# Глава 7

В этой главе мы обсудим проблемы, возникающие в процессе разработки крупных и сложных систем программного обеспечения. Эту область компьютерных наук называют *технологией разработки программного обеспечения*, поскольку создание ПО, безусловно, следует понимать как процесс технический. Цель исследований, проводимых в этой области компьютерных наук, — найти принципы, определяющие ход процесса разработки программного обеспечения и способные обеспечить создание эффективных и надежных программных продуктов.

# Технология разработки программного обеспечения

Технология программного обеспечения — это отрасль компьютерных наук, которая ищет основные принципы, определяющие успешную разработку больших и сложных программных систем. Проблемы, с которыми приходится сталкиваться при разработке таких систем, — это нечто большее, чем просто расширенные версии задач, возникающих при написании небольших программ. В частности, разработка подобных систем, как правило, требует усилий многих людей на протяжении длительного времени, причем за этот период могут меняться как требования к создаваемой системе, так и состав персонала, занимающегося ее разработкой. Поэтому предмет технологии разработки программного обеспечения включает и такие аспекты, как управление проектом и персоналом, которые обычно ассоциируются с менеджментом, а не с областью компьютерных наук. Однако в этой главе мы сконцентрируем наше внимание только на вопросах, относящихся к компьютерным наукам.

## 7.1. Предмет технологии разработки программного обеспечения

Чтобы иметь представление о проблемах, возникающих при проектировании программного обеспечения, рассмотрим любой достаточно большой и сложный объект (автомобиль, многоэтажное офисное здание или даже кафедральный собор), а затем представим себе, что в наши обязанности входит как разработка его проекта, так и управление процессом его сооружения. Как оценить время, деньги и другие ресурсы, которые потребуются для выполнения проекта? Как разделить проект на управляемые части? Как убедиться, что созданные части совместимы? Как обеспечить взаимодействие сотрудников, работающих над разными частями проекта? Как оценить скорость выполнения работы? Как справиться с обилием всевозможных деталей (выбором дверных ручек, проектированием скульптурных украшений, поисками голубого стекла для витражей, расчетом прочности колонн, планированием работ по монтажу системы отопления)? На подобные вопросы приходится отвечать и в процессе разработки крупной системы программного обеспечения.

Поскольку проектирование является хорошо разработанной областью, можно полагать, что уже существует множество разработанных методов проектирования, предоставляющих ответы на такие вопросы. Частично эти рассуждения верны, однако в них не учитываются фундаментальные различия в свойствах программного обеспечения и объектов проектирования из других областей науки и техники. Эти различия существенно усложняют реализацию проектов по разработке программного обеспечения, что ведет к перерасходу средств, задержкам в поставках готовых продуктов и неудовлетворенности клиентов.

В свою очередь, выявление этих различий и стало первым шагом в развитии дисциплины технологии разработки программного обеспечения.

Одно из этих различий связано с возможностью создания новой системы из предварительно изготовленных компонентов общего назначения. Уже многие годы при создании сложных объектов в традиционных областях проектирования значительные преимущества достигаются просто за счет использования в качестве строительных блоков некоторых готовых компонентов. Конструктору автомобиля не нужно конструировать для него новый мотор или трансмиссию, — вместо этого он может воспользоваться уже созданными версиями этих компонентов. Однако в области разработки программного обеспечения в этом отношении имеется значительное отставание. В прежние времена ранее разработанные компоненты, как правило, не являлись универсальными, т.е. их внутренняя конструкция зависела от характерных особенностей конкретного приложения. А это означало, что возможности использования общих компонентов были весьма ограничены, — попытка повторно использовать подобные компоненты неизбежно требовала их перепроектирования. В результате исторически сложилось так, что достаточно сложные системы программного обеспечения чаще всего создавались практически “с нуля”. Как мы увидим далее в этой главе, в настоящее время в этом отношении уже достигнут значительный прогресс, хотя многое все еще предстоит сделать.

Другое различие между разработкой программного обеспечения и традиционным инженерным проектированием связано с отсутствием количественных методов, называемых **метриками**, которые можно было бы использовать для измерения характеристик программного обеспечения. Например, чтобы спрогнозировать стоимость разработки программной системы, хотелось бы оценить сложность предлагаемого продукта, но методы измерения “сложности” программного обеспечения уклончивы. Аналогичным образом оценка качества программного продукта является сложной задачей. В случае механических устройств важной мерой качества является среднее время между отказами, которое, по существу, является показателем того, насколько хорошо устройство выдерживает износ. В противоположность этому программное обеспечение не изнашивается, поэтому данный метод измерения качества нельзя применить в разработке программного обеспечения.

Трудности, связанные с количественным измерением свойств программного обеспечения, являются одной из причин того, что в технологии разработки программного обеспечения предпринимаются настойчивые попытки найти строгую основу в том же смысле, какой имеет место в механике и электротехнике. В то время как эти последние области инженерии стоят на фундаменте устоявшейся науки физики, в технологии разработки программного обеспечения поиск своих корней все еще продолжается.

Поэтому в настоящее время исследования в области технологии разработки программного обеспечения разворачиваются на двух уровнях. Одни исследователи, которых иногда называют *практиками*, работают над развитием методов разработки, предназначенных для немедленного применения, тогда как другие исследователи, которых называют *теоретиками*, ведут поиск основополагающих принципов и теорий, на основании которых впоследствии можно будет разработать более надежные технологические методы. Основанные на субъективных представлениях, многие разработанные (и применяемые) в прошлом практиками методы были заменены другими подходами, которые со временем также могут устареть. Между тем успехи теоретиков остаются весьма относительными.

Прогресс в исследованиях как практиков, так и теоретиков имеет очень большое значение. Жизнь нашего общества уже невозможна без использования компьютерных систем и связанного с ними программного обеспечения. Экономика, здравоохранение, государственные учреждения, законодательство, транспорт и оборона любого современного государства зависят от больших систем программного обеспечения. Тем не менее надежность этих систем по-прежнему остается большой проблемой. Ошибки в программном обеспечении уже приводили к таким катастрофическим (или почти катастрофическим) последствиям, как принятие растущей луны за ядерную атаку, потеря 5 миллионов долларов банком Нью-Йорка только за один день, утрата проб космического пространства, собранных космической станцией, радиационное облучение людей, вызвавшее их смерть или паралич, и даже одновременное нарушение работы линий телефонной связи в обширных географических регионах.

Тем не менее не следует считать, что ситуация совсем уж мрачная. В действительности уже достигнут большой прогресс в преодолении таких проблем, как отсутствие готовых компонентов и метрик. Более того, применение компьютерных технологий в процессе разработки программного обеспечения привело к появлению того, что сейчас называют **средствами автоматизации разработки программ** (*CASE* — Computer-Aided Software Engineering). Этот подход позволяет упростить и рационализировать весь процесс разработки программного обеспечения. Появление концепции CASE привело к разработке разнообразных компьютеризированных систем, известных как **CASE-инструменты**, включающих *системы планирования проектов* (предназначены для оказания помощи в проведении оценки затрат, составлении графика проекта и распределении персонала), *системы управления проектами* (используются для организации мониторинга хода выполнения проекта), *средства документирования* (предназначены для оказания помощи в написании и организации документации), *системы прототипирования и моделирования* (используются в разработке прототипов), *системы проектирования интерфейсов* (предназначены

для ускорения разработки графических интерфейсов) и *системы программирования* (используются для организации и упрощения процедур написания и отладки программ). Одни из этих инструментов представляют собой нечто чуть большее, чем обычные текстовые процессоры, программное обеспечение для работы с электронными таблицами или системы связи по электронной почте, которые изначально разрабатывались для общего использования, а затем были адаптированы программистами для своих нужд. Другие представляют собой довольно сложные пакеты, разработанные, в первую очередь, для среды разработки программного обеспечения. И действительно, системы, известные как **интегрированная среда разработки** (*IDE* — Integrated Development Environments), объединяют в себе инструменты для разработки программного обеспечения (редакторы, компиляторы, средства отладки и т.д.) в единый интегрированный пакет. Ярким примером таких систем являются системы для разработки приложений для смартфонов. Они не только включают все инструменты программирования, необходимые для написания и отладки программного обеспечения смартфона, но также предоставляют программисту программы-эмуляторы, которые позволяют ему увидеть на дисплее обычного персонального компьютера то, как разрабатываемое им программное обеспечение будет работать и выглядеть непосредственно на экране смартфона.

В дополнение к прилагаемым исследователями усилиям профессиональные организации и организации, отвечающие за стандартизацию, включая ISO, Ассоциацию по вычислительной технике (*ACM* — Association for Computing Machinery) и Институт инженеров электротехники и электроники (*IEEE* — Institute of Electrical and Electronics Engineers), также вступили в борьбу за улучшение состояния дел в отрасли разработки программного обеспечения. Их усилия варьируются от принятия кодексов профессионального поведения и этики, направленных на повышение профессионализма разработчиков программного обеспечения и противодействие небрежному отношению к обязанностям отдельных исполнителей, до установления стандартов по измерению качества организаций, занимающихся разработкой программного обеспечения, а также предоставления рекомендаций, призванных помочь этим организациям улучшить свои позиции.

Одна из важнейших проблем, охватывающих все аспекты разработки больших программных систем, заключается в том, что объем необходимых усилий обязательно предполагает участие в работе многих людей. Хотя небольшие системы вполне могут быть спроектированы одним целеустремленным человеком, сама природа процесса разработки программного обеспечения в крупных проектах требует, чтобы его многочисленные участники имели возможность общаться и эффективно сотрудничать. Поэтапная совместная работа над большим программным проектом предполагает совсем иной набор навыков в

сравнении с написанием кода в индивидуальном порядке, но одновременно позволяет снизить сложность задач, стоящих перед каждым отдельным исполнителем. В действительности именно потому, что большие программные системы являются настолько сложными, возникает необходимость разбить всю систему на более мелкие компоненты — такие, чтобы исполнители-люди смогли полностью понять возложенную на них часть общей работы.

## Ассоциация по вычислительной технике

Основанная в 1947 году, Ассоциация по вычислительной технике (*ACM* — Association for Computing machinery) является международной научной и образовательной организацией, задача которой — распространение навыков, теорий и приложений из области информационных технологий. Штаб-квартира этой организации, расположенная в Нью-Йорке, руководит деятельностью множества групп по различным направлениям (*SIG* — Special Interest Group), занимающихся такими проблемами, как архитектура компьютеров, искусственный интеллект, применение компьютеров в биомедицинских исследованиях, компьютеры и общество, обучение в области компьютерных наук, компьютерная графика, гипертекст/гипермедиа, операционные системы, языки программирования, имитация и моделирование, а также разработка программного обеспечения.

