью технологического процесса создания ПО, компоненты программных продуктов и действия людей, участвующих в создании ПО. Опишем типы моделей технологического процесса создания программного обеспечения.

1. *Модель последовательности работ.* Показывает последовательность этапов, выполняемых в процессе создания ПО, включая начало и завершение каждого этапа, а также зависимость между выполнением этапов. Этапы в этой модели соответствуют определенным работам, выполняемым разработчиками ПО.

2. *Модели потоков данных и процессов.* В них процесс создания ПО представляется в виде множества процессов, в

ходе реализации которых выполняются преобразования определенных данных. Например, на вход процесса создания спецификации ПО поступают определенные данные, на выходе этой активности получают данные, которые поступают на вход активности, соответствующей проектированию ПО, и т.д. Активность в такой модели часто является процессом более низкого порядка, чем этапы работ в модели предыдущего типа. Преобразования данных при реализации активностей могут выполнять как разработчики ПО, так и компьютеры.

3. *Ролевая модель.* Модель этого типа представляет роли людей, включенных в процесс создания ПО, и действия, выполняемые ими в этих ролях.

Существует также большое количество разнообразных моделей процесса разработки программного обеспечения.

1. *Каскадный подход.* Весь процесс создания ПО разбивается на отдельные этапы: формирование требований к ПО, проектирование и разработка программного продукта, его тестирование и т.д. Переход к следующему этапу осуществляется только после того, как полностью завершаются работы на предыдущем.

2. *Эволюционный подход.* Здесь последовательно перемежаются этапы формирования требований, разработки ПО и его аттестации. Первоначальная программная система быстро разрабатывается на основе некоторых абстрактных требований. Затем они уточняются и детализируются в соответствии с требованиями заказчика. Далее система дорабатывается и аттестуется в соответствии с новыми уточненными требованиями. Такая последовательность действий может повториться несколько раз.

3. *Формальные преобразования.* Основан на разработке формальной математической спецификации программной системы и преобразовании этой спецификации посредством специальных математических методов в программы. Такое преобразование удовлетворяет условию "сохранения корректности". Это означает, что полученная программа будет в точности соответствовать разработанной спецификации.

4. *Сборка программного продукта из ранее созданных компонентов.* Предполагается, что отдельные составные части программной системы уже существуют, т.е. созданы ранее. В

этом случае технологический процесс создания ПО основное внимание уделяет интеграции отдельных компонентов в общее целое, а не созданию этих компонентов.

5.4. Структура затрат на создание ПО

Структура затрат на создание программного обеспечения существенно зависит от процессов, используемых при разработке программного обеспечения, а также от типа разрабатываемого программного продукта.

Если принять общую стоимость создания программного обеспечения за 100 единиц, то распределение стоимостей отдельных этапов производства может иметь такой вид, как на рис. 5.1.

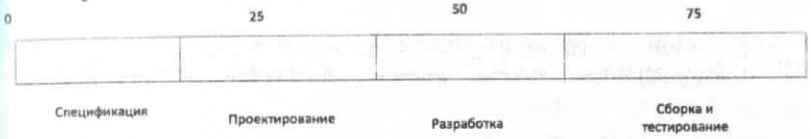


Рис. 5.1. Распределение стоимостей отдельных этапов производства ПО

Такая структура затрат возможна тогда, когда затраты на создание спецификации, проектирование ПО, его разработку и сборку подсчитываются отдельно. Отметим, что часто стоимость этапа сборки и тестирования превышает стоимость этапа непосредственно разработки ПО.

Например, на рис. 5.1 показана структура затрат, при которой на тестирование программной системы приходится примерно 40% общей стоимости затрат. Вместе с тем для некоторых критических систем эта статья расходов может превышать 50%.

При использовании эволюционного подхода к разработке ПО практически невозможно провести четкое разграничение между этапами создания спецификации, проектирования и разработки ПО.

Поэтому структуру затрат, представленную на рис. 5.1, следует изменить так, как показано на рис. 5.2. Здесь оставлен отдельный этап разработки спецификации, поскольку общая спецификация высшего уровня создается еще до начала создания программного продукта.

Создание спецификации нижнего уровня, проектирование, реализация, сборка и тестирование ПО при таком подходе

5.5. Методы инженерии программного обеспечения

Методы инженерии программного обеспечения представляют собой структурный подход к созданию ПО, который способствует производству высококачественного программного продукта эффективным, в экономическом аспекте, способом. Такие методы, как структурный анализ и JSD (метод Джексона разработки систем), впервые были представлены еще в 1970-х годах.

Эти методы, названные функционально-модульными или функционально-ориентированными, связаны с определением основных функциональных компонентов программной системы и в свое время широко использовались. В 80-90-х годах к этим методам добавились объектно-ориентированные методы, предложенные Бучем (Booch) и Рамбо (Rumbaugh).

Эти методы, использующие разные подходы, ныне интегрированы в единый унифицированный метод, построенный на основе унифицированного языка моделирования UML (Unified Modeling Language).

Все упомянутые методы основаны на идее создания моделей системы, которые можно представить графически, и на использовании этих моделей в качестве спецификации системы или ее структуры.

Не существует идеального и универсального метода – каждый метод имеет свою область применимости. Например, объектно-ориентированные методы часто применяются для создания интерактивных (диалоговых) программных систем, но практически не используются при разработке систем, работающих в режиме реального времени.

5.6. CASE-технология

Термин CASE обозначает Computer-Aided Software Engineering – автоматизированная разработка программного обеспечения. Под этим понимается широкий спектр программ, применяемых для поддержки и сопровождения различных этапов создания программного обеспечения: анализа системных требований, моделирования системы, ее отладки и тестирования и др.

Все современные методы создания программного обеспечения используют соответствующие CASE-средства: редакторы нотаций,

применяемых для описания моделей, модули анализа, проверяющие соответствие модели правилам метода, и генераторы отчетов, помогающие при создании документации на разрабатываемое программное обеспечение.

Кроме того, CASE-средства могут включать генератор кода, который автоматически генерирует исходный код программ на основе модели системы, а также руководство пользователя.

CASE-средства, предназначенные для анализа спецификаций и проектирования программного обеспечения, иногда называют CASE-средствами верхнего уровня, поскольку они применяются на начальной стадии разработки программных систем. В то же время CASE-средства, нацеленные на поддержку разработки и тестирования программного обеспечения, т.е. отладчики, системы анализа программ, генераторы тестов и редакторы программ, подчас называют CASE-средствами нижнего уровня.

5.7. Характеристики качественного программного обеспечения

Кроме функциональных возможностей, присущих программным продуктам по определению, эти продукты обладают и другими показателями, характеризующими их качество. Данные показатели не вытекают непосредственно из того, какие действия может выполнять программный продукт. Они характеризуют поведение программы во время выполнения ею своих действий, структуру и организацию исходного кода программы, ее документированность. Примером таких показателей может служить время ожидания пользователем ответа на свой запрос или понятность программного кода.

Конечно, множество тех показателей или характеристик, которые можно ожидать от программного обеспечения, зависит от типа программной системы.

Например, банковская система должна быть защищенной, интерактивная игра должна быть чувствительной к действиям пользователя-игрока, систему телефонных переключений прежде всего характеризует ее надежность и т.д.

Но эти специфические показатели, как и множество других подобных характеристик, можно обобщить в виде показателей качественных программных систем, приведенных в табл. 2.

Таблица 2. Основные показатели качественного программного обеспечения

Описание	Показатель
Удобство сопровождения	ПО должно быть таким, чтобы существовала возможность его усовершенствования в ответ на измененные требования заказчика или пользователя. Это определяющий показатель, поскольку любое ПО неминуемо подвергается модернизации вследствие изменений, происходящих в реальном мире
Надежность	Определяется рядом характеристик, таких как безотказность, защищенность и безопасность. Надежность ПО означает, что возможные сбои в работе системы не приведут к физическому или экономическому ущербу
Эффективность	Работа ПО не должна приводить к расточительному расходованию таких системных ресурсов, как память или время занятости процессора. Поэтому эффективность ПО описывается следующими характеристиками: скорость выполнения, используемое процессорное время, объем требуемой памяти и т.п.
Удобство в использовании	ПО должно быть удобным в эксплуатации и не требовать чрезмерного напряжения усилий пользователя того уровня, на которого оно рассчитано. Это означает, что программная система должна обладать соответствующим пользовательским интерфейсом и необходимой документацией

Большинство методов, средств и технологий инженерии программного обеспечения ориентированы на то, чтобы помочь в создании программных систем с этими показателями качественного программного обеспечения.

5.8. Основные проблемы, стоящие перед специалистами по программному обеспечению

На сегодняшний день специалисты по программному обеспечению столкнулись с описанными ниже проблемами.

1. *Проблема наследования ранее созданного ПО.* Многие большие программные системы, эксплуатируемые в настоящее время, созданы много лет назад, но до сих пор выполняют свои функции надлежащим образом. Проблема наследования означает поддержку и модернизацию таких систем, причем при минимальных финансовых и временных затратах.

2. *Проблема все возрастающей разнородности программных систем.* В настоящее время программное обеспечение должно быть способно работать в качестве систем, распределенных в компьютерных сетях, состоящих из компьютеров разных типов и использующих различные операционные системы. Проблема возрастающей разнородности программных систем состоит в том, что необходимо разрабатывать надежные программные системы, способные работать совместно с ПО разных типов.

3. *Проблема, порожденная требованием уменьшения времени на создание ПО.* Многие традиционные технологии создания качественного программного обеспечения требуют больших временных затрат. Вместе с тем сегодня запросы рынка ПО и требования к программным системам меняются очень быстро. Поэтому и ПО должно меняться с соответствующей скоростью. Проблема, порожденная требованием уменьшения времени на создание ПО, заключается в том, чтобы сократить время на разработку больших и сложных программных систем без снижения их качества.

Конечно, перечисленные проблемы связаны друг *с* другом. Например, возможна такая ситуация, когда необходимо быстро разработать на основе существующей системы ее сетевой вариант.

Для решения таких проблем необходимы новые средства и технологии, которые вобрали бы в себя все *лучшие* методы современной инженерии программного обеспечения.

Java –объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems. Приложения Java транслируются в специальный байт-код, поэтому они могут работать на любой виртуальной Java-машине. Java был выпушен — 23 мая 1995 года.

В начале его называли Oak. Его разрабатывал Джеймс Гослинг для программирования бытовых электронных устройств. Впоследствии он был переименован в Java и стал использоваться для написания клиентских приложений и серверного программного обеспечения.

Программы на Java транслируются в байт-код, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

Особенностью технологии Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы вызывают немедленное прерывание.

К недостаткам концепции виртуальной машины относят снижение производительности. Ряд усовершенствований несколько увеличил скорость выполнения программ на Java:

- применение технологии трансляции байт-кода в машинный код во время работы программы (JIT-технология) с возможностью сохранения версий класса в машинном коде;
- широкое использование платформенно-ориентированного кода (native-код) в стандартных библиотеках;
- аппаратные средства, обеспечивающие ускоренную обработку байт-кода (например, технология Jazelle).

Для задач время выполнения на Java составляет в среднем в полтора-два раза больше, чем для C/C++, в некоторых случаях Java быстрее, а в отдельных случаях в 7 раз медленнее.

С другой стороны, для большинства из них потребление памяти Java-машиной было в 10—30 раз больше, чем программой на C/C++.

6.2. Среды программирования

На сегодня есть очень много программных средств разработки программного обеспечения для мобильных приложений.

Android Studio — это интегрированная среда разработки (IDE) для работы с платформой Android. IDE находится в свободном доступе начиная с версии 0.1, опубликованной в мае 2013, а потом перешла в стадию бета-тестирования, начиная с версии 0.8, которая была выпущена в июне 2014 года.

Первая версия Android Studio 1.0 была выпущена в декабре 2014 года, после этого прекратилась поддержка плагина Android Development Tools (ADT) для Eclipse.

Android Studio основана на программном обеспечении IntelliJ IDEA от компании JetBrains. Он считается официальном средством разработки Android приложений. Данная программная среда разработки доступна для Windows, OSX и Linux.

17 мая 2017 года на конференции Google I/O была представлена новая программа – язык Kotlin, используемый в Android Studio, официальным языком программирования для платформы Android в добавление к Java и C++.

Новые функции Android Studio появляются в каждой новой версии. Сейчас доступны следующие функции:

- Расширенный редактор макетов: WYSIWYG, способность работать с UI компонентами при помощи Drag-and-Drop, функция предпросмотра макета на нескольких конфигурациях экрана;
- Сборка приложений, основанная на Gradle;
- Различные виды сборок и генерация нескольких .apk файлов;
- Рефакторинг кода;
- Статический анализатор кода (Lint), позволяющий находить проблемы производительности, несовместимости версий и другое;
- Встроенный ProGuard и утилита для подписывания приложений;
- Шаблоны основных макетов и компонентов Android;
- Поддержка разработки приложений для Android Wear и Android TV;

- Встроенная поддержка Google Cloud Platform, которая включает в себя интеграцию с сервисами Google Cloud Messaging и App Engine;
- Android Studio 2.1 поддерживает Android N Preview SDK;
- Новая версия Android Studio 2.1 способна работать с обновленным компилятором Jack;
- Platform-tools для Linux;

В Android Studio 3.0 будут по стандарту включены инструменты языка Kotlin основанные на JetBrains IDE.

IntelliJ IDEA — интегрированная среда разработки программного обеспечения для многих языков программирования, как Java, JavaScript, Python.

Первая версия IntelliJ IDEA появилась в январе 2001 года и приобрела популярность как первая среда для Java с широким набором интегрированных инструментов.

Дизайн среды IntelliJ IDEA ориентирован на продуктивность работы программистов, позволяя сконцентрироваться на функциональных задачах.

Начиная с шестой версии IntelliJ IDEA предоставляет интегрированный инструментарий для разработки графического пользовательского интерфейса. Среди других возможностей, среда хорошо совместима со многими инструментами разработчиков, такими как CVS, Subversion, Apache Ant, Maven и JUnit.

В феврале 2007 года разработчики IntelliJ IDEA представили версию плагина для языка Ruby.

В версии IntelliJ IDEA 9.0 среда доступна в двух видах: Community Edition и Ultimate Edition. Community Edition является свободной версией, в ней реализована полная поддержка Java SE, Groovy, Scala, а также интеграция с наиболее популярными системами управления версиями. В Ultimate Edition реализована поддержка Java EE, UML-диаграмм, подсчёт покрытия кода, а также поддержка других систем управления версиями, языков и фреймворков.

При создание нового проекта в IntelliJ IDEA программа предлагает выбор языков программирования. IntelliJ IDEA поддерживает следующие языки:

- Java
- JavaScript
- CoffeeScript
- HTML/XHTML/HAML
- CSS/SASS/LESS
- XML/XSL/XPath
- YAML
- ActionScript/MXML
- Python
- Ruby
- Haxe
- Groovy
- Scala
- SQL
- PHP
- Kotlin
- Clojure
- Си
- C++
- Go

Многие языкы поддерживаются посредством плагинов сторонних разработчиков, так реализована поддержка OCaml, GLSL, Erlang, Fantom, Haskell, Lua, Mathematica, Rust, Perl5.