Веб-сайт этой ассоциации находится по адресу `http://www.acm.org`. Разработанный в этой ассоциации Кодекс этики и профессионального поведения можно найти по адресу `http://ethics.acm.org/code-of-ethics`.

### *Основные положения для запоминания*

- **Сотрудничество позволяет сократить размер и сложность задачи, возложенной на отдельного программиста.**
- **Сотрудничество в итеративной разработке программного обеспечения требует иного набора умений в сравнении с созданием программы в одиночку.**
- **При разработке программного обеспечения эффективное общение между отдельными участниками является необходимым условием их успешного сотрудничества.**

В остальной части этой главы будут обсуждаться как определенные фундаментальные принципы технологии разработки программного обеспечения (такие, как жизненный цикл программного обеспечения и модульность), так и некоторые из направлений, в которых сейчас ведутся исследования в этой области (такие, как выявление и применение шаблонов проектирования и

появление повторно используемых программных компонентов). Также будет проанализирован тот эффект, который парадигма объектно-ориентированного программирования произвела на эту область компьютерных наук.

### 7.1. Вопросы и упражнения

1. Почему количество строк в программе нельзя считать надежной мерой ее сложности?
2. Предложите метрику для оценки качества программного обеспечения. Какие, по вашему мнению, слабые места она имеет?
3. С помощью какого метода можно определить, сколько ошибок содержится в определенной части программного обеспечения?
4. Приведите два примера направлений в области технологии разработки программного обеспечения, в которых определенный успех уже был достигнут либо продвижение наблюдается непосредственно в данный момент.

## 7.2. Жизненный цикл программного обеспечения

Важнейшим понятием в технологии разработки программного обеспечения является жизненный цикл программы.

### Жизненный цикл в целом

Жизненный цикл программы представлен на рис. 7.1. Здесь отражен тот факт, что, будучи однажды созданной, программа входит в цикл, включающий ее использование и модификацию (другой термин — *сопровождение*); продолжительность этого цикла распространяется на все время жизни программы. Такая картина характерна и для многих промышленных изделий. Отличие заключается лишь в том, что для таких изделий фазу модификации точнее было бы назвать ремонтом или техническим обслуживанием, тогда как в случае программного обеспечения фаза модификации обычно предполагает внесение исправлений или обновлений. В действительности программное обеспечение переходит в фазу модификации потому, что в нем обнаруживаются ошибки, возникают те или иные изменения в области его применения, что требует внесения соответствующих изменений в программное обеспечение, или из-за того, что предыдущая модификация в одной части программного обеспечения, вызвала появление проблем в других его частях.

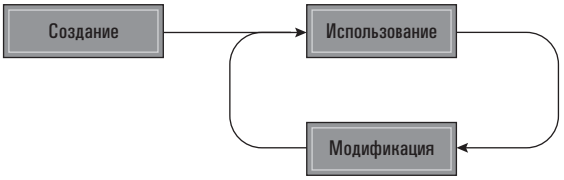



**Рис. 7.1.** Жизненный цикл программы

Независимо от того, почему программа модифицируется, необходимо, чтобы некто (как правило, не автор ее исходной версии) изучил и понял программу и ее документацию, а если не всю программу, то хотя бы ту часть, которая относится к делу. В противном случае любая модификация может вызвать намного больше новых проблем, чем позволит решить уже имеющихся. Достижение необходимой степени понимания является сложной задачей, даже если программа правильно разработана и документирована. Фактически именно на этой стадии отдельные фрагменты программного обеспечения просто отбрасывают, исходя из утверждений (часто вполне справедливых), что проще разработать новую систему “с нуля”, чем успешно модифицировать уже существующий пакет.

Опыт показывает, что незначительные дополнительные усилия, затраченные при разработке программы, впоследствии могут существенно изменить ситуацию, когда потребуется внести в нее изменения. Например, при обсуждении операторов описания данных в главе 6 было показано, как вместо безличного конкретного числового значения в программе может использоваться константа с символическим именем, что значительно упрощает последующие изменения. Как следствие б*о*льшая часть исследований в области технологии разработки программного обеспечения концентрирует свое внимание именно на фазе разработки в жизненном цикле программы, имея своей целью достижение преимуществ за счет дополнительных усилий на данной стадии, обеспечивающих определенные выгоды впоследствии.

## Традиционные этапы разработки программ

Фаза разработки в жизненном цикле программы традиционно включает следующие этапы: анализ требований, проектирование, реализацию и тестирование (рис. 7.2).

***Основные положения для запоминания***

- Решая поставленную задачу, программист разрабатывает, реализует, тестирует, отлаживает и сопровождает (модифицирует) программу.

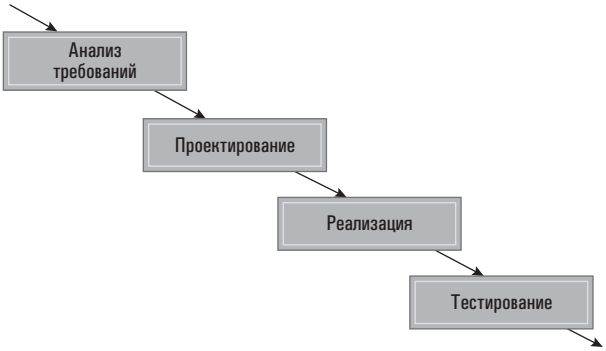



**Рис. 7.2.** Традиционное представление фазы разработки в жизненном цикле программного обеспечения

## Анализ требований

Жизненный цикл программного обеспечения начинается с анализа требований, основная задача которого состоит в определении того, какие услуги будет предоставлять данная система, каковы будут условия предоставления этих услуг (временны*е* ограничения, безопасность и т.д.) и каким образом внешний мир будет взаимодействовать с данной системой.

Анализ требований, выдвигаемых к создаваемой системе, предполагает значительный вклад со стороны **заинтересованных лиц** (это ее будущие пользователи, а также тех, кто имеет иные интересы, например юридические или финансовые). Фактически в тех случаях, когда конечным пользователем является организация, такая как компания или государственное агентство, которая намеревается нанять стороннего разработчика ПО для фактического выполнения требуемого программного проекта, анализ требований может начаться с технико-экономического обоснования, проводимого исключительно заказчиком. В других случаях разработчик ПО может заниматься производством готового **коммерческого программного обеспечения** (*COTS* — Commercial Off-The-Shelf) для массового рынка, которое затем будет продаваться в розничных магазинах или загружаться через Интернет. В такой ситуации пользователь является более расплывчатой фигурой и анализ требований может начинаться с изучения рынка разработчиком программного обеспечения.

В любом случае процесс анализа требований включает сбор и анализ нужд потенциальных пользователей, обсуждение с заинтересованными сторонами возможных компромиссов в отношении их пожеланий и устанавливаемых ограничений, возможного уровня затрат, а также осуществимости поставленных задач. В конечном итоге вырабатывается набор требований, определяющих функции и услуги, которые должна обеспечивать готовая система программного

обеспечения. Эти требования записываются в документе, который называют **спецификацией требований к программному обеспечению**. В некотором смысле этот документ представляет собой письменное соглашение между всеми заинтересованными сторонами, назначение которого — направлять процесс разработки программного обеспечения и предоставлять средства для разрешения любых споров, которые могут возникнуть по ходу процесса разработки. Важность спецификации требований к программному обеспечению демонстрируется тем фактом, что профессиональные организации, такие как IEEE, а также крупные заказчики программного обеспечения, такие как Министерство обороны США, приняли соответствующие стандарты по их составлению.

С точки зрения разработчика программного обеспечения, спецификация требований к ПО должна четко определять ту цель, которую необходимо достичь посредством разрабатываемого программного обеспечения. Однако очень часто этот документ оказывается неспособным обеспечить требуемую стабильность. В действительности большинство практиков в области разработки ПО утверждают, что недостаточный уровень общения и изменение требований являются основными причинами перерасхода средств и задержек с поставкой готового продукта во всей отрасли разработки программного обеспечения. Очень немногие клиенты будут настаивать на серьезных изменениях в плане этажа здания после того, как его фундамент уже будет построен, однако существует множество организаций, которые расширили или иным образом изменили желаемые характеристики разрабатываемой программной системы много позже того, как разработка программного обеспечения уже началась. Так бывает, когда компания-заказчик приходит к заключению, что система, которая изначально разрабатывалась только для ее дочерней компании, теперь будет предназначаться для всей корпорации, либо когда новые технологические достижения позволяют достичь намного большего, чем это было возможно в тот момент, когда проводился первоначальный анализ требований. Как бы там ни было, инженеры-программисты доказали, что прямое и частое общение с заинтересованными сторонами проекта является обязательным.

***Основные положения для запоминания***

- Консультации и тесное общение с пользователями программы являются важным аспектом разработки программного обеспечения, позволяющим решать любые возникающие проблемы.
- Разработка программного обеспечения включает выявление возникающих у программистов и конечных пользователей вопросов, оказывающих влияние на решение проблем.

## Проектирование

Результат этапа анализа требований — это описание создаваемого программного продукта, тогда как этап проектирования предусматривает разработку плана построения создаваемой системы. В некотором смысле анализ требований заключается в определении проблемы, которую предстоит решить, а проектирование — в отыскании способа решения этой проблемы. С точки зрения непрофессионала, анализ требований часто приравнивается к решению, *что* программная система должна делать, тогда как проектирование приравнивается к нахождению решения, *как* система будет это делать. Хотя такое определение можно считать достаточно наглядным, многие разработчики программного обеспечения утверждают, что оно некорректно, поскольку на самом деле многое из категории *как*, рассматривается уже при анализе требований, тогда как многое из категории *что* определяется лишь в процессе проектирования.

Именно на стадии проектирования устанавливается внутренняя структура создаваемой системы программного обеспечения. А значит, результатом этапа проектирования должно быть подробное описание структуры системы программного обеспечения, которое затем будет преобразовано в программы.

Если проект предполагает возведение офисного здания, а не создание системы программного обеспечения, этап проектирования будет включать разработку подробных структурных планов здания, отвечающих всем установленным требованиям. Например, такие планы должны будут включать набор чертежей, описывающих предлагаемое здание на различных уровнях детализации. Именно на основании этих документов и будет построено реальное здание. Методы разработки подобных планов развивались в течение многих лет и включают стандартизированные системы обозначений и многочисленные методологии моделирования и построения чертежей.

Аналогичным образом создание диаграмм и моделирование играют важную роль и в разработке программного обеспечения. Однако методологии и системы обозначений, используемые разработчиками программного обеспечения, не являются настолько устоявшимися, как в области архитектуры и строительства. При сравнении с четко определенной дисциплиной архитектуры практика разработки программного обеспечения выглядит очень динамичной, поскольку исследователи и сейчас всеми силами продолжают попытки найти лучшие подходы к процессу разработки программного обеспечения. Подробнее об этих изменениях в подходах к проектированию ПО речь пойдет в разделе 7.3, а в разделе 7.5 мы обсудим некоторые из существующих систем обозначений и связанные с ними методологии построения диаграмм и моделирования.

### Ассоциация IEEE

Международная некоммерческая ассоциация IEEE (Institute of Electrical and Electronics Engineers — Институт инженеров по электротехнике и радиоэлектронике) была создана в 1963 году в результате слияния американских обществ IAEE (основанного в 1884 году 25-ю инженерами-электротехниками, среди которых был и Томас Эдисон) и IRE (основанного в 1912 году). IEEE — всемирная организация инженеров в области электротехники, радиоэлектроники и радиоэлектронной промышленности со штаб-квартирой в г. Пискатауэй, штат Нью-Джерси, США. Она направляет деятельность 39-ти технических обществ, таких как общество инженеров аэрокосмических и электронных систем, общество специалистов в области лазеров и электрооптики, общество инженеров по робототехнике и автоматике, общество специалистов по самоходной технике, а также компьютерное общество. Наряду с другими видами деятельности, IEEE участвует в разработке стандартов. В частности, именно усилия этой организации позволили принять стандарты на форматы чисел с плавающей точкой (см. главу 1), которые используются в большинстве современных компьютеров.

Веб-сайт ассоциации IEEE находится по адресу `http://www.ieee.org`; вебсайт компьютерного общества — `http://www.computer.org`; а документ, содержащий кодекс этики организации IEEE, можно найти по адресу `http://www.ieee.org/about/whatis/code.html`.

## Реализация

Реализация включает собственно написание программ, создание файлов данных и разработку баз данных. Именно на этапе реализации становится заметным различие между задачами, стоящими перед **аналитиком программного обеспечения** (иначе называемого **системным аналитиком**) и **программистом**. Первый — это человек, принимающий участие во всем процессе разработки, возможно, с акцентом на этапы анализа требований и проектирования. Второй — это исполнитель, вовлеченный, прежде всего, в этап реализации. В самом узком понимании программисту поручается лишь написание программ, реализующих проект, созданный системным аналитиком. Указав на это различие, следует еще раз отметить, что в компьютерном сообществе не существует центрального органа, контролирующего использование терминологии. Многие из тех, кто носит звание системного аналитика, по сути являются программистами, и многие из тех, кого считают программистами (или, возможно, старшими программистами), на самом деле являются системными аналитиками в полном смысле этого слова. Подобное размывание терминологии вызывается тем фактом, что сегодня этапы процесса разработки программного обеспечения часто смешиваются, как будет показано ниже.

## Тестирование

В прошлом в традиционной схеме этапов разработки тестирование, по существу, приравнивалось к процессу отладки программ и подтверждению того, что конечный программный продукт совместим со спецификацией требований к программному обеспечению. Однако сегодня подобное видение этапа тестирования считается слишком узким. Программы нельзя считать единственными готовыми продуктами, которые должны быть подвергнуты тестированию в процессе разработки программного обеспечения. В действительности результат каждого промежуточного этапа во всем процессе разработки должен быть "проверен" на точность. Более того, как будет показано в разделе 7.6, теперь тестирование признано лишь одним из элементов в общей борьбе за обеспечение качества ПО, что является требованием, пронизывающим весь жизненный цикл программного обеспечения. Исходя из этого многие разработчики программного обеспечения теперь утверждают, что тестирование больше не следует рассматривать как отдельный этап в процессе разработки программного обеспечения. В действительности этот этап и его многочисленные проявления следует включить в другие этапы, создав тем самым трехступенчатый процесс разработки, составляющие элементы которого должны иметь такие имена, как "анализ *и подтверждение* требований", "проектирование и *проверка правильности* проекта" и "реализация и *тестирование*".

К сожалению, несмотря на использование современных методов обеспечения качества, большие системы программного обеспечения все еще могут содержать ошибки даже после продолжительного тестирования. Многие из этих ошибок могут оставаться незамеченными на протяжении всего жизненного цикла системы, в то время как другие могут стать причиной весьма опасных сбоев или отказов. Устранение таких ошибок является одной из важнейших задач технологии разработки программных систем. Тот факт, что количество обнаруживаемых ошибок все еще весьма значительно, говорит о том, что исследования в этой области необходимо продолжать.

### *7.2. Вопросы и упражнения*

1. Как в жизненном цикле программного обеспечения этап разработки влияет на этап модификации (или сопровождения)?
2. Кратко охарактеризуйте каждый из четырех этапов (анализ требований, проектирование, реализация и тестирование) фазы разработки в жизненном цикле программного обеспечения.
3. В чем состоит назначение спецификаций требований к программному обеспечению?

## 7.3. Методологии разработки программного обеспечения

Ранние подходы к проектированию программного обеспечения требовали строго последовательного выполнения этапов анализа, проектирования, реализации и тестирования. Предполагалось, что риск, связанный с использованием метода проб и ошибок при разработке больших систем программного обеспечения, слишком велик. В результате разработчики программного обеспечения настаивали на полном завершении разработки спецификаций требований к системе до начала ее проектирования. Точно так необходимо было полностью завершить этап проектирования системы до начала ее реализации. Подобная схема процесса разработки получила название **модель водопада** (по аналогии с движением потока падающей воды, поскольку процесс разработки должен был двигаться только в одном направлении).

Относительно недавно методы проектирования программного обеспечения были изменены таким образом, чтобы отразить противоречие между высокоструктурированной средой, диктуемой моделью водопада, и свободно развивающимся процессом проб и ошибок, который жизненно важен при творческом подходе к решению задач. Эти изменения выразились в появлении **пошаговой модели** разработки программного обеспечения. В соответствии с этой моделью требуемая система программного обеспечения создается поэтапно. Первый вариант является некоторой упрощенной версией требуемой системы с ограниченным набором функций. После того как эта версия будет протестирована и, возможно, оценена будущим пользователем, к ней последовательно добавляют и тестируют другие функции, и так до тех пор, пока система не приобретет законченный вид. Например, если разрабатываемая система является приложением для ведения личных дел пациентов больницы, то ее первое приближение может включать только функцию просмотра личных дел пациентов, составляющих лишь небольшую выборку из обширного общего архива. После того как эта версия окажется работоспособной, в ней поэтапно будут реализованы дополнительные функции, такие как добавление новых записей и обновление уже существующих.

***Основные положения для запоминания***

- Последовательное добавление новых проверенных программных сегментов к корректно работающей программе позволяет создавать большие правильно работающие программы.

Другой моделью, которая представляет собой отход от строгого соблюдения модели водопада, является **итерационная модель**, которая похожа на пошаговую модель и фактически иногда приравнивается к ней, хотя эти две функции различны. В то время как пошаговая модель несет в себе понятие *расширения* каждой предварительной версии продукта в более крупную версию, итерационная модель включает в себя концепцию *уточнения* каждой версии. В действительности пошаговая модель включает в себя базовый итеративный процесс, а итерационная модель может предусматривать постепенное добавление функций.

***Основные положения для запоминания***

- Итеративный, пошаговый процесс разработки ПО позволяет создать программу, корректно решающую поставленные задачи.

Важным примером итеративных методов является **RUP** (*Rational Unified Process* — рациональный унифицированный процесс), который был создан компанией Rational Software, в настоящее время являющейся подразделением корпорации IBM. По сути, RUP — это парадигма разработки программного обеспечения, переопределяющая этапы фазы разработки в жизненном цикле программного обеспечения и предоставляющая рекомендации по выполнению этих этапов. Эти рекомендации, а также CASE-инструменты для их поддержки, продаются корпорацией IBM. Сегодня методология RUP широко применяется в индустрии программного обеспечения. На практике ее высокая популярность привела к разработке непатентованной версии, получившей название "**унифицированный процесс**", которая доступна на некоммерческой основе.

В пошаговых и итеративных моделях иногда используют современную тенденцию в разработке программного обеспечения, предполагающую создание **прототипов**, т.е. когда создаются и оцениваются неполные версии разрабатываемой системы, называемые прототипами. В пошаговой модели прототип развивается в конечную версию системы, поэтому данный вариант метода создания прототипов именуется **эволюционным**. В других случаях прототипы могут отбрасываться, уступая место новым реализациям конечного проекта. Такой вариант метода создания прототипов называется методом с **отбрасыванием прототипов**. Примером использования этого варианта может служить **быстрое создание прототипов**, при котором на ранних стадиях разработки очень быстро создается серия упрощенных вариантов разрабатываемой системы. Такой прототип может состоять всего лишь из нескольких эскизов компоновки экрана, показывающих, как система будет взаимодействовать с пользователем

и каковы будут ее возможности. В данном случае задача состоит не в создании рабочей версии продукта, а в получении средства демонстрации, предназначенного для углубления взаимопонимания между участвующими в разработке сторонами. В частности, метод быстрого создания прототипов доказал свою эффективность в отношении систематизации требований к системе, выдвинутых на этапе анализа, а также в качестве средства для проведения презентаций возможностей системы ее потенциальным заказчикам.

Менее формальное воплощение пошаговых и итеративных идей, которое уже на протяжении многих лет используется компьютерными энтузиастами и любителями, широко известно как Разработка ПО: **разработка с открытым исходным кодом**. Это подход, на основании которого производится б*о*льшая часть современного бесплатного программного обеспечения. Возможно, наиболее ярким примером его использования является операционная система Linux, которая изначально разрабатывалась как система с открытым исходным кодом под руководством Линуса Торвальдса. Разработка программного пакета с открытым исходным кодом обычно происходит следующим образом: один автор пишет первоначальную версию программного обеспечения (как правило, для удовлетворения собственных потребностей) и размещает исходный код и сопровождающую его документацию в Интернете. Оттуда все это может быть загружено и использовано другими, причем совершенно бесплатно. Поскольку эти другие пользователи имеют полный исходный код и документацию, у них есть возможность изменять или улучшать это программное обеспечение в соответствии со своими потребностями либо исправлять обнаруженные ошибки. Они сообщают об этих изменениях первоначальному автору, который включает их в опубликованную версию программного обеспечения, делая эту расширенную версию доступной для дальнейших изменений. На практике подобный программный пакет может развиваться очень быстро, проходя через несколько последовательных расширений всего за одну неделю.