Intel XDK – программное обеспечение обеспечивает работу над всем жизненным циклом разработки кроссбраузерных мобильных приложений с использованием веб-технологии (в частности HTML5 и Cordova).

Основным преимуществом Intel XDK является возможность разработки на CSS и JavaScript, потом компилировать проект в установочные файлы для IOS, Android и Windows Phone – .ipa, .apk и .appx соответственно. Потом данные файлы можно загружать в магазины приложений. Возможности Intel XDK:

- Позволяет легко разрабатывать кроссплатформенные приложения;
- Включает в себя инструменты для создания, отладки и сборки ПО, а также эмулятор устройств;
- Поддерживает разработку для Android, Apple iOS, Microsoft Windows 8;

- Языки разработки HTML5 и JavaScript.

Marmalade SDK – кроссплатформенное SDK. Ситоит из набора библиотек, образцов, инструментов и документаций, необходимых для разработки, тестирования и развертывания приложений для мобильных устройств.

Marmalade SDK ранее назывался Airplay SDK, потом был переименован в июне 2011 года в Marmalade SDK, после выхода версии 5.0. SDK начал работать как внутренняя библиотека, используемая для разработки видеоигр для мобильных устройств в Ideaworks3D.

В Marmalade SDK однократно написание программы и компилирование её на все поддерживаемые платформы, без необходимости программирования на различных языках программирования и использования различных API для каждой платформы.

В 2016 году Marmalade SDK был приобретен японской компанией GMO Cloud.

Для использования Marmalade SDK необходимо лицензия. Лицензия требуется для каждого компьютера, на котором установлен Marmalade SDK.

Marmalade SDK поддерживает развертывание приложений на различных платформах в зависимости от уровня приобретенной лицензии:

- Google Android (Все типы лицензий);
- BlackBerry OS PlayBook (Indie и выше);
- Apple iOS (Все типы лицензий);
- LG Smart TV (Professional и лицензированные LG разработчики);
- Apple Mac OS X (Plus и выше);
- Microsoft Windows (Plus и выше);
- Microsoft Windows Phone 8 (Indie и выше);
- Tizen (Indie и выше).

Основа Marmalade SDK состоит из двух основных слоев:

- Низкоуровневый C API называется Marmalade System. Он позволяе получить программисту доступ к функциям устройства, таким как управление памятью, доступ к файлам и сети, данным ввода (например: акселерометр, клавиатура, сенсорный экран), звуку.

- Marmalade Studio C++ API, который обеспечивает функциональность высокого уровня, в основном направлен на поддержку 2D (например, обработка растровых изображений и шрифтов) и 3D-рендеринга графики.

6.3. Основные типы данных

В языке Java есть 8 примитивных (скалярных, простых) типов: boolean, byte, char, short, int, long, float, double. Существует также примитивный тип — void, однако переменные и поля такого типа не могут быть объявлены в коде, а сам тип используется для описания соответствующего ему класса.

Кроме того, с помощью класса Void можно узнать, является ли определённый метод типа void:

Hello.class.getMethod("main", Array.newInstance(String.class, 0).getClass()).getReturnType() == Void.TYPE.

Длины и диапазоны значений примитивных типов определяются стандартом и приведены в таблице 3. Тип char сделали двухбайтовым для удобства локализации.

Был добавлен новый тип byte, в отличие от других языков, он не является беззнаковым. Типы float и double могут иметь специальные значения и «не число» (NaN).

Для типа double они обозначаются Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY, Double.NaN; для типа float — так же, но с приставкой Float вместо Double. Минимальные и максимальные значения, принимаемые типами float и double, тоже стандартизованы (таблица 3).

Таблица 3. Типы данных

Тип	Длина (в байтах)	Диапазон или набор значений
boolean	1 в массивах, 4 в переменных	true, false
byte	1	−128..127
char	2	$0..2^{16}-1$, или 0..65535
short	2	$-2^{15}..2^{15}-1$, или −32768..32767
int	4	$-2^{31}..2^{31}-1$, или −2147483648..2147483647
long	8	$-2^{63}..2^{63}-1$, или примерно

		$-9.2\cdot10^{18}..9.2\cdot10^{18}$
float	4	$-(2-2^{-23})\cdot2^{127}..(2-2^{-23})\cdot2^{127}$, или примерно $-3.4\cdot10^{38}..3.4\cdot10^{38}$, а также $-\infty$, ∞, NaN
double	8	$-(2-2^{-52})\cdot2^{1023}..(2-2^{-52})\cdot2^{1023}$, или примерно $-1.8\cdot10^{308}..1.8\cdot10^{308}$, а также $-\infty$, ∞, NaN

Такая стандартизация была необходима, чтобы сделать язык платформенно-независимым, что является одним из требований к Java. Некоторые процессоры используют для промежуточного хранения результатов 10-байтовые регистры или другими способами улучшают точность вычислений.

Для того, чтобы сделать Java максимально совместимой между разными системами, в ранних версиях любые способы повышения точности вычислений были запрещены.

Однако это приводило к снижению быстродействия. Ухудшение точности ради платформенной независимости не кому нужно, тем более если за это приходится платить замедлением работы программ.

Преобразования при математических операциях в языке Java действуют следующие правила:

1. Если один операнд имеет тип double, другой тоже преобразуется к типу double.
2. Иначе, если один операнд имеет тип float, другой тоже преобразуется к типу float.
3. Иначе, если один операнд имеет тип long, другой тоже преобразуется к типу long.
4. Иначе оба операнда преобразуются к типу int.

Данный способ неявного преобразования встроенных типов полностью совпадает с преобразованием типов в C++.

В языке Java имеются только динамически создаваемые объекты. Переменные объектного типа и объекты в Java – совершенно разные сущности. Переменные объектного типа являются ссылками, то есть неявными указателями на динамически создаваемые объекты. Это подчёркивается синтаксисом описания переменных. Так, в Java нельзя писать:

```
double a[10][20];
Foo b(30);
```

а нужно:

```
double[][] a = new double[10][20];
Foo b = new Foo(30);
```


При присваиваниях, передаче в подпрограммы и сравнениях объектные переменные ведут себя как указатели, то есть присваиваются, копируются и сравниваются адреса объектов. А при доступе с помощью объектной переменной к полям данных или методам объекта не требуется никаких специальных операций разыменовывания – этот доступ осуществляется так, как если бы объектная переменная была самим объектом.

Объектными являются переменные любого типа, кроме примитивного. Явных указателей в Java нет. В отличие от указателей C, C++ и других языков программирования, ссылки в Java в высокой степени безопасны благодаря жёстким ограничениям на их использование, в частности:

- Нельзя преобразовывать объект типа int или любого другого примитивного типа в указатель или ссылку и наоборот.
- Над ссылками запрещено выполнять операции ++, —, +, – или любые другие арифметические операции.
- Преобразование типов между ссылками жёстко регламентировано.

За исключением ссылок на массивы, разрешено преобразовывать ссылки только между наследуемым типом и его наследником, причём преобразование наследуемого типа в наследующий должно быть явно задано и во время выполнения производится проверка его осмысленности. Преобразования ссылок на массивы разрешены лишь тогда, когда разрешены преобразования их базовых типов, а также нет конфликтов размерности.

В Java нет операций взятия адреса (&) или взятия объекта по адресу (*). Амперсанд (&) означает «побитовое и» (двойной амперсанд — «логическое и»). для булевых типов одиночный амперсанд означает «логическое и», отличающееся от двойного тем, что цепь проверок не прекращается при получении в выражении значения false.

Благодаря таким специально введенным ограничениям в Java невозможно прямое манипулирование памятью на уровне физических адресов.

Если нужен указатель на примитивный тип, используются классы-обёртки примитивных типов: Boolean, Byte, Character, Short, Integer, Long, Float, Double.

7 ГЛАВА. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

7.1. Введение в объектно-ориентированное проектирование

Объектно-ориентированное проектирование представляет из себя операции и функции в понятиях *объект*. Программная система состоит из взаимодействующих объектов, которые имеют собственное локальное состояние и могут выполнять определенный набор операций, определяемый состоянием объекта (рис. 7.1).

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.

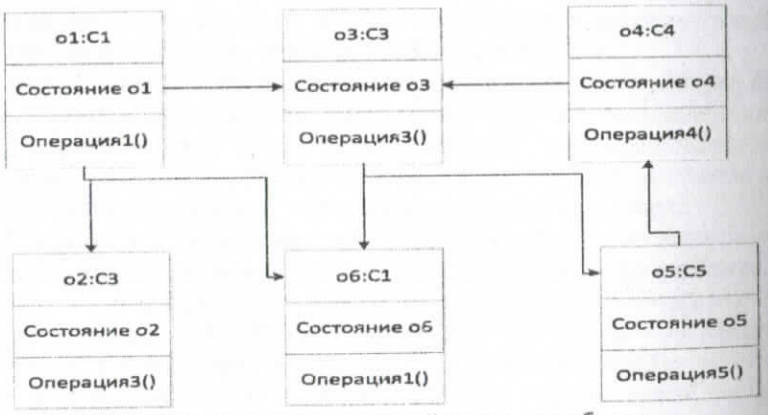


Рис. 7.1. Система взаимодействующих объектов

Объектно-ориентированное проектирование – 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 с учетом системных требований. В объектно-ориентированной модели определение всех объектов подчинено решению конкретной задачи.
- *Объектно-ориентированное программирование.* Реализация архитектуры 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 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. Иногда повторное использование эффективнее всего реализовать с помощью коллекций объектов (компонентов или объектных структур), а не через отдельные объекты.

7.2. Объекты и классы объектов

В настоящее время широко используются понятия *объект* и *объектно-ориентированный*. Эти термины применяются к различным типам объектов, методам проектирования, системам и языкам программирования. Во всех случаях применяется общее правило, согласно которому объект инкапсулирует данные о своем внутреннем строении. Это правило отражено в моем определении объекта и класса объектов.

Объект – это нечто, способное пребывать в различных состояниях и имеющее определенное множество операций. Состояние определяется как набор атрибутов объекта. Операции, связанные с объектом, предоставляют сервисы (функциональные возможности) другим объектам (клиентам) для выполнения определенных вычислений.

Объекты создаются в соответствии с определением класса объектов, которое служит шаблоном для создания объектов. В него включены объявления всех атрибутов и операций, связанных с объектом данного класса.

Нотация, которая используется здесь для обозначения классов объектов, определена в UML. Класс объектов представляется как прямоугольник с названием класса, разделенный на две секции. В верхней секции перечислены атрибуты объектов. Операции, связанные с данным объектом, расположены в нижней секции.

Пример такой нотации представлен на рис. 7.2, где показан класс объектов, моделирующий служащего некой организации. В UML термин *операция* является спецификацией некоторого действия, а термин *метод* обычно относится к реализации данной операции.

Класс **Работник** определяется рядом атрибутов, в которых содержатся данные о служащих, в том числе их имена и адрес, коды социального обеспечения, налоговые коды и т.д. Конечно, на самом деле атрибутов, ассоциированных с классом, больше, чем изображено на рисунке.

Определены также операции, связанные с объектами: **принять** (выполняется при поступлении на работу), **уволить** (выполняется при увольнении служащего из организации), **пенсия** (выполняется, если служащий становится пенсионером организации) и **изменитьДанные** (выполняется в случаях, если требуется внести изменения в имеющиеся данные о работнике).

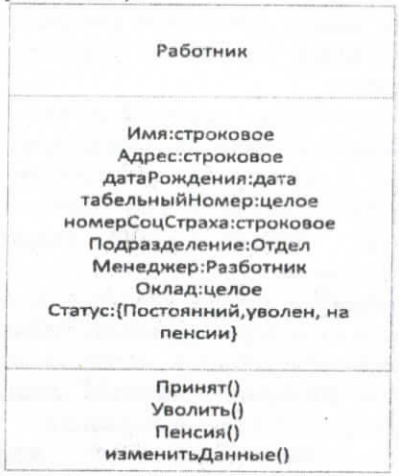


Рис. 7.2. Объект Работник

Взаимодействие между объектами осуществляется посредством запросов к сервисам (вызов методов) из других объектов и при необходимости путем обмена данными, требующимися для поддержки сервиса. Копии данных, необходимых для работы сервиса, и результаты работы сервиса передаются как параметры. Вот несколько примеров такого стиля взаимодействия.

```
// Вызов метода, ассоциированного с объектом Buffer (Буфер),
// который возвращает следующее значение в буфер v =
circularBuffer.Get();
// Вызов метода, связанного с объектом thermostat (термостат),
// который поддерживает нужную температуру
thermostat.setTemp (20);
```

В некоторых распределенных системах взаимодействие между объектами реализовано непосредственно в виде текстовых сообщений, которыми обмениваются объекты. Объект, получивший

сообщение, выполняет его грамматический разбор, идентифицирует сервис и связанные с ним данные и запускает запрашиваемый сервис. Однако, если объекты сосуществуют в одной программе, вызовы 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.

Классы объектов можно упорядочить или в виде иерархии обобщения или в виде иерархии наследования, которые показывают отношения между основными и частными классами объектов. 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. 7.3, 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.

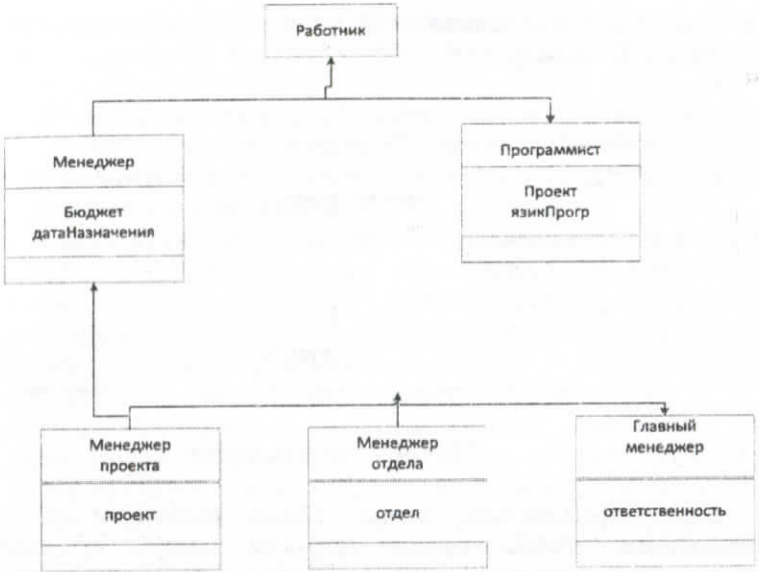


Рис. 7.3. Иерархия обобщения

На рис. 7.3 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. 7.4 показаны связи между

объектами классов **Работник** и **Отдел** и между объектами классов **Работник** и **Менеджер**.

Рис. 7.4. 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.

7.3. 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.

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 7.1 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 7.1. Реализация активного объекта, использующего потоки языка Java

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

7.4. Процесс объектно-ориентированного проектирования

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 я значительно упростил 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 я воспользуюсь 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. 7.5 изображена одна из возможных архитектур системы, которую можно построить на основе предложенного описания. Она представляет собой многоуровневую архитектуру, 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.