Возможно, наиболее заметный отход от модели водопада представлен набором методологий, известных как **гибкие методы** (agile methods), каждый из которых предполагает раннее и быстрое внедрение на пошаговой основе, реагирование на изменяющиеся требования и снижение акцента на строгий анализ и разработку исходных требований к системе. Одним из примеров гибкого метода является **экстремальное программирование** (*XP* — Extreme Programming). Следуя модели XP, программное обеспечение разрабатывается командой из менее чем дюжины человек, работающих в общем рабочем пространстве, в котором они свободно обмениваются идеями и помогают друг другу в процессе разработки проекта. Согласно этой модели программное обеспечение разрабатывается пошагово посредством повторяющихся ежедневных циклов неформального анализа требований, проектирования, реализации и тестирования. В результате новые

расширенные версии создаваемого программного пакета появляются на регулярной основе и каждая из них может быть оценена заинтересованными сторонами проекта и использована для указания направления дальнейшего расширения системы. Подводя итог, можно сказать, что гибкие методы характеризуются высокой изменчивостью, которая резко контрастирует с моделью водопада, с которой у нас ассоциируется образ группы менеджеров и программистов, работающих в отдельных офисах и строго выполняющих четко определенные фрагменты общей задачи разработки программного обеспечения.

Контраст, получаемый при сравнении модели водопада и метода XP, подчеркивает всю широту диапазона методологий, которые сейчас используются в процессах разработки программного обеспечения в надежде найти лучшие способы создания надежного программного обеспечения, причем максимально эффективным способом. Исследования в этой области — это непрекращающийся процесс. Определенный прогресс уже достигнут, но предстоит еще немало работы.

### *7.3. Вопросы и упражнения*

1. Кратко охарактеризуйте различия между традиционной моделью водопада в разработке программного обеспечения и более новыми парадигмами пошаговой и итерационной разработки.
2. Назовите три парадигмы разработки ПО, которые демонстрируют отход от жестко определенной последовательности этапов разработки в модели водопада.
3. В чем заключаются различия между традиционным эволюционным методом создания прототипов и разработкой программ с открытым исходным кодом?
4. Какие потенциальные проблемы, по вашему мнению, могут возникнуть с точки зрения прав собственности на программное обеспечение, разработанное с использованием методологии открытого исходного кода?

## 7.4. Модульность

Одним из ключевых положений в разделе 7.2 было утверждение о том, что для модификации программы необходимо разобраться в принципах ее работы или по крайней мере тех ее частей, которые относятся к делу. Зачастую этого нелегко достичь даже в небольших программах, а в крупных системах

программного обеспечения это было бы практически невозможно, если бы не принцип **модульности**, предполагающий разделение программного обеспечения на поддающиеся осмыслению элементы, обычно называемые **модулями**, каждый из которых сконструирован так, чтобы выполнять только определенную часть общей задачи.

## Реализация модулей

Модульности можно достичь многими способами. В главах 5 и 6 было показано, что в контексте императивной парадигмы модули можно реализовать в виде отдельных функций. В противоположность этому в контексте объектно-ориентированной парадигмы в качестве основных модульных составляющих использовались объекты. Эти различия очень важны, поскольку они определяют основную цель в самом начале процесса разработки программного обеспечения. Что будет такой целью: представить общую задачу в виде отдельных контролируемых процессов или же идентифицировать объекты в системе и понять, как они взаимодействуют?

Чтобы проиллюстрировать это, давайте рассмотрим, как процесс разработки простой модульной программы для симуляции игры в теннис будет происходить при выборе для этой программы императивной либо объектно-ориентированной парадигмы. В императивной парадигме работа начинается с рассмотрения действий, которые должны произойти. Поскольку каждая серия ударов по мячу инициируется игроком, подающим мяч, имеет смысл начать рассмотрение с функции `Serve()`(подача), представляющей процесс подачи. Ее задача — вычислить начальную скорость и направление полета мяча, исходя из характеристик игрока, возможно, с добавлением небольшой вероятности. Далее нужно будет определить путь мяча. (Попадет ли он в сетку? Где произойдет отскок от поверхности корта?) Пусть решено выполнять все эти вычисления в другой функции с именем `ComputePath()` (вычисление пути). На следующем этапе необходимо определить, сможет ли другой игрок вернуть мяч, и если это так, потребуется вычислить новую скорость и направление полета мяча. Все эти вычисления можно поместить в функцию с именем `Return()`(ответный удар).

Продолжая таким образом, в конечном счете мы можем прийти к модульной структуре, представленной на рис. 7.3 в виде **структурной схемы**. На этой схеме функции представлены прямоугольниками, а зависимости функций (реализованные вызовами функций) — стрелками. В частности, на этой схеме видно, что вся игра контролируется функцией с именем `ControlGame()` (управление игрой), а для выполнения своей задачи функция `ControlGame()` вызывает на выполнение функции `Serve()` (подача мяча), `Return()` (ответный удар), `ComputePath()` (вычисление пути) и `UpdateScore()` (обновить счет).

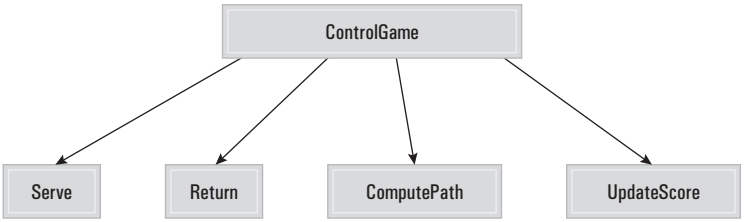



**Рис. 7.3.** Структурная схема простой игры

Обратите внимание, что структурная схема не показывает, как каждая функция выполняет свою задачу. Скорее, функции на ней просто идентифицируются и указываются зависимости между ними. В действительности функция `ControlGame()` может выполнять свою задачу, сначала вызывая функцию `Serve()`, затем неоднократно вызывая функции `ComputePath` ) и `Return()` — до тех пор, пока одна из них не сообщит об ошибке, после чего, перед повторением всего цикла, она вызывает функцию `UpdateScore()`, после чего снова обращается к функции `Serve()`, начиная новый цикл.

На этом этапе мы получили только очень упрощенную схему желаемой программы, но наша точка зрения уже была продемонстрирована. В соответствии с императивной парадигмой мы разрабатывали программу, рассматривая *действия*, которые должны быть выполнены, и поэтому получили проект, в котором модули являются функциями.

А теперь давайте еще раз повторим процедуру разработки этой игры, но на этот раз в контексте объектно-ориентированной парадигмы. На первый взгляд, все просто: в игре есть два игрока, которых можно представить двумя объектами: `PlayerA` и `PlayerB`. Эти объекты будут иметь одинаковую функциональность, но разные характеристики. (Оба должны уметь подавать и отбивать мяч, но могут делать это с разными навыками и силой удара.) Следовательно, эти объекты могут быть экземплярами одного и того же класса. (Напомним, что в главе 6 было введено понятие класса: это шаблон, в котором определены функции, называемые методами, и атрибуты, называемые переменными экземпляра, которые должны быть связаны с каждым объектом.) Этот класс, который мы назовем `PlayerClass`, будет содержать методы `serve()` и `return()`, обеспечивающие имитацию действий игрока, т.е. подачу и ответный удар соответственно. Он также будет содержать атрибуты, такие как `skill` (умение) и `endurance` (выносливость), значения которых будут отражать эти характеристики каждого игрока. На данном этапе проект игры может быть представлен в виде диаграммы, показанной на рис. 7.4. Здесь мы видим, что `PlayerA` и `PlayerB` являются экземплярами класса `PlayerClass` и этот класс содержит атрибуты `skill` и `endurance`, а также методы `serve()` и `returnVolley()`. (Обратите внимание, что на рис. 7.4 имена объектов подчеркнуты, чтобы отличать их от имен классов.)

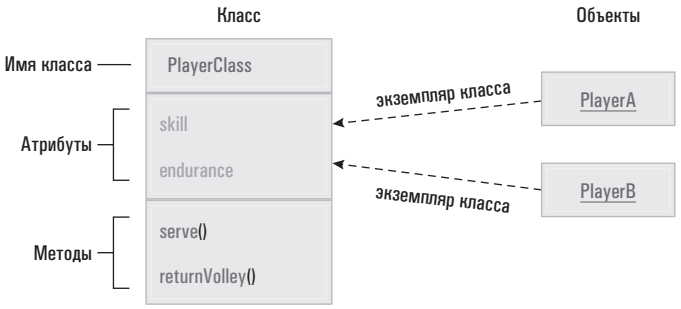



**Рис. 7.4.** Структура класса `PlayerClass` и объекты, созданные на его основе

Далее нам нужен объект, который будет играть роль судьи, определяющего, являются ли действия, совершенные игроками, допустимыми. Например, задел ли поданный мяч сетку и попал ли он в ту часть корта, в которую разрешается попадание при подаче? Для этой цели мы можем создать объект с именем Judge (судья), который будет содержать методы `evaluateServe` (оценка подачи) и `evaluateReturn` (оценка ответного удара). Если объект Judge определяет подачу или ответный удар как допустимый, игра продолжается. В противном случае объект Judge отправляет сообщение другому объекту с именем `Score` (счет) для фиксации соответствующих результатов.

На данный момент проект игры в теннис включает уже четыре объекта: `PlayerA`, `PlayerB`, Judge и `Score`. Чтобы уточнить структуру нашего проекта, рассмотрим последовательность событий, которые могут происходить во время подачи мяча, — она показана на рис. 7.5, на котором объекты нашей программы представлены в виде прямоугольников. Назначение этого рисунка — представление всех взаимодействий, которые имеют место между этими объектами в результате вызова метода `serve()` объекта `PlayerA`. События появляются в хронологическом порядке, соответствующем перемещению вниз по рисунку. Как показано первой горизонтальной стрелкой, объект `PlayerA` сообщает о своей подаче мяча объекту Judge, вызывая его метод `valuServe()` (оценить подачу). Далее объект Judge определяет, что подача мяча прошла успешно, и предлагает объекту `PlayerB` отбить его, вызвав метод `returnVolley()` этого объекта. Цикл передач мяча завершается, когда объект Judge определяет, что объект `PlayerA` допустил ошибку, и просит объект `Score` записать результат.

Как и в случае императивной версии этого примера, объектно-ориентированный вариант программы на этом этапе очень упрощен. Однако мы уже достаточно продвинулись, чтобы увидеть, каким образом выбор объектно-ориентированной парадигмы приводит к модульной структуре, в которой основные компоненты являются объектами.

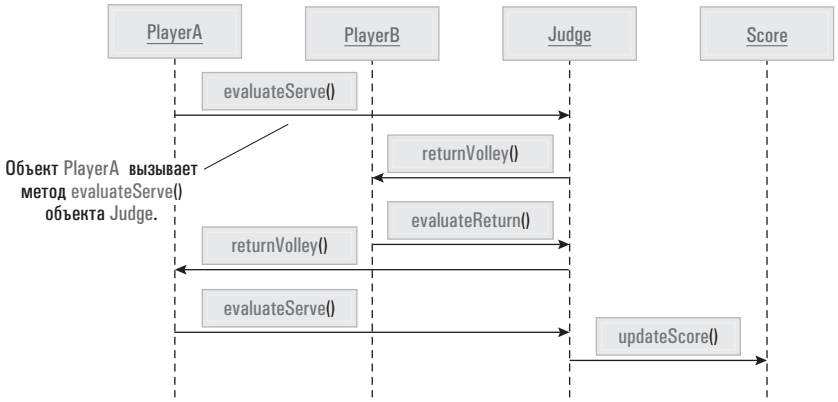



**Рис. 7.5.** Серия взаимодействий между объектами, вызванная выполнением метода `serve()` объекта `PlayerA`

## Связанность модулей

Выше в этой главе модульность была предложена как средство получения управляемого программного обеспечения. Идея состоит в том, что каждая последующая модификация, вероятнее всего, коснется относительно небольшого числа модулей, так что в процессе ее выполнения достаточно будет ограничиться рассмотрением только этой части системы, вместо того чтобы возиться со всем пакетом. Данное утверждение, безусловно, основывается на предположении, что внесение изменений в один из модулей не окажет непредвиденного влияния на работу других модулей системы. Соответственно, при проектировании модульной системы основная задача состоит в обеспечении максимальной независимости отдельных модулей, хотя некоторые взаимосвязи между модулями все-таки необходимы, поскольку данные модули должны образовать согласованно функционирующую систему. Наличие подобных связей между модулями системы называют **связанностью**. Следовательно, задача достижения максимальной независимости модулей соответствует минимизации их связанности. И действительно, одна из метрик, которые используются для измерения сложности программной системы (а следовательно, для получения средства оценки затрат на обслуживание программного обеспечения), заключается в измерении ее межмодульной связанности.

Связанность модулей системы может существовать в нескольких различных формах. Одна из них — это **связанность по управлению**. В этом случае один модуль передает управление другому так, как это происходит при возврате и передаче управления при вызове функции. Структурная схема на рис. 7.3 представляет пример связанности по управлению, которая существует между функциями. В частности, стрелка от модуля `ControlGame()` к модулю `Serve()`

указывает, что первый передает управление второму. Связанность по управлению также присутствует и на рис. 7.5, на котором стрелки определяют последовательность передачи управления от объекта к объекту.

Другая форма связанности модулей — это **связанность по данным**, т.е. совместное использование одних и тех же данных несколькими модулями. Если два модуля манипулируют одним и тем же элементом данных, то изменения, внесенные в один модуль, могут повлиять на работу другого, а изменения в формате самих данных могут оказать влияние на работу обоих модулей.

Связанность по данным между функциями может существовать в двух формах. Одной из них является явная передача данных от одной функции к другой в форме параметров. Такая связь представляется в структурной диаграмме стрелкой между функциями, которая дополнительно помечается с целью определения передаваемых данных. Сама же стрелка задает направление, в котором этот элемент данных передается. Например, на рис. 7.6 представлена расширенная версия рис. 7.3, на котором указано, что функция `ControlGame()` сообщает функции `Serve()`, характеристики какого именно игрока следует имитировать при вызове функции `Serve()`, и что функция `Serve()` при завершении работы передает функции `ControlGame()` информацию о вычисленной ею траектории мяча.

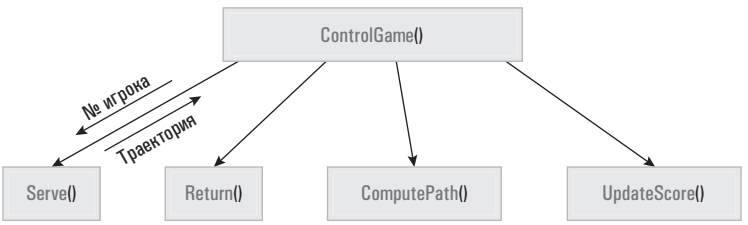



**Рис. 7.6.** Структурная схема, содержащая информацию о связанности по данным

Похожее связывание по данным имеет место между объектами и в объектно-ориентированном варианте проекта. Например, когда объект `PlayerA` просит объект `Judge` оценить его подачу (см. рис. 7.5), он должен передать последнему сведения о траектории мяча. С другой стороны, одно из преимуществ объектно-ориентированной парадигмы состоит в том, что она по своей сути стремится свести связанность по данным между объектами к минимуму. Это связано с тем, что методы внутри объекта обычно включают в себя все те функции, которые манипулируют внутренними данными объекта. Например, объект `PlayerA` будет содержать информацию о характеристиках этого игрока, а также обо всех методах, которые требуют для своего выполнения эту информацию. Соответственно, нет необходимости передавать эту информацию другим объектам, а

следовательно, межобъектная связанность по данным здесь всегда сводится к минимуму.

В противоположность явной передаче данных в качестве параметров, данные также могут быть неявно распределены между модулями в форме **глобальных данных**. Они представляют собой элементы данных, автоматически доступные для всех модулей в системе, что отличает их от локальных элементов данных, доступных только внутри конкретных модулей, если их явно не передают другому модулю. Большинство языков высокого уровня предоставляют способы реализации как глобальных, так и локальных данных, но к использованию в системе глобальных данных следует относиться с осторожностью. Проблема заключается в том, что человеку, пытающемуся внести изменения в модуль, использующий глобальные данные, может быть сложно установить, как именно этот модуль взаимодействует с другими модулями. Короче говоря, использование глобальных данных может снизить полезность модуля в качестве абстрактного инструмента.

## Связность элементов модуля

Почти столь же важной задачей, как минимизация связей между модулями, является достижение максимальной внутренней связности элементов внутри каждого модуля. Термин **связность** (cohesion) относится именно к этим внутренним связям или, другими словами, к степени взаимосвязанности внутренних частей модуля. Чтобы убедиться в важности понятия связности, необходимо выйти за рамки процесса первоначальной разработки системы и обратиться ко всему жизненному циклу программного обеспечения. Если возникает необходимость внести изменения в модуль, то существование множества разнообразных действий внутри него может существенно усложнить процесс, который в противном случае мог бы оказаться совсем простым. Следовательно, помимо поиска любых возможностей уменьшить связывание отдельных модулей, разработчики программного обеспечения должны стремиться к достижению самого высокого уровня связности элементов внутри каждого модуля.

Самая слабая форма связности — **логическая связность**. Этот тип связности внутри модуля строится на том факте, что его внутренние элементы выполняют действия, сходные по своей логической природе. Например, рассмотрим модуль, осуществляющий все связи системы с внешним миром. “Клеем”, скрепляющим элементы такого модуля, является то, что все действия внутри этого модуля так или иначе относятся к обмену информацией с внешним миром. Однако назначение такого обмена в каждом конкретном случае может сильно различаться. Например, одни действия могут быть связаны с получением данных, а другие — с выдачей сообщений об ошибках.

Более сильная форма связности — **функциональная связность**, которая означает, что все части модуля фокусируются на выполнении одного действия. При проектировании в рамках императивной парадигмы функциональная связность часто может быть увеличена за счет выделения подзадач в отдельные модули с последующим использованием этих модулей в качестве абстрактных инструментов. Это хорошо демонстрируется в нашем примере игры в теннис (см. рис. 7.3), в котором модуль `ControlGame()` использует другие модули в качестве абстрактных инструментов, в результате чего он может сосредоточиться на наблюдении за игрой, не отвлекаясь на детали осуществления подачи мяча, ответного удара по мячу и ведения счета.

В объектно-ориентированном проектировании объекты в целом обычно являются только логически связными, так как методы внутри объектов зачастую выполняют слабо связанные действия. Единственное, что объединяет все методы объекта, — это то, что они выполняют эти действия с одним и тем же объектом. Например, в нашей игре в теннис каждый представляющий игрока объект будет содержать методы как для подачи мяча, так и для совершения ответного удара, в которых выполняются существенно разные действия. Следовательно, каждый такой объект представляет собой лишь логически связный модуль. Однако разработчики программного обеспечения должны стремиться делать каждый метод внутри объекта функционально связным. Другими словами, даже если объект в целом является всего лишь логически связным, каждый метод внутри объекта должен выполнять всего лишь одну функционально связную задачу, как показано на рис. 7.7.

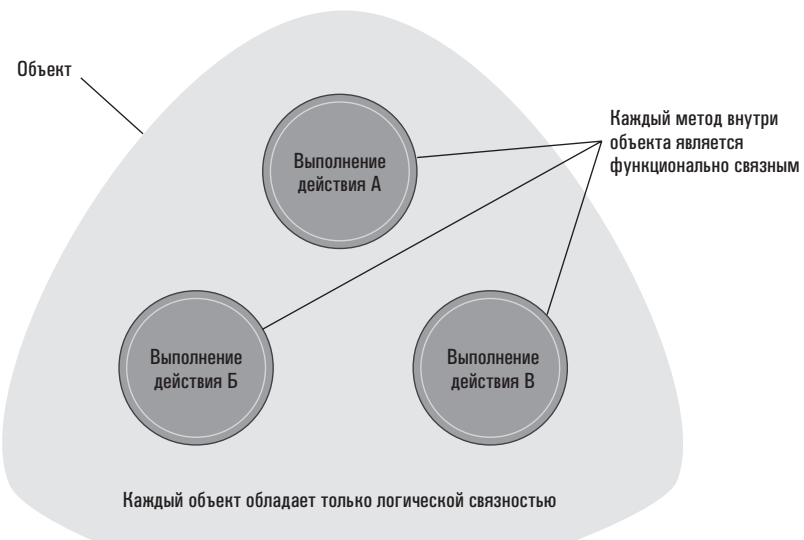



**Рис. 7.7.** Логическая и функциональная связность внутри объекта

## Сокрытие информации

Один из краеугольных камней хорошего модульного дизайна заложен в концепции **сокрытия информации**, которая предполагает ограничение доступности любой информации лишь определенной частью программной системы. Здесь термин *информация* должен интерпретироваться в широком смысле, включая любые сведения о структуре и содержании программного блока. В общем понимании она включает в себя данные, тип используемых структур данных, системы кодирования, внутреннюю композиционную структуру модуля, логическую структуру процедурных единиц и любые другие факторы, касающиеся внутренних свойств модуля.