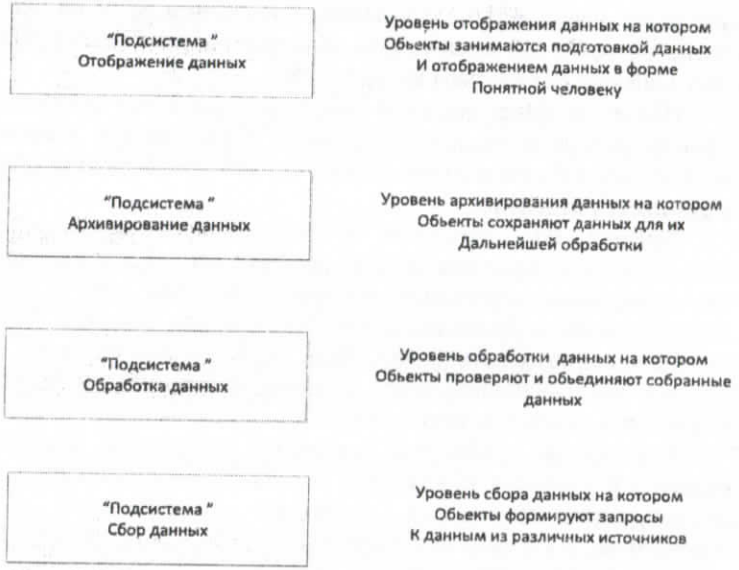


Рис. 7.5. Многоуровневая архитектура системы построения карт погоды

На рис. 7.5 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. Я использую здесь это обозначение, чтобы показать, что каждый уровень включает в себя множество других компонентов.

На рис. 7.6 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**.

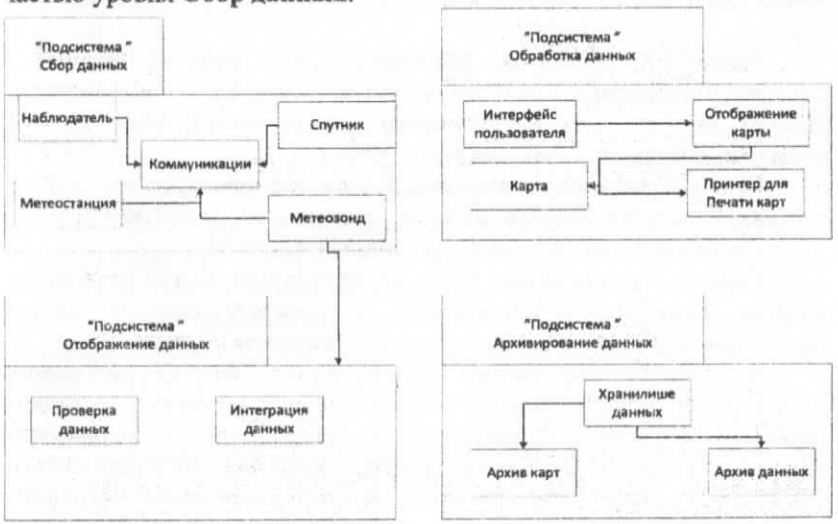


Рис. 7.6. Подсистемы в системе построения карт погоды

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 (рис. 7.4), которая дает простую блок-схему общей архитектуры системы. С помощью *пакетов* языка UML ее можно представить в развернутом виде как совокупность подсистем (рис. 7.6).

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. 7.7. 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.

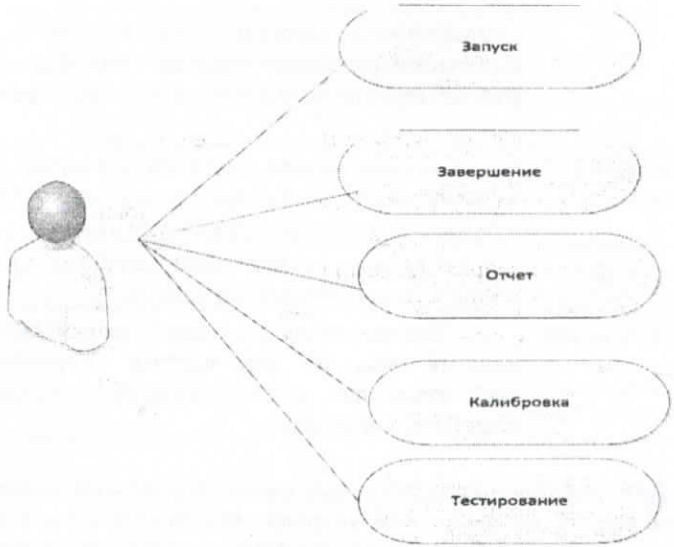


Рис. 7.7. Варианты использования метеостанции

Такое описание помогает разработчикам проекта идентифицировать объекты в системе и понять, что система должна делать. Я использую стилизованную форму описания, которая четко определяет, как происходит обмен информацией, как инициируется взаимодействие и т.д. Эта форма описания показана в табл. 3, где представлен вариант использования **Отчет** (рис. 7.7).

Таблица 3. 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

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.

Если 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. 7.8 внутри большого прямоугольника **Метеостанция** расположены три прямоугольника UML. Здесь я использовал систему нотации UML (текст в прямоугольниках с загнутыми углами) с тем, чтобы представить дополнительную информацию.

Рис. 7.8. 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. 7.6.

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.

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 источников. Объекты и операции, первоначально определенные на основе неформального описания системы, вполне могут послужить отправной точкой при проектировании.

Для усовершенствования и расширения описания первоначальных объектов можно использовать дополнительную информацию, полученную из области применения ПО или анализа сценариев. Дополнительную информацию также можно получить в ходе обсуждения с пользователями разрабатываемой системы или анализа имеющихся систем.

При определении объектов метеостанции мы используем смешанный подход. Чтобы описать все объекты, потребуется много места, поэтому на рис. 7.9 показали только пять классов объектов. **ПочвенныйТермометр**, **Анемометр** и **Барометр** являются объектами области приложения, а объекты **Метеостанция** и **МетеоДанные** определены на основе описания системы и описания сценариев (вариантов использования). Все объекты связаны с различными уровнями в архитектуре системы.

1. Класс объектов **Метеостанция** предоставляет основной интерфейс метеостанции при работе с внешним окружением. Поэтому операции класса соответствуют взаимодействиям, показанным на рис. 4.7. В данном случае, чтобы описать все эти взаимодействия, я использую один класс объектов, но в других проектах для представления интерфейса системы, возможно, потребуется использовать несколько классов.
2. Класс объектов **МетеоДанные** инкапсулирует итоговые данные от различных приборов метеостанции. Связанные с ним операции собирают и обобщают данные.
3. Классы объектов **ПочвенныйТермометр**, **Анемометр** и **Барометр** отображают реальные аппаратные средства метеостанции, соответствующие операции этих классов должны управлять данными приборами.

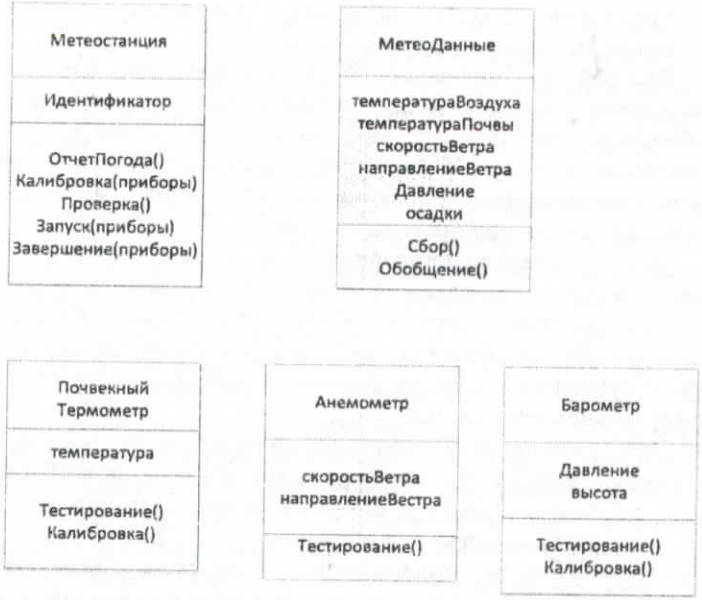


Рис. 7.9. Примеры классов объектов системы метеостанции

На этом этапе проектирования знания из области приложения ПО можно использовать для идентификации будущих объектов и сервисов.

В нашем примере известно, что метеостанции обычно расположены в удаленных местах. Они оснащены различными приборами, которые иногда дают сбой в работе. Отчет о неполадках в приборах должен отправляться автоматически. А это значит, что необходимы атрибуты и операции, которые проверяли бы правильность функционирования приборов.

Кроме того, необходимо идентифицировать данные, собранные со всех станций; таким образом, каждая метеостанция должна иметь собственный идентификатор.

Мы решили не создавать объекты, ассоциированные с активными объектами каждого прибора. Чтобы снять данные в нужное время, объекты приборов вызывают операцию **сбор** объекта **МетеоДанные**. Активные объекты имеют собственное управление, в

нашем примере предполагается, что каждый прибор сам определяет, когда нужно проводить замеры.

Но такой подход имеет недостаток: если принято решение изменить временной интервал сбора данных или если разные метеостанции собирают данные через разные промежутки времени, тогда необходимо вводить новые классы объектов.

Создавая объекты приборов, снимающие показания по запросу, любые изменения в стратегии сбора данных можно легко реализовать без изменения объектов, связанных с приборами.

Модели системной архитектуры показывают объекты и классы объектов, составляющих систему, и при необходимости типы взаимоотношений между этими объектами. Такие модели служат мостом между требованиями к системе и ее реализацией. А это значит, что к данным моделям предъявлены противоречивые требования. Они должны быть абстрактными настолько, чтобы лишние данные не скрывали отношения между моделью архитектуры и требованиями к системе. Однако, чтобы программист мог принимать решения по реализации, модель должна содержать достаточное количество информации.

Эти противоречие можно обойти, разработав несколько моделей разного уровня детализации. Там, где существуют тесные рабочие связи между разработчиками требований, проектировщиками и программистами, можно обойтись одной обобщенной моделью. В этом случае конкретные решения по архитектуре системы можно принимать в процессе реализации системы. Когда связи между разработчиками требований, проектировщиками и программистами не такие тесные (например, если система проектируется в одном подразделении организации, а реализуется в другом), требуется более детализированная модель.

Следовательно, в процессе проектирования важно решить, какие требуется модели и какой должна быть степень их детализации. Это решение зависит также от типа разрабатываемой системы. Систему последовательной обработки данных можно спроектировать на основе встроенной системы реального времени разными способами с использованием различных моделей архитектуры. Существует множество систем, которым требуются все виды моделей. Но уменьшение числа созданных моделей сокращает расходы и время проектирования.

Существует два типа объектно-ориентированных моделей системной архитектуры.

1. Статические модели, которые описывают статическую структуру системы в терминах классов объектов и взаимоотношений между ними. Основными взаимоотношениями, которые документируются на данном этапе, являются отношения обобщения, отношения "используют используются" и структурные отношения.
2. Динамические модели, которые описывают динамическую структуру системы и показывают взаимодействия между объектами системы (но не классами объектов). Документируемые взаимодействия содержат последовательность составленных объектами запросов к сервисам и описывают реакцию системы на взаимодействия между объектами.

В языке моделирования UML поддерживается огромное количество возможных статических и динамических моделей. Буч (Booch) предлагает девять различных типов схем для представления моделей. Чтобы показать все модели, не хватит места, да и не все из них пригодны для примера с метеостанцией. Здесь рассматриваются три типа моделей.

1. Модели подсистем, которые показывают логически сгруппированные объекты. Они представлены с помощью диаграммы классов, в которой каждая подсистема обозначается как пакет. Модели подсистем являются статическими.
2. Модели последовательностей, которые показывают последовательность взаимодействий между объектами. Они представляются в UML с помощью диаграмм последовательности или кооперативных диаграмм. Это динамические модели.
3. Модели конечного автомата, которые показывают изменение состояния отдельных объектов в ответ на определенные события. В UML они представлены в виде диаграмм состояния. Модели конечного автомата являются динамическими.

Другие типы моделей рассмотрены ранее в этой и предыдущих главах. Модели вариантов использования показывают взаимодействия с системой (рис. 7.7), модели объектов дают описание классов объектов (рис. 7.2), модели обобщения и

наследования (рис. 7.8) показывают, какие классы являются обобщениями других классов, модель агрегирования выявляет взаимосвязи между коллекциями объектов.

Модель подсистем является одной из наиболее важных и полезных статических моделей, поскольку показывает, как можно организовать систему в виде логически связанных групп объектов. Мы уже встречали примеры такого типа модели на рис. 7.6, где изображены подсистемы системы построения карт погоды. В UML пакеты являются структурами инкапсуляции и не отображаются непосредственно в объектах разрабатываемой системы. Однако они могут отображаться, например, в виде библиотек Java.

На рис. 7.10 показаны объекты подсистем метеостанции. В данной модели также представлены некоторые связи. Например, объект **КонтроллерКоммуникаций** связан с объектом **Метеостанция**, а объект **Метеостанция** связан с пакетом **Сбор данных**. Совместная модель пакетов и классов объектов позволяет показать логически сгруппированные системные элементы.

Рис. 7.10. Пакеты системы метеостанции

Модель последовательностей – одна из наиболее полезных и наглядных динамических моделей, которая в каждом узле взаимодействия документирует последовательность происходящих между объектами взаимодействий. Опишем основные свойства модели последовательности.

1. Объекты, участвующие во взаимодействии, располагаются горизонтально вверху диаграммы. От каждого объекта исходит пунктирная вертикальная линия – линия жизни объекта.
2. Время направлено сверху вниз по пунктирным вертикальным линиям. Поэтому в данной модели легко увидеть последовательность операций.
3. Взаимодействия между объектами представлены маркированными стрелками, связывающими вертикальные линии. Это не поток данных, а представление сообщений или событий, основных в данном взаимодействии.
4. Тонкий прямоугольник на линии жизни объекта обозначает интервал времени, в течение которого данный объект был управляющим объектом системы. Объект берет на себя управление в верхней части прямоугольника и передает управление другому объекту внизу прямоугольника. Если в системе имеется иерархия вызовов, то управление не передается до тех пор, пока не завершится последний возврат в вызове первоначального метода.

Сказанное выше проиллюстрировано на рис. 7.11, где изображена последовательность взаимодействий в тот момент, когда внешняя система посылает метеостанции запрос на получение данных. Диаграмму можно прокомментировать следующим образом.

1. Объект:**КонтроллерКоммуникаций**, являющийся экземпляром одноименного класса, получает внешний запрос "отправить отчет о погоде". Он подтверждает получение запроса. Половинная стрелка показывает, что, отправив сообщение, объект не ожидает ответа.
2. Этот объект отправляет сообщение объекту, который является экземпляром класса **Метеостанция**, чтобы создать метеорологический отчет. Объект:**КонтроллерКоммуникаций** затем приостанавливает работу. Используемый стиль стрелок показывает, что объекты:**КонтроллерКоммуникаций** и:**Метеостанция** могут выполняться параллельно.