Смысл сокрытия информации заключается в том, чтобы не допустить в действиях модулей ненужной зависимости или нежелательного влияния на другие модули. В противном случае корректность работы модуля может вызвать сомнения, возможно, из-за ошибок при разработке других модулей или из-за ошибочных действий при модификации программного обеспечения. Например, если модуль не запрещает использование своих внутренних данных со стороны других модулей, то эти данные могут быть ими повреждены. Или, если один модуль спроектирован так, чтобы использовать преимущества внутренней структуры другого, он может начать работать некорректно, если эта внутренняя структура будет изменена.

Важно отметить, что сокрытие информации имеет два аспекта: в первом оно воспринимается как цель проектирования, а во втором — как цель реализации. Модуль должен быть *спроектирован* так, чтобы другие модули не нуждались в доступе к его внутренней информации. С другой стороны, модуль должен быть *реализован* таким способом, который обеспечит неприкосновенность его границ. Примерами первого подхода являются максимизация связности и минимизация связывания. Примеры второго подхода включают использование локальных переменных, применение инкапсуляции и использование четко определенных структур управления.

И в завершение следует отметить, что сокрытие информации занимает центральное место в теме абстракции и использования абстрактных инструментов. Действительно, концепция абстрактного инструмента — это концепция “черного ящика”, внутренние особенности которого могут игнорироваться пользователем, что позволяет ему сосредоточиться исключительно на особенностях того более крупного приложении, над которым он работает. В этом смысле сокрытие информации соответствует концепции изоляции абстрактного инструмента во многом так же, как защищенный от взлома корпус может использоваться для защиты сложного и потенциально опасного электронного оборудования. В обоих случаях обеспечиваются защита внешних пользователей от

внутренних опасностей компонента и одновременно защита его внутреннего содержания от нарушений со стороны пользователей.

## Компоненты

Мы уже упоминали, что одним из препятствий в области разработки программного обеспечения является отсутствие готовых, поставляемых со стороны строительных блоков, из которых можно было бы успешно строить большие программные системы. Модульный подход к разработке программного обеспечения подает большие надежды в этом отношении. В частности, особенно полезной оказывается парадигма объектно-ориентированного программирования, поскольку объекты представляют собой законченные, автономные модули, имеющие четко определенные интерфейсы с их окружением. Как только объект или, вернее, класс, разработан для выполнения определенной роли, его можно использовать для выполнения этой роли в любой программе, требующей подобных действий. Кроме того, наследование обеспечивает средство уточнения определений заранее созданных классов в тех случаях, когда эти определения должны быть настроены в соответствии с потребностями конкретного приложения. Поэтому неудивительно, что для объектно-ориентированных языков программирования C++, Java и C# разработаны обширные библиотеки готовых "шаблонов", на основании которых программисты легко могут реализовать такие объекты, которые необходимы им для выполнения конкретных ролей. В частности, для языка C++ существует стандартная библиотека шаблонов C++ (STL), в среде программирования языка Java доступен интерфейс Java Application Programmer Interface (API), а программисты на языке C# имеют доступ к библиотеке классов .NET Framework Class Library.

Тот факт, что объекты и классы имеют необходимый потенциал для предоставления сборных строительных блоков при разработке программного обеспечения, вовсе не означает, что они идеальны. Одна из проблем заключается в том, что для сборки они предоставляют относительно небольшие блоки. Поэтому объект на самом деле является частным случаем более общего понятия **компонента**, который по определению является повторно используемой единицей программного обеспечения. На практике большинство компонентов создается с использованием объектно-ориентированной парадигмы и имеет вид набора из одного или нескольких объектов, функционирующих как автономная самодостаточная единица.

Исследования в области разработки и использования компонентов привели к появлению новой области, известной как **компонентная архитектура** (также известная как *разработка программного обеспечения на основе компонентов*), в которой традиционную роль программиста заменяет **ассемблер компонентов**, конструирующий программные системы из готовых компонентов, которые

во многих средах разработки представляются в виде значков в графическом интерфейсе. Вместо того чтобы заниматься внутренним программированием компонентов, методология ассемблера компонентов предполагает простой выбор необходимых компонентов из коллекций предопределенных компонентов с последующим их соединением — с минимальной настройкой — для получения желаемой функциональности. И действительно, основное свойство хорошо разработанного компонента состоит в том, что его можно расширить для охвата функций конкретного приложения без необходимости внесения внутренних изменений.

***Основные положения для запоминания***

- **Разработка корректно работающих компонентов программ с последующим их комбинированием способствует созданию корректных программ.**

Той областью, в которой компонентная архитектура нашла для себя благодатную почву, являются системы для смартфонов. Из-за ограниченности ресурсов этих устройств отдельные их приложения на самом деле представляют собой всего лишь набор взаимодействующих компонентов, каждый из которых предоставляет в распоряжение приложения некоторую дискретную функцию. Например, каждый отображаемый в приложении экран обычно представляет собой отдельный компонент. За кулисами могут существовать и другие сервисные компоненты, предназначенные, скажем, для сохранения информации на карте памяти и ее последующего считывания, для выполнения некоторой непрерывной функции (например, воспроизведения музыки) или для доступа к информации через Интернет. Каждый из этих компонентов запускается и останавливается индивидуально, по мере необходимости, с целью эффективного обслуживания пользователя. Тем не менее каждое приложение выглядит как непрерывная серия отображаемых экранов и выполняемых действий.

Помимо мотивации к ограничению использования системных ресурсов, компонентная архитектура смартфонов приносит существенные дивиденды и при интеграции между приложениями. Например, приложение Facebook (известная социальная сеть) при запуске на смартфоне может использовать компоненты приложения Контакты для добавления всех друзей пользователя на Facebook в качестве контактов. Более того, приложение Телефон (приложение, реализующее функции телефонной связи) также может получить доступ к компонентам приложения Контакты для поиска абонента входящего вызова. В результате при поступлении входящего звонка от друга в сети Facebook на экране смартфона может быть отображена его фотография (вместе с его последним сообщением в Facebook).

### 7.4. Вопросы и упражнения

1. Чем роман отличается от энциклопедии в смысле степени связанности, существующей между его элементами, такими как главы, разделы или отдельные записи? Что можно сказать о связности этих элементов?
2. Спортивные мероприятия часто разделены на составляющие единицы. Например, футбольный матч разделен на таймы, а теннисный матч — на сеты. Проанализируйте связь между такими "модулями". В каком смысле такие единицы являются связными?
3. Совместимы ли задачи максимизации связности и минимизации связанности? Другими словами, будет ли естественным образом уменьшаться связанность при возрастании связности?
4. Дайте определения связанности, связности и сокрытию информации.
5. Расширьте структурную диаграмму, представленную на рис. 7.3, включив в нее сообщения, которые должны передаваться между модулями `ControlGame()` и `UpdateScore()`.
6. Нарисуйте диаграмму, подобную приведенной на рис. 7.5 и представляющую последовательность действий в том случае, если объект `PlayerA` выполнит неправильную подачу.
7. Чем отличается работа традиционного программиста от действий ассемблера компонентов?
8. Предполагая, что на большинстве смартфонов установлено несколько приложений класса личных организаторов (Календарь, Контакты, Часы, Почта, Карты, приложения доступа к социальным сетям, мессенджеры и т.д.), какие комбинации их функциональных компонентов вы сочтете полезными и интересными?

## 7.5. Инструменты и методы проектирования

В этом разделе мы исследуем некоторые методы моделирования и системы обозначений, используемые при создании программного обеспечения на этапах анализа и проектирования. Некоторые из них были разработаны еще в те годы, когда в области создания программного обеспечения доминировала императивная парадигма. Позднее одни из них нашли полезное применение и в контексте объектно-ориентированной парадигмы, тогда как другие, такие как структурная схема (см. рис. 7.3), по-прежнему являются специфическими

лишь для императивной среды. Мы начнем с рассмотрения некоторых методов, которые сохранились еще со времен доминирования императивного подхода, а затем перейдем к изучению новых объектно-ориентированных инструментов и обсуждению все возрастающей роли шаблонов проектирования.

### Контроль версий

В настоящее время системы контроля версий являются частью большинства крупных проектов по разработке программного обеспечения. Глобальные и локальные сетевые системы контроля версий позволяют членам команды совместно работать над исходным кодом, предоставляя согласованный механизм для отслеживания отдельных изменений в коде. Инструменты контроля версий, такие как Git, Subversion или Mercurial, позволяют членам группы работать с локальной копией общего проекта, вносить в нее изменения, сделанные другими участниками, работающими параллельно, и направлять им собственные новые изменения, когда они будут к этому готовы. Системы контроля версий могут точно отслеживать, когда и кем каждое конкретное изменение вносится в общий проект, и могут откатывать код до более ранней версии, чтобы отменить изменения, вызвавшие нежелательные последствия. Контроль версий также может использоваться для создаваемой документации, для данных конфигурации и для структуры процесса сборки.

В действительности базовые функции контроля версий теперь включены во многие популярные пакеты программного обеспечения (фактически в любую программу с кнопкой Отменить, позволяющей откатывать множественные изменения), а также в облачные службы, такие как Google Docs.

## Традиционные инструменты разработки

Хотя императивная парадигма предполагает создание программного обеспечения в терминах процедур или функций, способ определения этих функций состоит в том, чтобы рассматривать данные, которыми требуется манипулировать, а не сами функции. Теория говорит о том, что изучая, как данные перемещаются по системе, можно определить точки, в которых либо изменяются форматы данных, либо сливаются или разделяются пути их прохождения. В свою очередь, это именно те места, в которых и происходит обработка, а следовательно, анализ потока данных приводит к идентификации функций. Средством представления информации, полученной в результате таких исследований движения данных, является **диаграмма потоков данных**. На диаграмме потоков данных стрелки определяют пути прохождения данных, овалы представляют точки, в которых осуществляется манипулирование данными, а прямоугольники представляют

источники и места хранения данных. В качестве примера на рис. 7.8 показана элементарная диаграмма потоков данных, представляющая систему выставления счетов пациентам больницы. Обратите внимание: на диаграмме показано, что платежи (поступающие от пациентов) и записи пациентов (поступающие из файлов больницы) объединяются в овале Обработка платежей, из которого обновленные записи возвращаются в файлы больницы.