3. Объект, который является экземпляром класса **Метеостанция**, отправляет сообщение объекту:**МетеоДанные**, чтобы подвести итоги по метеорологическим данным. Здесь другой стиль стрелок указывает на то, что объект:**Метеостанция** ожидает ответа.
4. После составления сводки, управление передается объекту:**Метеостанция**. Пунктирная стрелка обозначает возврат управления.
5. Этот объект передает сообщение объекту:**КонтроллерКоммуникаций**, из которого был прислан запрос, чтобы передать данные в удаленную систему. Затем объект:**Метеостанция** приостанавливает работу.
6. Объект:**КонтроллерКоммуникаций** передает сводные данные в удаленную систему, получает

подтверждение и затем переходит в состояние ожидания следующего запроса.

Рис. 7.11. Последовательность операций во время сбора данных

Из диаграммы последовательностей видно, что объекты **КонтроллерКоммуникаций** и **Метеостанция** в действительности являются параллельными процессами, выполнение которых может

приостанавливаться и снова возобновляться. Здесь существенно, что экземпляр объекта **КонтроллерКоммуникаций** получает сообщения от внешней системы, расшифровывает полученные сообщения и инициализирует действия метеостанции.

При документировании проекта для каждого значительного взаимодействия необходимо создавать диаграмму последовательностей. Если разрабатывается модель вариантов использования, то диаграмму последовательности нужно создавать для каждого заданного варианта.

Диаграммы последовательностей обычно применяются при моделировании комбинированного поведения групп объектов, однако при желании можно также показать поведение одного объекта в ответ на обрабатываемые им сообщения. В UML для описания моделей конечного автомата используются диаграммы состояний.

На рис. 7.12 представлена диаграмма состояния объекта **Метеостанция**, которая показывает реакцию объекта на запросы от разных сервисов.

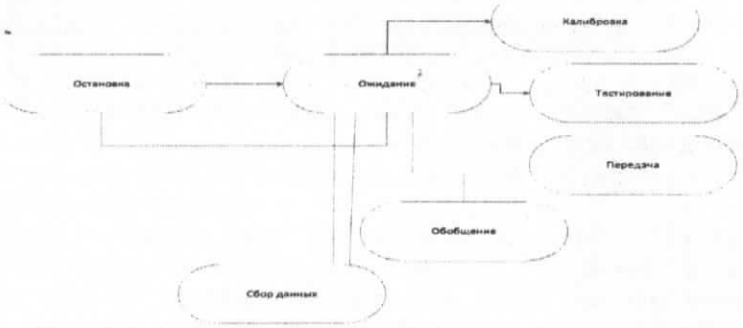


Рис. 7.12. Диаграмма состояний объекта Метеостанция

Эту диаграмму можно прокомментировать следующим образом.

1. Если объект находится в состоянии **Останов**, он может отреагировать только сообщением **запуск()**. Затем он переходит в состояние ожидания дальнейших сообщений. Немаркированная стрелка с черным кружком указывает на то, что это состояние является начальным.

2. В состоянии **Ожидание** система ожидает дальнейших сообщений. При получении сообщения **завершение()** объект возвращается в состояние завершения работы **Останов**.
3. Если получено сообщение **отчет()**, то система переходит в состояние обобщения данных **Обобщение**, а затем в состояние передачи данных **Передача**, в котором информация передается через объект **КонтроллерКоммуникаций**. Затем система возвращается в состояние ожидания.
4. Получив сообщение **калибровать()**, система последовательно проходит через состояния калибровки, тестирования, передачи и лишь после этого переходит в состояние ожидания. В случае получения сообщения **тестировать()**, система сразу переходит в состояние тестирования.
5. Если получен сигнал **время**, система переходит в состояние сбора данных, в котором она собирает данные от приборов. Каждый прибор по очереди также получает инструкцию "снять свои данные".

Обычно не нужно создавать диаграммы состояний для всех определенных в системе объектов. Большинство объектов относительно просты, и модель конечного автомата просто излишняя.

Важной частью любого процесса проектирования является специфицирование интерфейсов между различными компонентами системы. Интерфейсы необходимо определить так, чтобы объекты и другие компоненты можно было проектировать параллельно. Определив интерфейс, разработчики других объектов могут считать, что интерфейс уже реализован.

Одному объекту не обязательно должен соответствовать один интерфейс. Один и тот же объект может иметь несколько интерфейсов, причем каждый из них предлагает свой способ поддержки методов. Такая поддержка имеется непосредственно в Java, где интерфейсы объявляются отдельно от объектов и объекты "реализуют" интерфейсы. Другими словами, через один интерфейс можно получить доступ к набору объектов.

Проектирование интерфейсов объектов связано со спецификацией интерфейса в объекте или группе объектов. Под этим подразумевается определение сигнатур и семантик сервисов, которые поддерживаются этим объектом или группой объектов. В UML интерфейсы можно определить подобно диаграмме классов. Однако

раздела свойств там нет, поэтому в стандарте UML шаблон «интерфейс» следует включать в именную часть.

Я предпочитаю альтернативный подход, в котором при определении интерфейса применяется язык программирования.

В листинге 7.2 показана спецификация интерфейса на языке Java для метеостанции. По мере усложнения интерфейсов такой подход оказывается более эффективным, так как для обнаружения ошибок и противоречий в описании интерфейса можно воспользоваться средствами проверки синтаксиса, имеющимися в компиляторе языка программирования.

Из представленного описания видно, что некоторые методы могут использовать разное количество параметров. Например, метод завершение без параметров применяется к целой станции, а тот же метод с параметрами может отключить один прибор.

Листинг 7.2. Описание интерфейса метеостанции

```
interface Метеостанция {
    public void Метеостанция();
    public void запуск();
    public void запуск (Прибор i);
    public void завершение();
    public void
завершение(Прибор i) ;
public void
отчетПогода();
    public void тестировать();
    public void тестировать(Прибор i);
    public void калибровать(Прибор i);
    public int получитьИдНомер();
    }//Метеостанция
```

Главное преимущество объектно-ориентированного подхода к проектированию системы состоит в том, что он упрощает задачу внесения изменений в системную архитектуру, поскольку представление состояния объекта не оказывает на нее никакого влияния. Изменение внутренних данных объекта не должно влиять на другие объекты системы.

Более того, так как объекты слабо связаны между собой, обычно новые объекты просто вставляются без значительных воздействий на остальные компоненты системы.

Чтобы проиллюстрировать стабильность объектно-ориентированного подхода, предположим, что в каждую метеостанцию потребовалось добавить возможность наблюдения за степенью загрязнения окружающей среды, т.е. необходимо добавить приборы, измеряющие состав воздуха, чтобы вычислить количество различных загрязнителей.

Снятые измерения по загрязнению воздуха передаются с таким же интервалом времени, что и остальные метеорологические данные.

Для модификации проекта необходимо внести ряд изменений.

1. Класс объектов, именуемый **КачествоВоздуха** следует вставить как часть объекта **Метеостанция** на одном уровне с объектом **МетеоДанные**.

Рис. 7.13. Новые объекты для наблюдения за загрязнением воздуха

2. В объект **Метеостанция** необходимо добавить метод **отчетКачествоВоздуха**, чтобы информация о состоянии воздуха отправлялась на центральный компьютер. Программу управления метеостанцией необходимо изменить так, чтобы при получении запроса с верхнего уровня объекта **Метеостанция** осуществлялся автоматический сбор данных по загрязнению воздуха.

3. Необходимо добавить объекты, которые представляют типы приборов измеряющих степень загрязнения воздуха. В нашем примере можно добавить приборы, которые измеряли бы уровень оксида натрия, дыма и паров бензина.

На рис. 7.13 показан объект **Метеостанция** и новые объекты, добавленные в систему. За исключением самого верхнего уровня системы (объект **Метеостанция**) в имеющиеся объекты не потребовалось вносить изменений. Добавление в систему сбора данных о загрязнении воздуха не оказало никакого влияния на сбор метеорологических данных.

Контрольные вопросы

1. Что представляет из себя объектно-ориентированное проектирование?
2. Из каких этапов состоит объектно – ориентированный подход?
3. Что такое объектно-ориентированный анализ ?
4. Что такое объект?
5. Что обозначает UML?
6. Дайте определение понятию параллельные объекты.
7. Как проходит процесс объектно-ориентированного проектирования?

8 ГЛАВА. ПРОЕКТИРОВАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

8.1. Введение в проектирование интерфейса пользователя

На сегодня проектирование вычислительных систем охватывает широкий спектр проектных действий – от проектирования аппаратных средств до проектирования интерфейса пользователя. Организации разработчики часто нанимают специалистов для проектирования аппаратных средств и очень редко для проектирования интерфейсов.

Специалистам по разработке ПО иногда приходится проектировать и интерфейс пользователя. Если в больших компаниях в этот процесс вовлекаются специалисты по инженерной психологии, то в небольших компаниях услугами таких специалистов практически не пользуются.

Хорошо спроектированный интерфейс пользователя важен для успешной работы системы. Сложный в применении интерфейс приводит к ошибкам пользователя. Иногда они просто отказываются работать с программной системой, несмотря на ее функциональные возможности.

Если информация представляется плохо или непоследовательно, пользователи могут понять ее неправильно, в результате чего их последующие действия могут привести к повреждению данных или даже к сбою в работе системы.

Раньше интерфейсы пользователя были текстовыми или создавались в виде специальных форм. Сейчас почти все пользователи работают на персональных компьютерах.

Все современные персональные компьютеры поддерживают графический интерфейс пользователя (graphical user interface – GUI), который подразумевает использование цветного графического экрана с высоким разрешением и позволяет работать с мышью и с клавиатурой.

Текстовые интерфейсы еще достаточно широко применяются, особенно в наследуемых системах, в наше время пользователи предпочитают работать с графическим интерфейсом. В табл. 4 перечислены основные элементы GUI.

Таблица 4. Элементы графических интерфейсов пользователя

Элементы	Описание
Окна	Позволяют отображать на экране информацию разного рода
Пиктограммы	Представляют различные типы данных. В одних системах пиктограммы представляют файлы, в других – процессы
Меню	Ввод команд заменяется выбором команд из меню
Указатели	Мышь используется как устройство указания для выбора команд из меню и для выделения отдельных элементов в окне
Графические элементы	Могут использоваться совместно с текстовыми

Графические интерфейсы обладают рядом преимуществ.

1. Их относительно просто изучить и использовать. Пользователи, не имеющие опыта работы с компьютером, могут легко и быстро научиться работать с графическим интерфейсом.
2. Каждая программа выполняется в своем окне (экране). Можно переключаться из одной программы в другую, не теряя при этом данные, полученные в ходе выполнения программ.
3. Режим полноэкранного отображения окон дает возможность прямого доступа к любому месту экрана.

Разработчики и программисты обычно компетентны в использовании таких технологий, как классы Swing в языке Java или HTML, являющиеся основой реализации интерфейсов пользователя. Однако эту технологию далеко не всегда применяют надлежащим образом, в результате чего интерфейсы пользователя получаются неэлегантными, неудобными и сложными в использовании.

На рис. 8.1 изображен итерационный процесс проектирования пользовательского интерфейса. Наиболее эффективным подходом к проектированию интерфейса пользователя является разработка с применением моделирования пользовательских функций.

В начале процесса прототипирования создаются бумажные макеты интерфейса, затем разрабатываются экранные формы, моделирующие взаимодействие с пользователем.

Желательно, чтобы конечные пользователи принимали активное участие в процессе проектирования интерфейса. В одних случаях пользователи помогут оценить интерфейс; в других будут полноправными членами проектной группы.

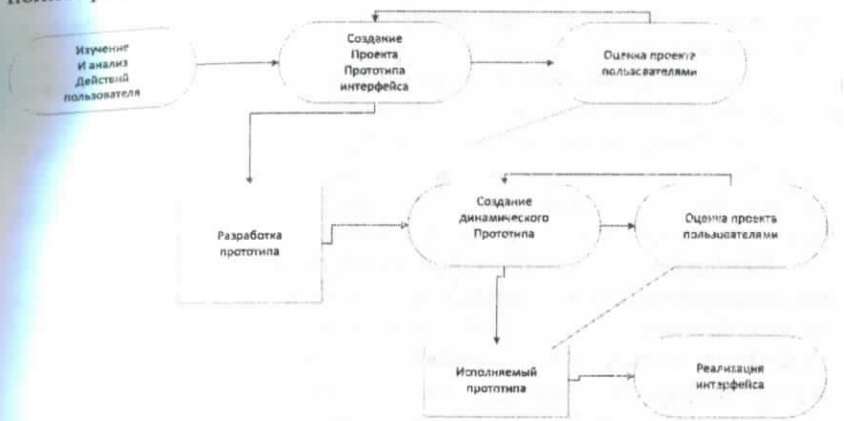


Рис. 8.1. Процесс проектирования интерфейса пользователя

Важным этапом процесса проектирования интерфейса пользователя является анализ деятельности пользователей, которую должна обеспечить вычислительная система. Не изучив того, что, с точки зрения пользователя, должна делать система, невозможно сформировать реалистический взгляд на проектирование эффективного интерфейса. Для анализа нужно применять различные методики, а именно: анализ задач, этнографический подход, опросы пользователей и наблюдения за их работой.

8.2. Принципы проектирования интерфейсов пользователя

Разработчики интерфейсов всегда должны учитывать физические и умственные способности людей, которые будут работать с программным обеспечением. Люди на короткое время могут запомнить весьма ограниченный объем информации и совершают ошибки, если приходится вводить вручную большие объемы данных или работать в напряженных условиях. Физические возможности людей могут существенно различаться, поэтому при

проектировании интерфейсов пользователя необходимо постоянно помнить об этом.

Основой принципов проектирования интерфейсов пользователя являются человеческие возможности. Ниже представлены основные принципы, применимые при проектировании любых интерфейсов пользователя.

Принцип учета знаний пользователя предполагает следующее: интерфейс должен быть настолько удобен при реализации, чтобы пользователям не понадобилось особых усилий, чтобы привыкнуть к нему. В интерфейсе должны использоваться термины, понятные пользователю, а объекты, управляемые системой, должны быть напрямую связаны с рабочей средой пользователя.

Например, если разрабатывается система, предназначенная для авиадиспетчеров, то управляемыми объектами в ней должны быть самолеты, траектории полетов, сигнальные знаки и т.п. Основную реализацию интерфейса в терминах файловых структур и структур данных необходимо скрыть от конечного пользователя.

Принцип согласованности интерфейса пользователя предполагает, что команды и меню системы должны быть одного формата, параметры должны передаваться во все команды одинаково и пунктуация команд должна быть схожей. Такие интерфейсы сокращают время на обучение пользователей. Знания, полученные при изучении какой-либо команды или части приложения, можно затем применить при работе с другими частями системы.

В данном случае речь идет о согласованности низкого уровня. И создатели интерфейса всегда должны стремиться к нему. Однако желательна согласованность и более высокого уровня. Например, удобно, когда для всех типов объектов системы поддерживаются одинаковые методы (такие, как печать, копирование и т.п.). Однако полная согласованность невозможна и даже нежелательна.

Например, операцию удаления объектов рабочего стола целесообразно реализовать посредством их перетаскивания в корзину. Но в текстовом редакторе такой способ удаления фрагментов текста кажется неестественным.

Всегда нужно соблюдать следующий принцип: количество неожиданностей должно быть минимальным, так как пользователей раздражает, когда система вдруг начинает вести себя непредсказуемо.

При работе с системой у пользователей формируется определенная модель ее функционирования. Если его действие в

одной ситуации вызывает определенную реакцию системы, естественно ожидать, что такое же действие в другой ситуации приведет к аналогичной реакции. Если же происходит совсем не то, что ожидалось, пользователь либо удивляется, либо не знает, что делать. Поэтому разработчики интерфейсов должны гарантировать, что схожие действия произведут похожий эффект.

Очень важен принцип восстанавливаемости системы, так как пользователи всегда допускают ошибки. Правильно спроектированный интерфейс может уменьшить количество ошибок пользователя (например, использование меню позволяет избежать ошибок, которые возникают при вводе команд с клавиатуры), однако все ошибки устранить невозможно.

В интерфейсах должны быть средства, по возможности предотвращающие ошибки пользователя, а также позволяющие корректно восстановить информацию после ошибок. Эти средства бывают двух видов.

1. *Подтверждение деструктивных действий.* Если пользователь выбрал потенциально деструктивную операцию, то он должен еще раз подтвердить свое намерение.
2. *Возможность отмены действий.* Отмена действия возвращает систему в то состояние, в котором она находилась до их выполнения. Не лишней будет поддержка многоуровневой отмены действий, так как пользователи не всегда сразу понимают, что совершили ошибку.

Следующий принцип – поддержка пользователя. Средства поддержки пользователей должны быть встроены в интерфейс и систему и обеспечивать разные уровни помощи и справочной информации. Должно быть несколько уровней справочной информации – от основ для начинающих до полного описания возможностей системы. Справочная система должна быть структурированной и не перегружать пользователя излишней информацией при простых запросах к ней.

Принцип учета разнородности пользователей предполагает, что с системой могут работать разные их типы. Часть пользователей работает с системой нерегулярно, время от времени. Но существует и другой тип– "опытные пользователи", которые работают с приложением каждый день по несколько часов.

Случайные пользователи нуждаются в таком интерфейсе, который "руководил" бы их работой с системой, в то время как

опытным пользователям требуется интерфейс, который позволил бы им максимально быстро взаимодействовать с системой.

Некоторые пользователи могут иметь разные физические недостатки, в интерфейсе должны быть средства, которые помогли бы им перенастроить интерфейс под себя. Это могут быть средства, позволяющие отображать увеличенный текст, замещать звук текстом, создавать кнопки больших размеров и т.п.

Принцип признания многообразия категорий пользователей может противоречить другим принципам проектирования интерфейсов, например согласованности интерфейса. Аналогично, необходимый уровень справочной информации для разных типов пользователей может радикально отличаться. Невозможно создать такую справочную систему, которая подошла бы всем пользователям. Разработчик интерфейса должен всегда быть готовым к компромиссным решениям в зависимости от реальных пользователей системы.

8.3. Взаимодействие с пользователем

Разработчику интерфейса пользователя вычислительных систем необходимо решить две главные задачи: каким образом пользователь будет вводить данные в систему и как данные будут представлены пользователю. "Правильный" интерфейс должен обеспечивать и взаимодействие с пользователем, и представление информации.

Интерфейс пользователя обеспечивает ввод команд и данных в вычислительную систему. На первых вычислительных машинах был только один способ ввода данных – через интерфейс командной строки, причем для взаимодействия с машиной использовался специальный командный язык. Такой способ годился только для опытных пользователей, поэтому позже были разработаны более упрощенные способы ввода данных. Все эти виды взаимодействия можно отнести к одному из пяти основных стилей взаимодействия.

1. *Непосредственное манипулирование.* Пользователь взаимодействует с объектами на экране. Например, для удаления файла пользователь просто перетаскивает его в корзину.

2. *Выбор из меню.* Пользователь выбирает команду из списка пунктов меню. Очень часто выбранная команда воздействует только на тот объект, который выделен (выбран)

на экране. При таком подходе для удаления файла пользователь сначала выбирает файл, а затем команду на удаление.

3. *Заполнение форм.* Пользователь заполняет поля экранной формы. Некоторые поля могут иметь свое меню (выпадающее меню или списки). В форме могут быть командные кнопки, при щелчке мышью на которых инициируют некоторое действие. Чтобы удалить файл с помощью интерфейса, основанного на форме, надо ввести в поле формы имя файла и затем щелкнуть на кнопке удаления, присутствующей в форме.

4. *Командный язык.* Пользователь вводит конкретную команду с параметрами, чтобы указать системе, что она должна дальше делать. Чтобы удалить файл, пользователь вводит команду удаления с именем файла в качестве параметра этой команды.

5. *Естественный язык.* Пользователь вводит команду на естественном языке. Чтобы удалить файл, пользователь может ввести команду "удалить файл с именем XXX".

Каждый из этих стилей взаимодействия имеет преимущества и недостатки и наилучшим образом подходит разным типам приложений и различным категориям пользователей.

Конечно, стили взаимодействия редко используются в чистом виде, в одном приложении может использоваться одновременно несколько разных стилей. Например, в операционной системе Microsoft Window поддерживается несколько стилей: прямое манипулирование пиктограммами, представляющими файлы и папки, выбор команд из меню, ручной ввод некоторых команд, таких как команды конфигурирования системы, использование форм (диалоговых окон).

Пользовательские интерфейсы приложений World Wide Web базируются на средствах, предоставляемых языком HTML вместе с другими языками, например Java, который связывает программы с компонентами Web-страниц.

В основном интерфейсы Web страниц проектируются для случайных пользователей и представляют собой интерфейсы в виде форм. В Web-приложениях можно создавать интерфейсы, в которых применялся бы стиль прямого манипулирования, однако к моменту написания книги проектирование таких интерфейсов представляло достаточно сложную в аспекте программирования задачу.

В принципе необходимо применять различные стили взаимодействия для управления разными системными объектами. Данный принцип составляет основу модели Сихейма (Seeheim) пользовательских интерфейсов.

В этой модели разделяются представление информации, управление диалоговыми средствами и управление приложением. На самом деле такая модель является скорее идеальной, чем практической, однако почти всегда есть возможность разделить интерфейсы для разных классов пользователей.

На рис. 8.2 изображена подобная модель с разделенными интерфейсом командного языка и графическим интерфейсом, лежащая в основе некоторых операционных систем, в частности Linux.

Рис. 8.2. Множественный интерфейс

Разделение представления, взаимодействия и объектов, включенных в интерфейс пользователя, является основным принципом подхода "модель-представление-контроллер", который обсуждается в следующем разделе. Эта модель сравнима с моделью Сихейма, однако используется при реализации отдельных объектов интерфейса, а не всего приложения.

8.4. Представление информации

В любой интерактивной системе должны быть средства для представления данных пользователям. Данные в системе могут отображаться по-разному: например, вводимая информация может отображаться непосредственно на дисплее или преобразовываться в графическую форму. Хорошим тоном при проектировании систем считается отделение представления данных от самих данных.

До некоторой степени разработка такого ПО противоречит объектно-ориентированному подходу, при котором методы, выполняемые над данными, должны быть определены самими данными.

Однако в нашем случае предполагается, что разработчик объектов всегда знает наилучший способ представления данных; хотя это, конечно, не всегда так. Часто определить наилучший способ представления данных конкретного типа довольно трудно, в таком случае объектные структуры не должны быть "жесткими".

После того как представление данных в системе отделено от самих данных, изменения в представлении данных на экране пользователя происходят без изменения самой системы (рис. 8.3).

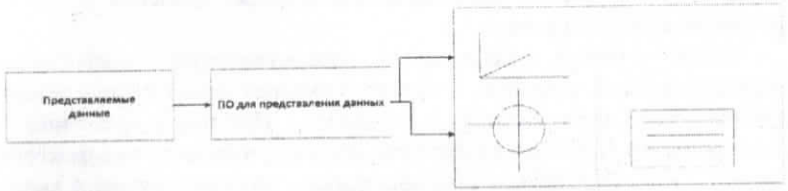


Рис. 8.3. Представление данных

Подход "модель-представление-контроллер" (МПК), представленный на рис. 5.4, получил первоначальное применение в языке Smalltalk как эффективный способ поддержки различных представлений данных. Пользователь может взаимодействовать с каждым типом представления. Отображаемые данные инкапсулированы в объекты модели.

Каждый объект модели может иметь несколько отдельных объектов представлений, где каждое представление – это разные отображения модели. Я уже иллюстрировал этот подход в

предыдущей главе, где рассматривались объектно ориентированные структуры приложений.

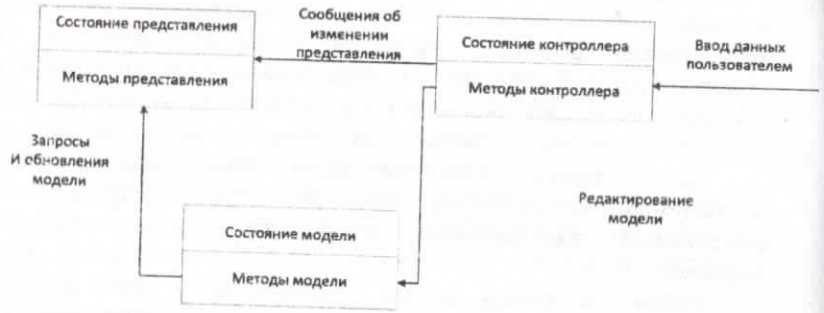


Рис. 8.4. Модель МПК взаимодействия с пользователем

Каждое представление имеет связанный с ним объект контроллера, который обрабатывает введенные пользователем данные и обеспечивает взаимодействие с устройствами. Такая модель может представить числовые данные, например, в виде диаграмм или таблиц. Модель можно редактировать, изменяя значения в таблице или параметры диаграммы.

Чтобы найти наилучшее представление информации, необходимо знать, с какими данными работают пользователи и каким образом они применяются в системе. Принимая решение по представлению данных, разработчик должен учитывать ряд факторов.

1. Что нужно пользователю – точные значения данных или соотношения между значениями?
2. Насколько быстро будут происходить изменения значений данных? Нужно ли немедленно показывать пользователю изменение значений?
3. Должен ли пользователь предпринимать какие-либо действия в ответ на изменение данных?
4. Нужно ли пользователю взаимодействовать с отображаемой информацией посредством интерфейса с прямым манипулированием?
5. Информация должна отображаться в текстовом (описательно) или числовом формате? Важны ли относительные значения элементов данных?

Если данные не изменяются в течение сеанса работы с системой, их можно представить либо в графическом, либо в текстовом виде, в зависимости от типа приложения. Текстовое представление данных занимает на экране мало места, но в таком случае данные нельзя охватить одним взглядом. С помощью разных стилей представления неизменяемые данные следует отделить от динамически изменяющихся данных. Например, статические данные можно выделить особым шрифтом, подчеркнуть особым цветом либо обозначить пиктограммами.

Если требуется точная цифровая информация и данные изменяются относительно медленно, их можно отображать в текстовом виде. Там, где данные изменяются быстро, обычно используется графическое представление.

В качестве примера рассмотрим систему, которая помесячно записывает и подбивает итоги по данным продаж некой компании. На рис. 8.5 видно, что одни и те же данные можно представить в виде текста и в графическом виде.

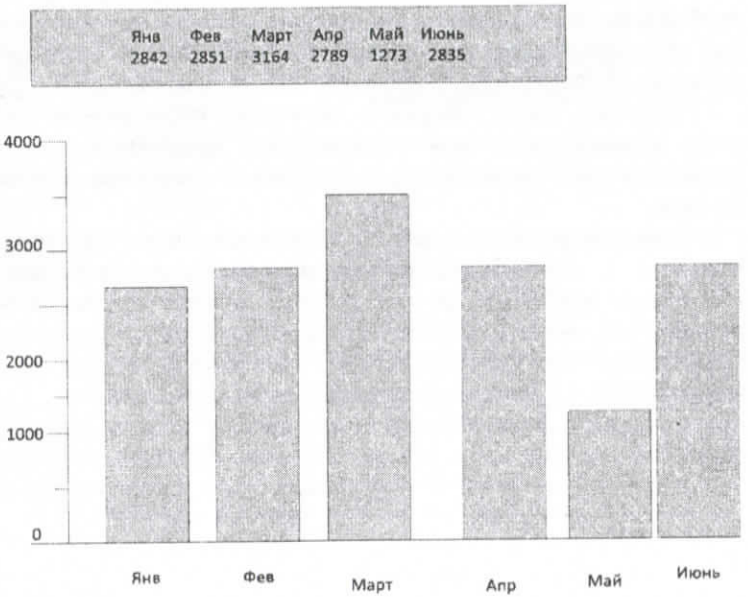


Рис. 8.5. Альтернативные представления данных

Менеджерам, изучающим данные о продажах, обычно больше нужны тенденции изменения или аномальные данные, чем их точные значения. Графическое представление этой информации в виде гистограммы позволяет выделить аномальные данные за март и май, значительно отличающиеся от остальных данных.

Как видно из рис. 8.5, данные в текстовом представлении занимают меньше места, чем в графическом.

Динамические изменения числовых данных лучше отображать графически, используя аналоговые представления. Постоянно изменяющиеся цифровые экраны сбивают пользователей с толку, поскольку точные значения данных быстро не воспринимаются.

Графическое отображение данных при необходимости можно дополнить точными значениями. Различные способы представления изменяющихся числовых данных показаны на рис. 8.6.

Непрерывные аналоговые отображения помогают наблюдателю оценить относительные значения данных.

На рис. 8.7 числовые значения температуры и давления приблизительно одинаковы. Но при графическом отображении видно, что значение температуры близко к максимальному, в то время как значение давления не достигло даже 25% от максимума.

Обычно, кроме текущего значения, наблюдателю требуется знать максимальные (или минимальные) возможные значения. Он должен в уме вычислять относительное состояние считываемых данных.

Дополнительное время, необходимое для расчетов, может привести к ошибкам оператора в стрессовых ситуациях, когда возникают проблемы и на дисплее отображаются аномальные данные.