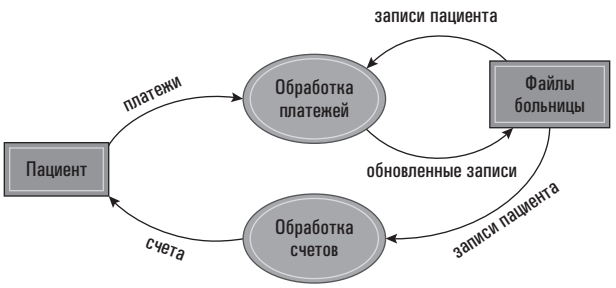



**Рис. 7.8.** Простая диаграмма потоков данных

Диаграммы потоков данных не только помогают идентифицировать процедуры на этапе проектирования в процессе разработки программного обеспечения, но и будут полезны при попытках разобраться в структуре создаваемой системы еще на этапе анализа. Действительно, построение диаграмм потоков данных может служить удобным средством улучшения взаимодействия между клиентами и разработчиками программного обеспечения, поскольку разработчик программного обеспечения пытается понять, чего хочет клиент, а клиент пытается описать свои ожидания. В результате эти диаграммы все еще находят себе применение, несмотря на то что императивная парадигма давно утратила свою популярность.

Другим инструментом, который годами использовался разработчиками программного обеспечения, является **словарь данных**, представляющий собой центральное хранилище информации об элементах данных, использующихся во всей программной системе. Эта информация включает в себя идентификатор, используемый для ссылки на данный элемент, из чего может состоять допустимое значение этого элемента (будет ли элемент всегда представлен числовым или, наоборот, всегда символьным значением; какой диапазон значений допустим для этого элемента), в котором этот элемент хранится (будет ли элемент храниться в файле или в базе данных и, если да, то в какой именно), и где в программном обеспечении имеются ссылки на этот элемент (каким модулям потребуется информация об этом элементе).

Одной из целей создания словаря данных является углубление взаимопонимания между заказчиками разрабатываемой системы программного обеспе-

чения и инженером-программистом, на которого возложена задача преобразования всех потребностей заказчика в спецификации требований к системе. В этом контексте создание словаря данных дает гарантии, что тот факт, что номер детали на самом деле не является числовым значением, будет выявлен еще на этапе анализа, а не обнаружен на более поздней стадии — проектирования или даже реализации. Другой целью, преследуемой при создании словаря данных, является обеспечение единообразия во всей разрабатываемой системе. Обычно именно при создании словаря в описании исходных данных выявляются факты избыточности и противоречивости. Например, элемент, на который в записях по складским запасам ссылаются как на `PartNumber`, в данных по реализации продукции может быть представлен как `PartId`. Более того, в отделе кадров элемент `Name` может использоваться для хранения данных о фамилиях сотрудников, тогда как в системе складского учета элемент `Name` может содержать название материала или детали, сохраняемой на складах.

## UML — унифицированный язык моделирования

Диаграммы потоков данных и словари данных стали важными инструментами в арсенале разработчиков программ задолго до появления объектно-ориентированной парадигмы и до сих пор продолжают находить себе полезное применение, даже несмотря на то что императивная парадигма, для которой они изначально разрабатывались, утратила свою популярность. Теперь мы обратимся к более современному набору инструментов, известному как **UML** (*Unified Modeling Language* — унифицированный язык моделирования), который был разработан уже с учетом особенностей объектно-ориентированной парадигмы. Тем не менее первый инструмент из этого набора, который мы сейчас рассмотрим, будет полезен независимо от выбранной парадигмы, поскольку он предназначен для получения лишь общего представления о создаваемой системе с точки зрения ее пользователя. Этот инструмент — **диаграмма вариантов использования** (use case diagram) или иначе **диаграмма прецедентов**, пример которой представлен на рис. 7.9.

Диаграмма вариантов использования отображает создаваемую систему в виде большого прямоугольника, в котором взаимодействия (называемые **вариантами использования** или **прецедентами**) между системой и ее пользователями представлены в виде овалов, а пользователи системы (называемые **актерами**) представлены в виде стилизованных человечков (даже в тех случаях, когда актер не является человеком). Следовательно, диаграмма на рис. 7.9 показывает, что проектируемая система ведения записей больницы будет использоваться как врачами, так и медсестрами для получения медицинских карточек пациентов.

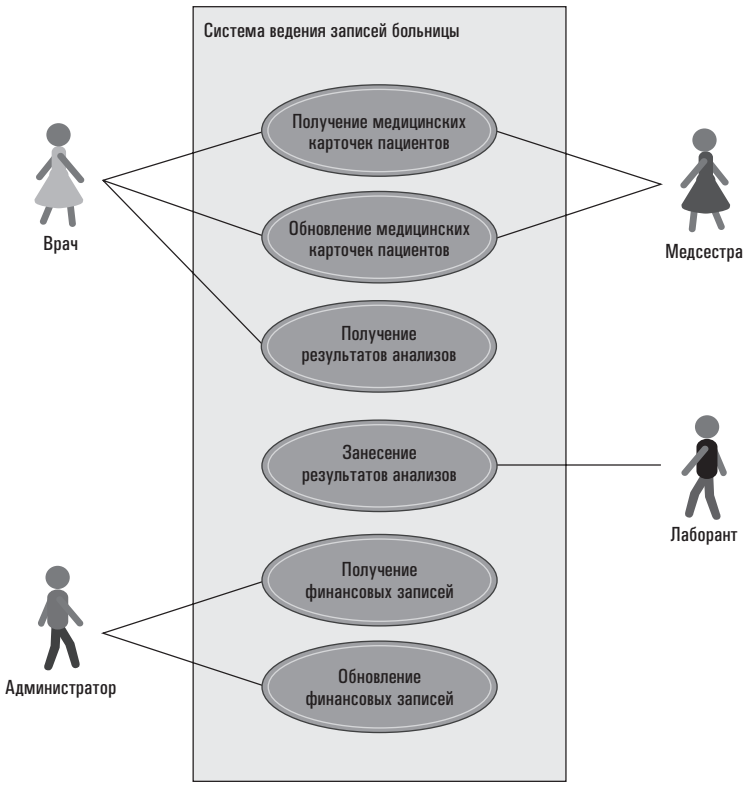



**Рис. 7.9.** Простая диаграмма вариантов использования (прецедентов)

***Основные положения для запоминания***

- **Функциональность программы часто описывается тем, как пользователь будет взаимодействовать с ней.**

Хотя диаграммы вариантов использования представляют взгляд на создаваемую систему программного обеспечения *извне*, язык UML также предлагает множество инструментов для представления *внутренней* организации системы при объектно-ориентированном подходе. Одним из них является **диаграмма классов**, которая представляет собой систему обозначений для представления структуры классов и отношений между классами (называемых в терминологии UML **ассоциациями**). В качестве примера рассмотрим отношения между врачами, пациентами и больничными палатами. Мы предполагаем, что объекты, представляющие эти сущности, создаются на основании классов `Physician`, `Patient` и `Room` соответственно.

На рис. 7.10 показано, как отношения между этими классами могут быть представлены в диаграмме классов языка UML. Здесь классы представлены прямоугольниками, а ассоциации — линиями. Линии ассоциаций могут (или не могут) быть помечены. Если они помечены, жирная стрелка может использоваться для указания направления, в котором должна читаться метка. Например, на рис. 7.10 стрелка после метки заботится о указывает на то, что врач заботится о пациенте, а не пациент заботится о враче. Иногда линиям ассоциации присваиваются две метки с целью обеспечить терминологию для прочтения ассоциации в обоих направлениях. Это положение иллюстрируется на рис. 7.10 на примере ассоциации между классами `Patient` и `Room`.


**Рис. 7.10.** Простая диаграмма классов

В дополнение к указанию ассоциаций между классами диаграмма классов может передавать *кратности* этих ассоциаций. То есть для каждой ассоциации можно указать, сколько экземпляров одного класса может быть связано через нее с экземплярами другого. Эта информация записывается на концах линий связи. В частности, на рис. 7.10 показано, что каждый пациент может занимать одну палату и в каждой палате может лежать нуль или один пациент. (Здесь предполагается, что каждая палата предназначена только для одного пациента.) Звездочка используется для обозначения произвольного неотрицательного числа. Таким образом, звездочка на рис. 7.10 указывает, что каждый врач может заботиться о многих пациентах, тогда как единица с другой стороны этой ассоциации означает, что о каждом пациенте заботится только один, лечащий, врач. (В нашем проекте принимаются во внимание только лечащие врачи.)

Для полноты картины следует отметить, что кратности ассоциаций встречаются в трех основных формах: в виде отношений “один-к-одному”, “один-ко-многим” и “многие-ко-многим”, как показано на рис. 7.11. Отношение “**один-к-одному**” иллюстрируется ассоциацией между пациентами и индивидуальными палатами, в которых они лежат. Здесь каждый пациент может занимать только одну палату, а каждая палата предоставляется только одному пациенту. Отношение “**один-ко-многим**” иллюстрируется ассоциацией между врачами и пациентами. В этом случае один врач связан со многими пациентами, тогда как каждый пациент связан только с одним (лечащим) врачом. Отношения “**многие-ко-многим**” появились бы в этом проекте в том случае, если бы в него дополнительно была включена концепция консультирующих врачей. Тогда каждый врач мог бы быть связан со многими пациентами, которых он консультирует, а каждый пациент мог бы быть связан со многими врачами, предоставлявшими ему консультации.