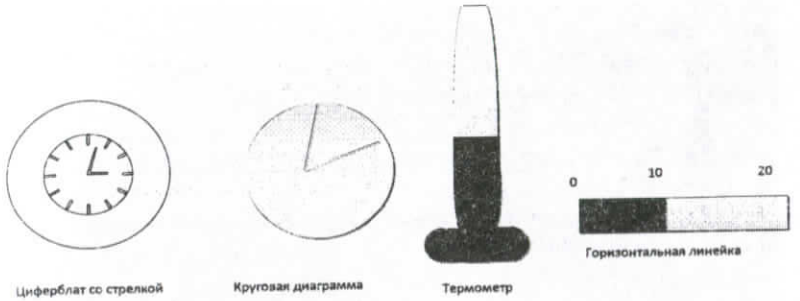


Рис. 8.6. Способы представления динамически изменяющихся числовых данных

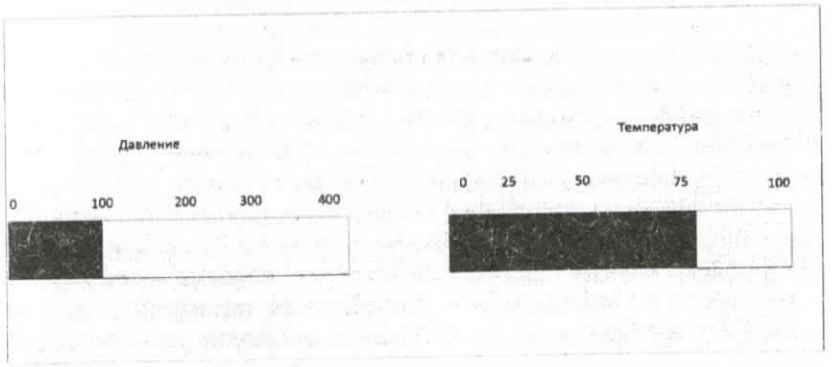


Рис. 8.7. Графическое представление данных, показывающее значения по отношению к максимальным

При представлении точных буквенно-цифровых данных для выделения особой информации можно использовать графические элементы. Вместо обычной строки данные лучше поместить в прямоугольник или отметить пиктограммой (рис. 8.8). Прямоугольник с сообщением помещается поверх текущего экрана, тем самым привлекая к нему внимание пользователя.

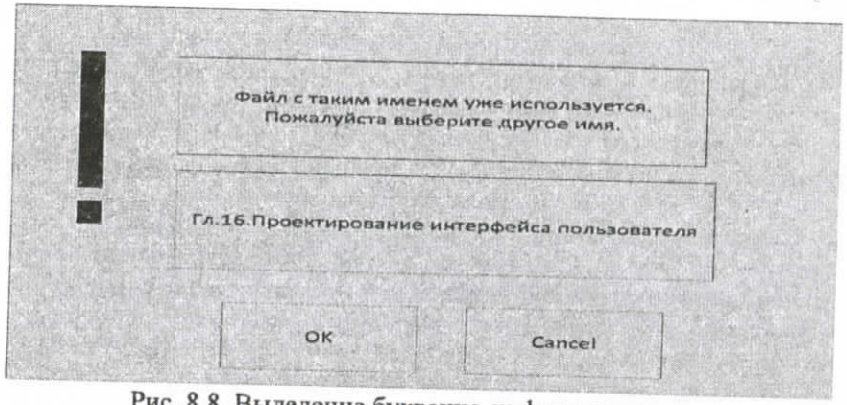


Рис. 8.8. Выделение буквенно-цифровых данных

Выделение информации с помощью графических элементов можно также использовать для привлечения внимания к изменениям, происходящим в разных частях экрана. Но, если изменения происходят очень быстро, не следует использовать графические элементы, поскольку быстрые изменения могут привести к наложению экранов, что сбивает с толку и раздражает пользователей.

При представлении больших объемов данных можно использовать разные приемы визуализации, которые указывают на родственные элементы данных. Разработчики интерфейсов должны помнить о возможностях визуализации, особенно если интерфейс системы должен отображать физические сущности (объекты). Вот несколько примеров визуализации данных.

1. Отображение метеорологических данных, собранных из разных источников, в виде метеорологических карт с изобарами, воздушными фронтами и т.п.
2. Графическое отображение состояния телефонной сети в виде связанного множества узлов.
3. Визуализация состояния химического процесса с показом давлений и температур в группе связанных между собой резервуаров и труб.
4. Модель молекулы и манипулирование ею в трехмерном пространстве посредством системы виртуальной реальности.
5. Отображение множества Web-страниц в виде дерева гипертекстовых ссылок.

Во всех интерактивных системах, независимо от их назначения, поддерживаются цветные экраны, поэтому в пользовательских интерфейсах часто используются различные цвета. В некоторых системах цвета применяют в основном для выделения определенных элементов; в других системах цветами обозначают разные уровни проектов.

Правильное использование цветов делает интерфейс пользователя более удобным для понимания и управления. Вместе с тем использование цветов может быть неправильным, в результате чего создаются интерфейсы, которые визуально неприглядны и даже провоцируют ошибки. Основным принципом разработчиков интерфейсов должно быть осторожное использование цветов на экранах. 14 правил эффективного использования цвета в пользовательских интерфейсах. Вот наиболее важные из них.

1. *Используйте ограниченное количество цветов.* Для окон не следует использовать более четырех или пяти разных цветов, в интерфейсе системы не должно быть более семи цветов.
2. *Используйте разные цвета для показа изменений в состоянии системы.* Если на экране изменились цвета, значит, произошло какое-то событие. Выделение цветом особенно важно в сложных экранах, в которых отображаются сотни разных объектов.
3. *Для помощи пользователю используйте цветовое кодирование.* Если пользователям необходимо выделять аномальные элементы, выделите их цветом; если требуется найти подобные элементы, выделите их одинаковым цветом.
4. *Используйте цветовое кодирование продуманно и последовательно.* Если в какой-либо части системы сообщения об ошибке отображаются, например, красным цветом, то во всех других частях подобные сообщения должны отображаться таким же цветом. Тогда красный цвет не следует использовать где-либо еще. Если же красный цвет используется еще где-то в системе, пользователь может интерпретировать появление красного цвета как сообщение об ошибке. Следует помнить, что у определенных типов пользователей имеются свои представления о значении отдельных цветов.
5. *Осторожно используйте дополняющие цвета.* Физиологические особенности человеческого глаза не

позволяют одновременно сфокусироваться на красном и синем цветах. Поэтому последовательность красных и синих изображений вызывает зрительное напряжение. Некоторые комбинации цветов также могут визуально нарушать или затруднять чтение.

Чаще всего разработчики интерфейсов допускают две ошибки: привязка значения к определенному цвету и использование большого количества цветов на экране. Использовать цвета для представления значения не следует по двум причинам. Около 10% людей имеют нечеткое представление о цветах и поэтому могут неправильно интерпретировать значение.

У разных групп людей различное восприятие цветов; кроме того, в разных профессиях существуют свои соглашения о значении отдельных цветов. Пользователи на основании полученных знаний могут неадекватно интерпретировать один и тот же цвет. Например, водителем красный цвет воспринимается как *опасность*. А у химика красный цвет означает *горячий*.

При использовании слишком ярких цветов или слишком большого их количества отображения становятся путанными. Многообразие цветов сбивает с толку пользователя и вызывает у него зрительное утомление. Непоследовательное использование цветов также дезориентирует пользователя.

8.5. Средства поддержки пользователя

Мы предлагали принцип проектирования, согласно которому интерфейс пользователя должен всегда обеспечивать некоторый тип оперативной справочной системы. Справочные системы – один из основных аспектов проектирования интерфейса пользователя. Справочную систему приложения составляют:

- сообщения, генерируемые системой в ответ на действия пользователя;
- диалоговая справочная система;
- документация, поставляемая с системой.

Поскольку проектирование полезной и содержательной информации для пользователя – дело весьма серьезное, оно должно оцениваться на том же уровне, что и архитектура системы или программный код. Проектирование сообщений требует

значительного времени и немалых усилий. Уместно привлекать к этому процессу профессиональных писателей и художников-графиков. При проектировании сообщений об ошибках или текстовой справки необходимо учитывать факторы, перечисленные в табл. 6.

Таблица 6. Факторы проектирования текстовых сообщений

Фактор	Описание
Содержание	Справочная система должна знать, что делает пользователь, и реагировать на его действия сообщениями соответствующего содержания
Опыт пользователя	Если пользователи хорошо знакомы с системой, им не нужны длинные и подробные сообщения. В то же время начинающим пользователям такие сообщения покажутся сложными, малопонятными и слишком краткими. В справочной системе должны поддерживаться оба типа сообщений, а также должны быть средства, позволяющие пользователю управлять сложностью сообщений
Профессиональный уровень пользователя	Сообщения должны содержать сведения, соответствующие профессиональному уровню пользователей. В сообщениях для пользователей разного уровня необходимо применять разную терминологию
Стиль сообщений	Сообщения должны иметь положительный, а не отрицательный оттенок. Всегда следует использовать активный, а не пассивный тон обращения. В сообщениях не должно быть оскорблений или попыток пошутить
Культура	Разработчик сообщений должен быть знаком с культурой той страны, где продается система. Сообщение, вполне уместное в культуре одной страны, может оказаться неприемлемым в другой

Пользователь получает при работе с программной системой, основывается на сообщениях об ошибках. Неопытные пользователи, совершив ошибку, должны понять появившееся сообщение об ошибке.

Новички и опытные пользователи должны предвидеть ситуации, при которых могут возникнуть сообщения об ошибках. Например, пусть пользователем системы является медсестра госпиталя, работающая в отделении интенсивной терапии. Обследование пациентов выполняется на соответствующем оборудовании, связанном с вычислительной системой. Чтобы просмотреть текущее состояние пациента, пользователь системы выбирает пункт меню Показать и набирает имя пациента в поле ввода (рис. 8.9).

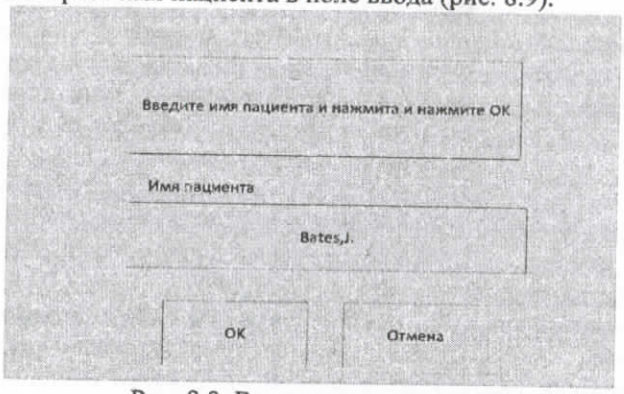


Рис. 8.9. Ввод имени пациента

Пусть медсестра ввела имя пациента Bates, вместо Pates. Система не находит пациента с таким именем и генерирует сообщение об ошибке. Сообщения об ошибке должны быть всегда вежливыми, краткими, последовательными и конструктивными, не содержать оскорблений. Не следует также использовать звуковые сигналы или другие звуки, которые могут сбить с толку пользователя. Неплохо включить в сообщения варианты исправления ошибки. Сообщение об ошибке должно быть связано с контекстно-зависимой справкой.

На рис. 5.10 показаны примеры двух сообщений об ошибке. Сообщение, расположенное слева, спроектировано плохо. Оно негативно (обвиняет пользователя в совершении ошибки), не

адаптировано к уровню знаний и опытности пользователя, не учитывает содержания ошибки.

В этом сообщении не предлагаются способы исправления сложившейся ситуации. Кроме того, в сообщении использованы специфические термины (номер ошибки), не понятные пользователю. Сообщение справа гораздо лучше. Оно положительно, в нем используются медицинские термины и предлагается простой способ исправления ошибки посредством щелчка на одной из кнопок. В случае необходимости пользователь может вызвать справку.

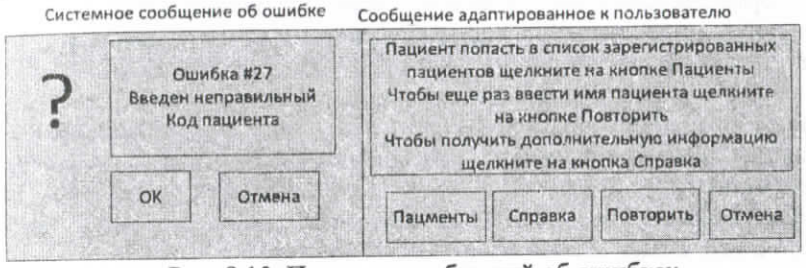


Рис. 8.10. Примеры сообщений об ошибках

При получении сообщения об ошибке пользователь часто не знает, что делать, и обращается к справочной системе за информацией. Справочная система должна предоставлять разные типы информации: как ту, что помогает пользователю в затруднительных ситуациях, так и конкретную информацию, которую ищет пользователь. Для этого справочная система должна иметь разные средства и разные структуры сообщений.

Справочная система должна обеспечивать пользователю несколько различных точек входа (рис. 8.11). Пользователь может войти в нее на верхнем уровне ее иерархической структуры и здесь обозреть все разделы справочной информации. Другие точки входа в справочную систему – с помощью окон сообщений об ошибках или путем получения описания конкретной команды приложения.

Все справочные системы имеют сложную сетевую структуру, в которой каждый раздел справочной информации может ссылаться на несколько других информационных разделов. Структура такой сети, как правило, иерархическая, с перекрестными ссылками, как показано на рис. 8.11. Наверху структурной иерархии содержится общая информация, внизу – более подробная.

При использовании справочной системы возникают проблемы, связанные с тем, что пользователи входят в сеть после совершения ошибки и затем перемещаются по сети справочной системы. Через некоторое время они запутываются и не могут определить, в каком месте справочной системы находятся. В такой ситуации пользователи должны завершить сеанс работы со справочной системой и вновь начать работу с некоторой известной точки справочной сети.

Отображение справочной информации в нескольких окнах упрощает подобную ситуацию. На рис. 8.12 показан экран, на котором расположены три окна справки. Однако пространство экрана всегда ограничено и разработчику следует помнить, что дополнительные окна могут скрыть другую нужную информацию.

Тексты справочной системы необходимо готовить совместно с разработчиками приложения. Справочные разделы не должны быть просто воспроизведением руководства пользователя, поскольку информация на бумаге и на экране воспринимается по-разному. Сам текст должен быть тщательно продуман, чтобы его можно было прочитать в окне относительно малого размера.

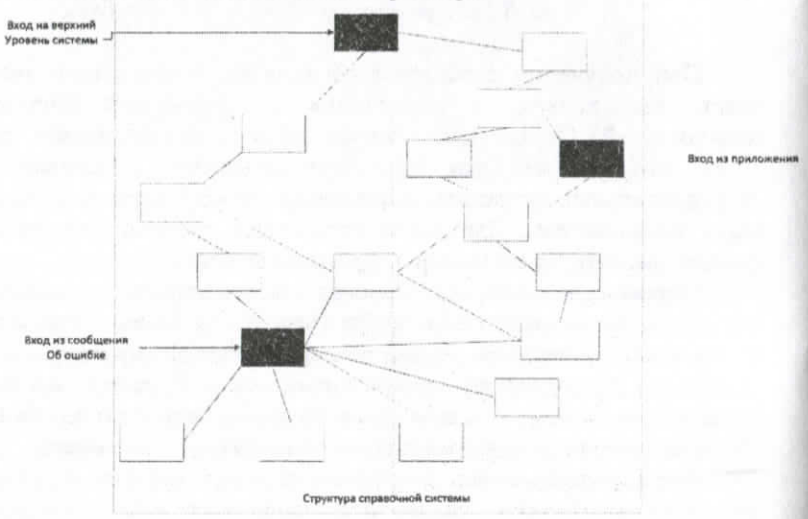


Рис. 8.11. Точки входа в справочную систему

Раздел справки **Переадресация** почты на рис. 8.12 сравнительно небольшой – в любом справочном разделе должна быть только самая

необходимая информация. В окне, отображающем справочный раздел, расположены три кнопки: для показа дополнительной информации, для перемещения по тексту раздела и для вызова списка справочных тем.

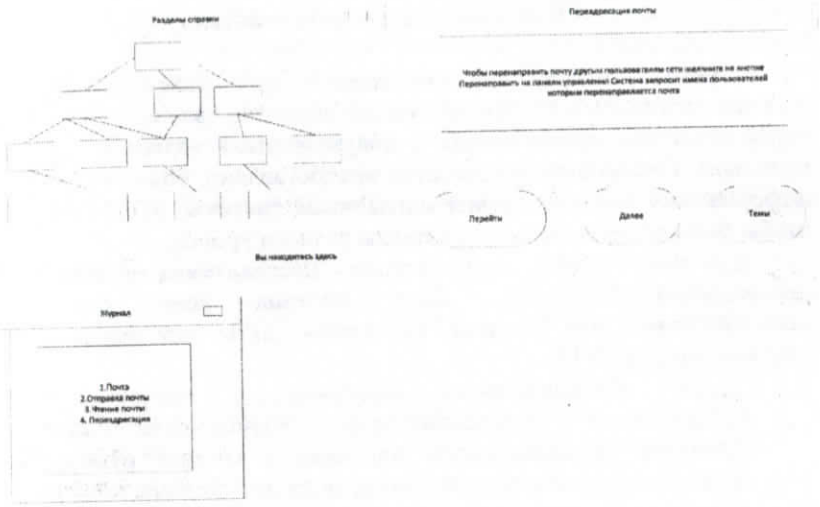


Рис. 8.12. Окна справочной системы

В окне **Журнал** показан список уже просмотренных разделов. Можно вернуться в один из них, выбрав соответствующий пункт из списка. Окно навигации по справочной системе – это графическая "карта" сети справочной системы. Текущая позиция на карте должна быть выделена цветом, тенями или, как в нашем случае, подписью.

У пользователей есть несколько возможностей перемещения между разделами справочной системы: можно перейти к разделу непосредственно из отображаемого раздела, можно выбрать нужный раздел из окна **Журнал**, чтобы просмотреть его еще раз, и, наконец, можно выбрать соответствующий узел на карте справочной сети и перейти к этому узлу.

Справочную систему можно реализовать в виде группы связанных Web-страниц или с помощью обобщенной гипертекстовой системы, интегрированной с приложением. Иерархическая структура

легко реализуется в виде гипертекстовых ссылок. Web-системы имеют преимущества: они просты в реализации и не требуют специального программного обеспечения. Однако при создании контекстно зависимой справки могут возникнуть трудности при связывании ее с приложением.

8.6. Документация пользователя

Строго говоря, документация не является частью пользовательского интерфейса, однако проектирование оперативной справочной поддержки вместе с документацией является хорошим правилом. Системные руководства предоставляют более подробную информацию, чем диалоговые справочные системы, и строятся так, чтобы быть полезными пользователям разного уровня.

Для того чтобы документация, поставляемая совместно с программной системой, была полезна всем системным пользователям, она должна содержать пять описанных ниже документов (рис. 8.13).

1. *Функциональное описание,* в котором кратко представлены функциональные возможности системы. Прочитав функциональное описание и вводное руководство, пользователь должен определить, та ли это система, которая ему нужна.
2. *Документ по инсталляции системы, в* котором содержится информация по установке системы. Здесь должны быть сведения о дисках, на которых поставляется система, описание файлов, находящихся на этих дисках, и минимальные требования к конфигурации. В документе должна быть инструкция по инсталляции и более подробная информация по установке файлов, зависящих от конфигурации системы.
3. *Вводное руководство,* представляющее неформальное введение в систему, описывающее ее "повседневное" использование. В этом документе должна содержаться информация о том, как начать работу с системой, как использовать общие возможности системы. Все описания должны быть снабжены примерами и содержать сведения о том, как восстановить систему после ошибки и как начать заново работу.

4. *Справочное руководство,* в котором описаны возможности системы и их использование, представлен список сообщений об ошибках и возможные причины их появления, рассмотрены способы восстановления системы после выявления ошибок.
5. *Руководство администратора,* необходимое для некоторых типов программных систем. В нем дано описание сообщений, генерируемых системой при взаимодействии с другими системами, и описаны способы реагирования на эти сообщения. Если в систему включена аппаратная часть, то в руководстве администратора должна быть информация о том, как выявить и устранить неисправности, связанные с аппаратурой, как подключить новые периферийные устройства и т.п.

Рис. 8.13. Типы пользовательской документации

Вместе с перечисленными руководствами необходимо предоставлять другую удобную в работе документацию. Для опытных пользователей системы удобны разного вида предметные указатели, которые помогают быстро просмотреть список возможностей системы и способы их использования.

8.7. Оценивание интерфейса

Это процесс, в котором оценивается удобство использования интерфейса и степень его соответствия требованиям пользователя. Таким образом, оценивание интерфейса является частью общего процесса тестирования и аттестации систем ПОЛ.

В идеале оценивание должно проводиться в соответствии с показателями удобства использования интерфейса, перечисленными в табл. 7. Каждый из этих показателей можно оценить численно. Например, изучаемость можно оценить следующим образом: опытный оператор после трехчасового обучения должен уметь использовать 80% функциональных возможностей системы. Однако чаще удобство использования интерфейса оценивается качественно, а не через числовые показатели.

Таблица 7. Показатели удобства использования интерфейса

Показатель	**Описание**
Изучаемость	Количество времени обучения, необходимое для начала продуктивной работы с системой Скорость работы Скорость реакции системы на действия пользователя
Устойчивость	Устойчивость системы к ошибкам пользователя
Восстанавливаемость	Способность системы восстанавливаться после ошибок пользователя
Адаптируемость	Способность системы "подстраиваться" к разным стилям работы пользователей

Полное оценивание пользовательского интерфейса может оказаться весьма дорогостоящим, в этот процесс будут вовлечены специалисты по когнитивной психологии и дизайнеры.

В процесс оценивания могут входить разработка и выполнение ряда статистических экспериментов с пользователями в специально созданных лабораториях и с необходимым для наблюдения оборудованием. Такое оценивание интерфейса экономически нерентабельно для систем, разрабатываемых в небольших организациях с ограниченными ресурсами.

Существуют более простые и менее дорогостоящие методики оценивания интерфейсов пользователя, позволяющие выявить отдельные дефекты в интерфейсах.

1. Анкеты, в которых пользователь дает оценку интерфейсу.

2. Наблюдения за работой пользователей с последующим обсуждением их способов использования системы при решении конкретных задач.
3. Видеонаблюдения типичного использования системы.
4. Добавление в систему программного кода, который собирал бы информацию о наиболее часто используемых системных сервисах и наиболее распространенных ошибках.

Анкетирование пользователей – относительно дешевый способ оценки интерфейса. Вопросы должны быть точными, а не общими. Не следует использовать вопросы типа "Пожалуйста, прокомментируйте практичность системы", так как ответы, вероятно, будут существенно различаться. Лучше задавать конкретные вопросы, например: "Оцените понятность сообщений об ошибках по шкале от 1 до 5. Оценка 1 означает полностью понятное сообщение, 5 – малопонятное". На такие вопросы легче ответить и более вероятно получить в результате полезную для улучшения интерфейса информацию.

Во время заполнения анкеты пользователи должны обязательно оценить собственный опыт и знания. Такого рода сведения позволят разработчикам зафиксировать, пользователи с каким уровнем знаний имеют проблемы с интерфейсом. Если проект интерфейса уже создан и прошел оценивание в бумажном виде, анкеты можно использовать даже до полной реализации системы.

При наблюдении пользователей за работой оценивается, как они взаимодействуют с системой, какие используют сервисы, какие совершают ошибки и т.п. Вместе с наблюдениями могут проводиться семинары, на которых пользователи рассказывают о своих попытках решить те или иные проблемы и о том, как они понимают систему и как используют ее для достижения целей.

Видеооборудование относительно недорого, поэтому к непосредственному наблюдению можно добавить видеозапись пользовательских семинаров для последующего анализа. Полный анализ видеоматериалов дорогостоящий и требует специально оснащенного комплекта с несколькими камерами, направленными на пользователя и на экран.

Однако видеозапись отдельных действий пользователя может оказаться полезной для обнаружения проблем. Чтобы определить, какие именно действия вызывают проблемы у пользователя, следует прибегнуть к другим методам оценивания.

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.

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 и совокупностью этих элементов. В объектно-ориентированных системах этого нет. Объекты могут быть простыми элементами, например списком, или сложными, например такими, как объект метеорологической станции 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, часто не подходят для объектно ориентированных систем.

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.

9.2. 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. 6.2. 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.

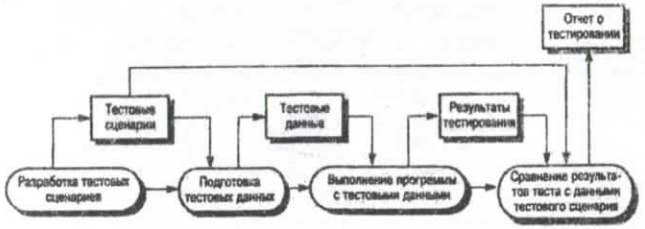


Рис. 9.2. Процесс тестирования дефектов

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.
3. Если в системе предполагается ввод пользователем каких-либо входных данных, тестируются функции с правильным и неправильным вводом данных.

Из опыта тестирования 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.

9.3. 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.

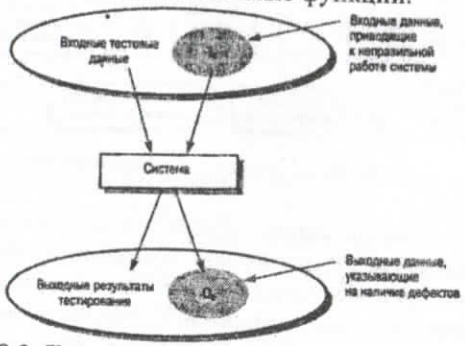


Рис. 9.3. Тестирование методом черного ящика

На рис. 6.3 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_e. 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.

9.4. Структурное тестирование

Метод структурного тестирования (рис. 6.4) 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.9.4. Структурное тестирование

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, 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
botto
m = 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.fou
nd = true;
return; }
//часть if
else
{ if
(elemArray[mid
] < key) bottom
= mid + 1;
e
lse
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. 9.5). При проверке программы в качестве тестовых данных необходимо взять последовательности с ключевыми элементами, расположенными на границах этих областей.

Рис. 9.5. Классы эквивалентности для бинарного поиска

9.5. 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 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 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. 9.6 показаны циклы и ветвления в графе потоков управления программы бинарного поиска.

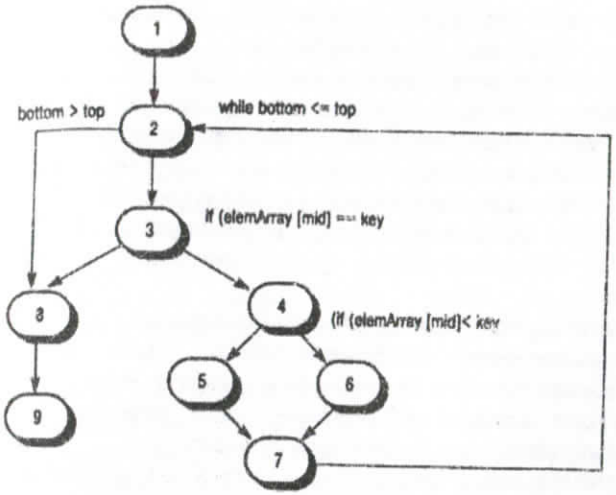


Рис. 9.6. Граф потоков управления программы бинарного поиска

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 C 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, 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.

9.6. 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. 9.7 последовательность тестов Tl, T2 и T3 сначала выполняется в системе, состоящей из модулей A и B. Если во время тестирования обнаружены дефекты, они исправляются. Затем в систему добавляется модуль C. 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.

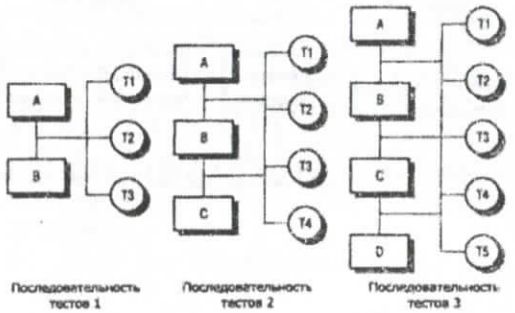


Рис. 9.7. Тестирование сборки

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. 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.

9.7. Нисходящее и восходящее тестирование

Методики нисходящего и восходящего тестирования (рис. 9.8) 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.

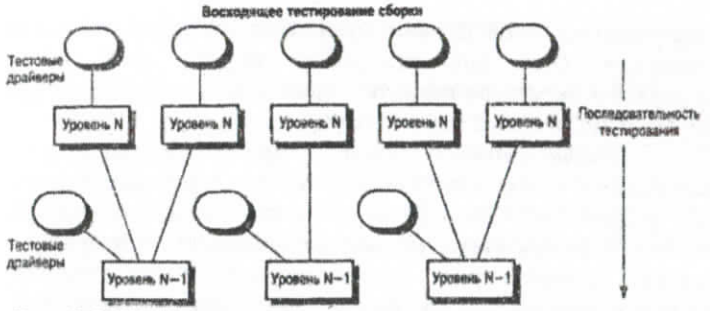


Рис. 9.8. Нисходящее и восходящее тестирование сборки

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 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.

9.8. 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. 6.9. Стрелки в верхней части схемы означают, что контрольные тесты применяются не к отдельным компонентам, а к подсистемам, полученным в результате комбинирования этих компонентов.

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.

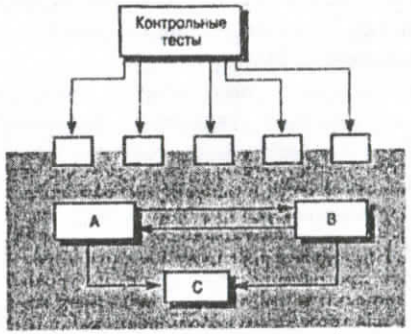


Рис. 9.9. Тестирование интерфейсов

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.

9.9. Тестирование с нагрузкой

После полной интеграции системы можно оценить такие интеграционные свойства системы, как производительность и надежность. Чтобы убедиться, что система может работать с заданной нагрузкой, разрабатываются тесты для измерения производительности. Обычно планируются серии тестов с постоянным увеличением нагрузки, пока производительность системы не начнет снижаться.

Некоторые классы систем проектируются с учетом работы под определенной нагрузкой. Например, система обработки транзакций проектируется так, чтобы обрабатывать 100 транзакций в секунду; сетевая операционная система – чтобы обрабатывать информацию от 200 отдельных терминалов. При тестировании с нагрузкой выполнение тестов начинается с максимальной нагрузки, указанной в проекте системы, и продолжается до тех пор, пока не произойдет сбой в работе системы. Данный тип тестирования выполняет две функции.

1. Тестируется поведение системы во время сбоя. В процессе эксплуатации могут возникать ситуации, при которых нагрузка в системе превышает максимально допустимую. В таких ситуациях

очень важно, чтобы сбой в системе не приводил к нарушению целостности данных или к потере сервисных возможностей.

2. Чтобы выявить дефекты, которые не проявляются в обычных режимах работы, система подвергается тестированию с нагрузкой. Хотя подобные дефекты не приводят к ошибкам при обычном использовании системы, на практике могут возникнуть необычные комбинации стандартных условий; именно они воспроизводятся во время тестирования с нагрузкой.

Тестирование с нагрузкой чаще всего применяется в распределенных системах. В таких системах при большой нагрузке сеть порой "забивается" данными, которыми обмениваются разные процессы. Постепенно процессы все больше замедляются, поскольку они ожидают данные запросов от других процессов.

9.10. Тестирование объектно-ориентированных систем

Мы рассмотрели два основных подхода к тестированию программного обеспечения – компонентное тестирование, при котором компоненты системы тестируются независимо друг от друга, и тестирование сборки, когда компоненты интегрированы в подсистемы и тестируется конечная система. Эти подходы в равной мере применимы и к объектно-ориентированным системам. Однако системы, разработанные по функциональной модели, и объектно-ориентированные системы имеют существенные отличия.

1. Объекты, как отдельные программные компоненты, представляют собой нечто большее, чем отдельные подпрограммы или функции.

2. Объекты, интегрированные в подсистемы, обычно слабо связаны между собой и поэтому сложно определить "самый верхний уровень" системы.

3. При анализе повторно используемых объектов их исходный код может быть недоступным для испытателей.

Эти отличия означают, что при проверке объектов можно применять тестирование методом белого ящика, основанное на анализе кода, а при тестировании сборки следует использовать другие подходы. Применительно к объектно-ориентированным системам можно определить четыре уровня тестирования.

1. Тестирование отдельных методов, ассоциированных с объектами. Обычно методы представляют собой функции или процедуры. Поэтому здесь можно использовать тестирование методами черного и белого ящиков, которые рассматривались ранее.
2. Тестирование отдельных классов объектов. Принцип тестирования методом черного ящика остается без изменений, однако, понятие "класса эквивалентности" необходимо расширить.
3. Тестирование кластеров объектов. Нисходящая и восходящая сборки оказываются не пригодными для создания групп связанных объектов. Поэтому здесь следует применять другие методы тестирования, например основанные на сценариях.
4. Тестирование системы. Верификация и аттестация объектно-ориентированной системы выполняется точно так же, как и для любых других типов систем.

9.11. Тестирование классов объектов

Подход к тестовому покрытию систем требует, чтобы все операторы в программе выполнялись хотя бы один раз, а также чтобы выполнялись все ветви программы. При тестировании объектов полное тестовое покрытие включает:

- раздельное тестирование всех методов, ассоциированных с объектом;
- проверку всех атрибутов, ассоциированных с объектом;
- проверку всех возможных состояний объекта.

В качестве примера возьмем метеорологическую станцию. Интерфейс объекта, соответствующего метеостанции, показан на рис. 6.10. У этого объекта только один атрибут, который является его идентификатором. Это константа, значение которой необходимо задать после инсталляции объекта. Таким образом, нужен тест, который проверял бы задание идентификатора. Необходимо также определить контрольные тесты для методов отчет, калибровать, тестировать, запуск и завершение. В идеале следует протестировать все эти методы независимо, но в некоторых случаях нужна последовательность тестов. Например, чтобы протестировать метод завершение, необходимо сначала выполнить метод запуск.

При тестировании состояний объекта Метеостанция используется модель состояний, показанная на рис. 9.11. С помощью этой модели можно определить последовательность состояний, которые нужно протестировать. В принципе следует проверить каждый возможный переход из одного состояния в другое, хотя на практике такой подход оказывается слишком дорогостоящим. В нашем примере необходимо протестировать такие последовательности состояний:

Останов → Ожидание → Останов

Ожидание → Калибровка →Тестирование → Передача → Ожидание

Ожидание → Калибровка → Ожидание → Обобщение → Передача → Ожидание

Использование наследования усложняет разработку тестов для классов объектов. Если класс предоставляет методы, унаследованные от подклассов, то необходимо протестировать все подклассы со всеми унаследованными методами. Понятие классов эквивалентности можно применить также и к классам объектов. Здесь тестовые данные из одного класса эквивалентности тестируют одни и те же свойства объектов.

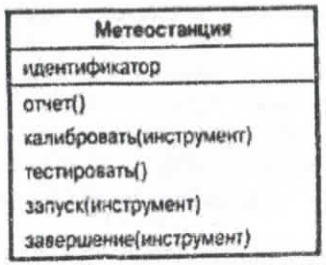


Рис. 9.11. Интерфейс объекта метеорологической станции

9.12. Интеграция объектов

При разработке объектно-ориентированных систем различия между уровнями интеграции менее заметны, поскольку методы и данные компонуются интегрируются в виде объектов и классов объектов. Тестирование классов объектов соответствует тестированию отдельных элементов. В объектно-ориентированных

системах нет непосредственного эквивалента тестированию модулей. Однако считается, что группы классов, которые совместно предоставляют набор сервисов, следует тестировать вместе. Такой вид тестирования называется тестированием кластеров.

Для объектно-ориентированных систем не подходит ни восходящая, ни нисходящая интеграция системы, поскольку здесь нет строгой иерархии объектов. Поэтому создание кластеров основывается на выделении методов и сервисов, реализуемых посредством этих кластеров. При тестировании сборки объектно-ориентированных систем используется три подхода.

1. Тестирование сценариев и вариантов использования. Варианты использования или сценарии описывают какой-либо один режим работы системы. Тестирование может базироваться на описании этих сценариев и кластеров объектов, реализующих данный вариант использования.

2. Тестирование потоков. Этот подход основывается на проверке системных откликов на ввод данных или группу входных событий. Объектно-ориентированные системы, как правило, событийно-управляемые, поэтому для них особенно подходит данный вид тестирования. При использовании этого подхода необходимо знать, как в системе проходит обработка потоков событий.

3. Тестирование взаимодействий между объектами. Это метод тестирования групп взаимодействующих объектов, предложенный в работе. Этот промежуточный уровень тестирования сборки системы основан на определении путей "метод-сообщение", отслеживающих последовательности взаимодействий между объектами.

Тестирование сценариев часто оказывается более эффективным, чем другие методы тестирования. Сам процесс тестирования можно спланировать так, чтобы в первую очередь проверялись наиболее вероятные сценарии и только затем исключительные сценарии.

Поэтому тестирование сценариев удовлетворяет основному принципу, согласно которому при тестировании больше внимания необходимо уделять наиболее часто используемым частям системы.

Чтобы проиллюстрировать тестирование сценариев, вновь рассмотрим систему метеостанции. Сценарии можно определить, исходя из разработанных вариантов использования, однако их может быть недостаточно для тестирования.

Поэтому при планировании тестирования можно использовать диаграммы взаимодействия, отображающие реализацию вариантов использования посредством системных объектов.

Рассмотрим рис. 9.12 на котором изображена последовательность операций, выполняемых при сборе метеоданных. Эту диаграмму можно использовать для определения тестируемых операций и для разработки тестовых сценариев.

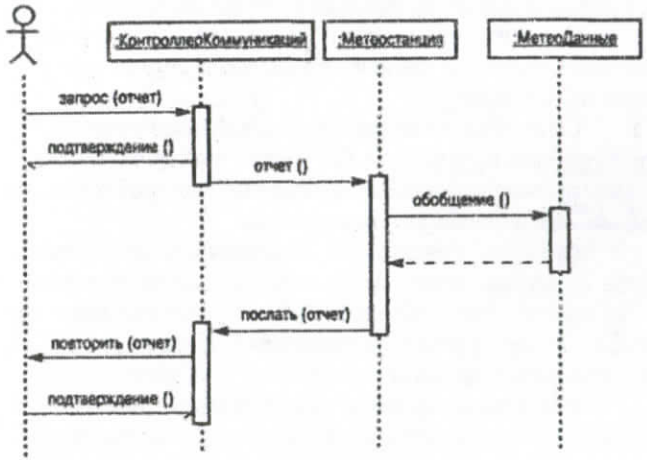


Рис. 9.12. Диаграмма последовательности сбора метеоданных

На рис. 9.12 видно, что при получении запроса на отчет о метеоданных в системе будет выполняться следующий поток методов:

КонтроллерКоммуникаций:запрос → Метеостанция:отчет → МетеоДанные:обобщение

После выбора сценариев для тестирования системы важно убедиться, что все методы каждого класса будут выполняться хотя бы один раз. Для этого можно составить технологическую карту проверок классов объектов и методов и при выборе сценария отмечать выполняемый метод.

Конечно, все комбинации методов выполнить невозможно, но по крайней мере можно удостовериться, что все методы протестированы как часть какой-либо последовательности выполняемых методов.

9.13. 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. 9.13 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.

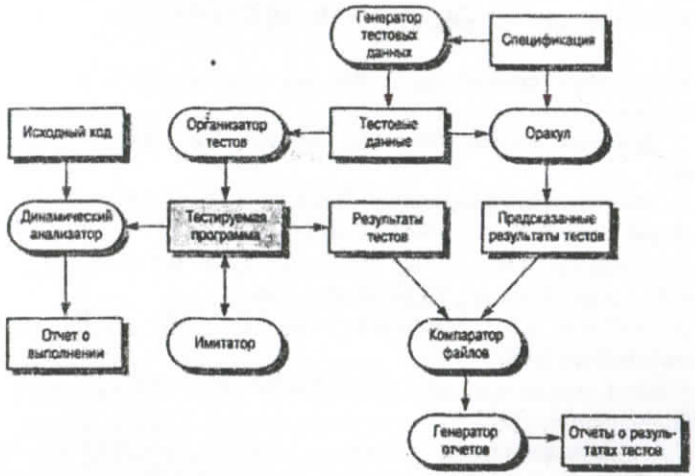


Рис. 9.13. Инструментальные средства тестирования

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.

ОСНОВНАЯ ЛИТЕРАТУРА

1. Software engineering - 9th ed. Ian Sommerville, 2011,Addison-Wesley
2. Software Project Survival Guide.S. McConnell, 1998, Microsoft Press.
3. Peopleware: Productive Projects and Teams, 2nd edition. T. DeMarco and T. Lister, 1999, Dorset House.
4. Waltzing with Bears: Managing Risk on Software Projects. T. DeMarco and T. Lister, 2003, Dorset House.
5. Software Cost Estimation with COCOMO II. B. Boehm et al., Prentice Hall, 2000.
6. Ten unmyths of project estimation. P. Armour, Comm. ACM, 45 (11), November 2002.
7. Agile Estimating and Planning. M. Cohn, Prentice Hall, 2005.
8. Achievements and Challenges in Cocomo-based Software Resource Estimation. B. W. Boehm and R. Valeridi, IEEE Software, 25 (5), September/October 2008.
9. Metrics and Models for Software Quality Engineering, 2nd edition. S. H. Kan, Addison-Wesley, 2003.
10. Software Quality Assurance: From Theory to Implementation. D. Galin, Addison-Wesley, 2004.
11. A Practical Approach for Quality-Driven Inspections. C. Denger, F. Shull, IEEE Software, 24 (2), March–April 2007.
12. Misleading Metrics and Unsound Analyses. B. Kitchenham, R. Jeffrey and C. Connaughton, IEEE Software, 24 (2), March–April 2007.
13. The Case for Quantitative Project Management. B. Curtis et al., IEEE Software, 25 (3), May–June 2008.
14. Configuration Management Principles and Practice. Anne Mette Jonassen Hass, Addison-Wesley, 2002.
15. Software Configuration Management Patterns: Effective Teamwork, Practical Integration. S. P. Berczuk with B. Appleton, Addison-Wesley, 2003.
16. High-level Best Practices in Software Configuration Management. L. Wingerd and C. Seiwald, 2006.
17. Agile Configuration Management for Large Organizations. P. Schuh, 2007.
18. Can you trust software capability evaluations? E. O'Connell and H. Saiedian, IEEE Computer, 33 (2), February 2000.

19. Software Process Improvement: Results and Experience from the Field. Conradi, R., Dybå, T., Sjøberg, D., Ulsund, T. (eds.), Springer, 2006.
20. CMMI: Guidelines for Process Integration and Product Improvement, 2nd edition. M. B. Chrissis, M. Konrad, S. Shrum, Addison-Wesley, 2007.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

21. Мирзияев Ш.М. Мы все вместе построим свободное, демократическое и процветающее государство Узбекистан. – Ташкент: Узбекистон, 2016. 56-с.
22. Мирзиёев Ш.М. Танқидий таҳлил, қатъий тартиб-интизом ва шахсий жавобгарлик – ҳар бир раҳбар фаолиятининг кундалик қоидаси бўлиши керак. Мамлакатимизни 2016 йилда ижтимоий-иқтисодий ривожлантиришнинг асосий якунлари ва 2017 йилга мўлжалланган иқтисодий дастурнинг энг муҳим устувор йўналишларига бағишланган Вазирлар Маҳкамасининг кенгайтирилган мажлисидаги маъруза. 2017 йил 14 январь – Тошкент: Ўзбекистон, 2017. 104-б.
23. Мирзиёев Ш.М. Қонун устуворлиги ва инсон манфаатларини таъминлаш - юрт тараққиёти ва халқ фаровонлигининг гарови. Ўзбекистон Республикаси Конституцияси қабул қилинганининг 24 йиллигига бағишланган тантанали маросимдаги маъруза. 2016 йил 7 декабрь. – Тошкент: Ўзбекистон, 2017. 48-б.
24. Мирзиёев Ш.М. Буюк келажагимизни мард ва олижаноб халқимиз билан бирга қурамиз. Мазкур китобдан Ўзбекистон Республикаси Президенти Шавкат Мирзиёевнинг 2016 йил 1 ноябрдан 24 ноябрга қадар Қорақалпоғистон Республикаси, вилоятлар ва Тошкент шаҳри сайловчилари вакиллари билан ўтказилган сайловолди учрашувларида сўзлаган нутқлари ўрин олган. –Тошкент: Ўзбекистон, 2017. 488-б.

Интернет-материалы:

25. http://www.SoftwareEngineering-9.com
26. http://git-scm.com
27. https:// github.com
28. http://dx.doi.org

Оглавление

А.Х.Нишанов, О.Рузибаев,
М.Ю.Дощанова, А.М.Матьякубова.

ПРОГРАММНЫЙ ИНЖИНИРИНГ

(Учебное пособие)

Ташкент – «Aloqachi» – 2019

Редактор: М. Миркамилов
Тех. редактор: А. Тагаев
Художник: Б. Эсанов
Корректор: Ф. Тагаева
Компьютерная верстка: Ш. Тухтамурадов

Изд. лиц. Ii №176. 11.06.2010.
Разрешено в печать: 3.06.2019.
Формат 60x84 1/16. Гарнитура «Times New Roman».
Усл. п.л. 11,0. Изд.п.л. 10,5. Тираж 100. Заказ № 116.