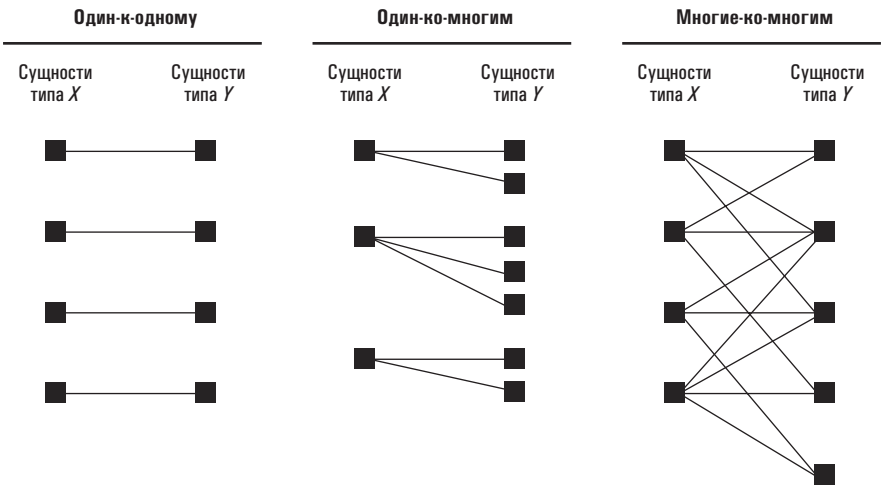



**Рис. 7.11.** Отношения "один-к-одному", "один-ко-многим" и "многие-ко-многим" между сущностями типа *X* и *Y*

В объектно-ориентированном проектировании часто бывает так, что один класс представляет собой более специализированную версию другого. В этих ситуациях говорят, что последний класс является обобщением первого. Язык UML предоставляет специальные обозначения для представления таких обобщений. Пример приведен на рис. 7.12, на котором представлены отношения обобщения между классами `MedicalRecord`, `SurgicalRecord` и `OfficeVisitRecord`. Здесь ассоциации между классами представлены стрелками с наконечниками без заливки, что в языке UML является стандартной нотацией для ассоциаций, отражающих отношение обобщения. Обратите внимание, что на диаграмме каждый класс представлен прямоугольником, содержащим имя, атрибуты и методы класса в формате, ранее использовавшемся на рис. 7.4. Это стандартный способ представления внутренних характеристик класса на диаграмме классов, принятый в языке UML. Информация, представленная на рис. 7.12, состоит в том, что класс `MedicalRecord` является обобщением как для класса `SurgicalRecord`, так и для класса `OfficeVisitRecord`. Фактически это означает, что классы `SurgicalRecord` и `OfficeVisitRecord` содержат все методы и свойства класса `MedicalRecord` плюс собственные компоненты, которые явно указаны в соответствующих прямоугольниках. Таким образом, классы `SurgicalRecord` и `OfficeVisitRecord` содержат свойства `patient`, `doctor` и `dateOfRecord`, но класс `SurgicalRecord` также содержит свойства `surgicalProcedure`, `hospital` и `dateOfDischarge` плюс метод `dischargePatient()`, тогда как класс `OfficeVisitRecord` содержит собственные свойства `symptoms` и `diagnosis`. Кроме того, все три класса включают метод `printRecord()`, позволяющий распечатать медицинскую карту. При этом

методы `printRecord()` в классах `SurgicalRecord` и `OfficeVisitRecord` являются специализациями метода `printRecord()` из класса `MedicalRecord`, поскольку каждый из них дополнительно будет печатать информацию, специфическую для его класса.

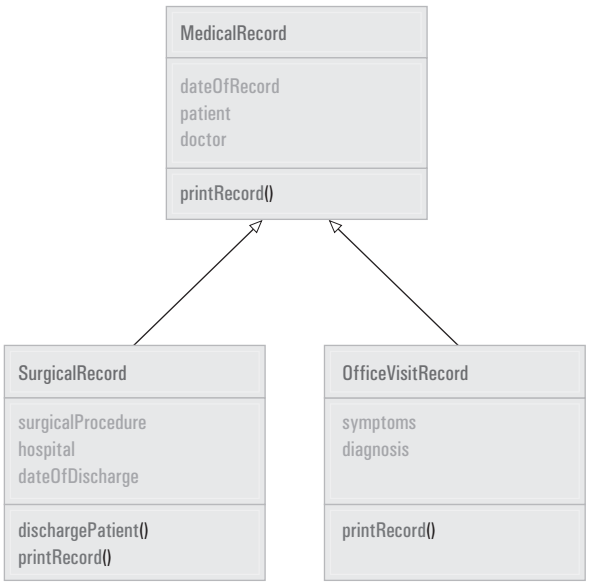



**Рис. 7.12.** Диаграмма классов с указанием отношений обобщения

Вспомните из главы 6 (раздел 6.5), что естественным способом реализации обобщений в среде объектно-ориентированного программирования является использование механизма наследования. Однако многие разработчики программного обеспечения предупреждают, что наследование не может использоваться для всех случаев обобщения. Причина в том, что наследование существенно повышает уровень связанности между классами — связанности, которая может оказаться нежелательной на более позднем этапе жизненного цикла программного обеспечения. Например, поскольку изменения внутри класса автоматически отражаются во всех унаследованных от него классах, кажущееся незначительным изменение, внесенное на этапе сопровождения программы, может привести к непредвиденным последствиям. В качестве примера предположим, что компания для своих сотрудников открыла базу отдыха, а это означает, что все люди, имеющие право пользоваться базой отдыха, являются сотрудниками компании. Чтобы получить список всех тех, кто имеет право пользоваться базой отдыха, программист мог бы воспользоваться механизмом наследования для создания класса `RecreationMember` на основе уже определенного ранее класса `Employee`. Однако если впоследствии компания преуспеет и решит сделать базу отдыха доступной и для членов семей сотрудников или, допустим, для

пенсионеров компании, то встроенную связь между классом `Employee` и классом `RecreationMember` потребуется разорвать. Таким образом, использовать механизм наследования просто для удобства недопустимо. В действительности применение этого механизма должно быть ограничено только теми случаями, в которых реализуемое обобщение является неизменным.

Диаграммы классов представляют статические свойства проектируемой программы. Они не отражают последовательности событий, которые будут происходить во время ее выполнения. Чтобы выразить такие динамические особенности, в языке UML предлагается несколько типов диаграмм, которые в совокупности называют **диаграммами взаимодействия**. Одним из типов диаграмм взаимодействия является **диаграмма последовательности**, отображающая связь между объектами (такими, как актеры, полные программные компоненты или отдельные объекты), которые участвуют в выполнении задачи. Эти диаграммы аналогичны рис. 7.5 в том смысле, что объекты на них представлены прямоугольниками с исходящими из них пунктирными линиями, направленными вниз. Каждый прямоугольник вместе с его пунктирной линией называется **линией жизни**. Взаимодействие между объектами — например, запрос на обслуживание — представляется помеченными линиями со стрелками (сигналами), соединяющими соответствующие линии жизни, где метка определяет запрашиваемое действие, а стрелка направлена в сторону того объекта, которому направлен запрос. Эти линии размещаются на диаграмме в хронологическом порядке, предполагающем чтение диаграммы сверху вниз. Взаимодействие, которое имеет место, когда объект завершает запрошенную задачу и возвращает управление запросившему ее выполнение объекту, как и в традиционном возврате из процедуры, представляется немаркированной стрелкой, направленной в сторону линии жизни источника запроса.

Таким образом, рис. 7.5, по сути, является диаграммой последовательности. Однако синтаксису, использованному на рис. 7.5, свойственно несколько недостатков. Во-первых, он не позволяет уловить симметрию между двумя игроками. Потребуется нарисовать отдельную диаграмму, чтобы представить на ней розыгрыш мяча, начинающийся с подачи игрока B, даже несмотря на то что эта последовательность взаимодействий будет очень похожа на последовательность, когда подает игрок A. Более того, в то время как на рис. 7.5 показан один конкретный розыгрыш мяча, в общем случае этот розыгрыш может продолжаться бесконечно. Формальные диаграммы последовательности предлагают способы представления подобных нюансов на одной диаграмме, и хотя нам нет необходимости подробно их изучать, будет полезно просто взглянуть на формальную диаграмму последовательности, показанную на рис. 7.13, которая представляет обобщенный розыгрыш мяча в терминах нашего проекта игры в теннис.

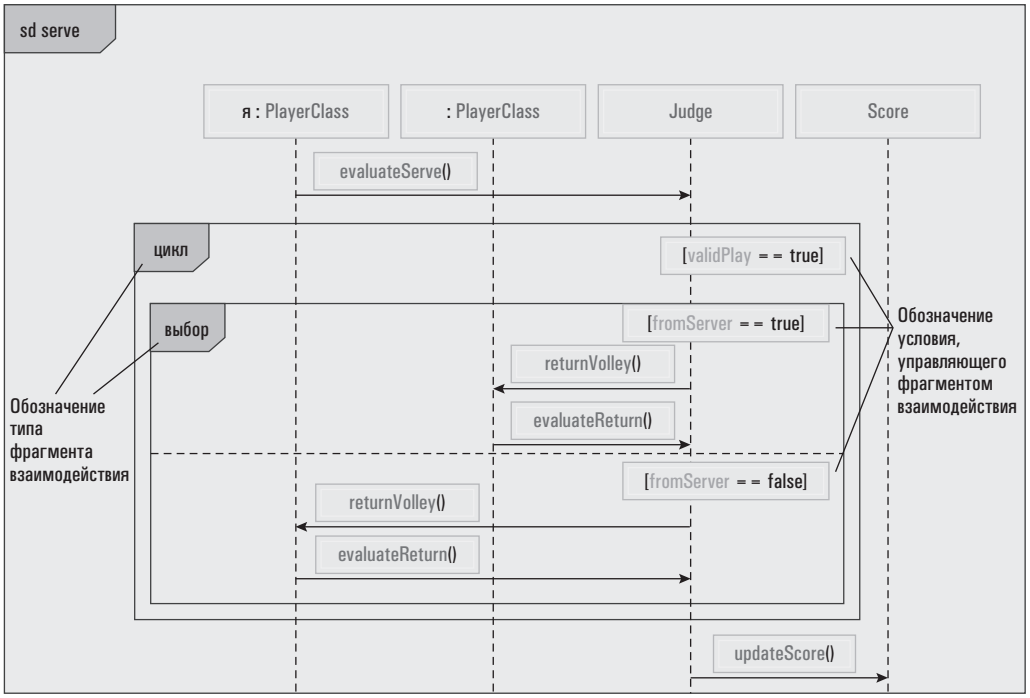



**Рис. 7.13.** Диаграмма последовательности, представляющая общую схему розыгрыша мяча

Обратите внимание: на рис. 7.13 видно, что вся диаграмма последовательности заключена в прямоугольник (называемый **фреймом**). В верхнем левом углу этой рамки находится пятиугольник, содержащий символы **sd** (означающие "диаграмма последовательности" — *sequence diagram*), за которыми следует идентификатор. Этот идентификатор может быть именем, идентифицирующим общую последовательность, или, как на рис. 7.13, именем метода, который вызывается для инициирования последовательности. Обратите внимание, что в отличие от рис. 7.5, прямоугольники, представляющие на рис. 7.13 игроков, не относятся к конкретным игрокам, а просто указывают, что они представляют объекты "типа" `PlayerClass`. Один из них обозначен как "**я**": это означает, что это тот объект, метод подачи которого был активирован для инициирования данной последовательности.

Другое важное замечание относительно рис. 7.13 касается двух внутренних прямоугольников. Это **фрагменты взаимодействия**, которые используются для представления альтернативных последовательностей на одной диаграмме. На рис. 7.13 присутствуют два фрагмента взаимодействия. Один из них помечен как "**цикл**", другой — как "**выбор**". По сути, это структуры `while` и `if-else`, с которыми вы уже познакомились при обсуждении языка Python в разделе 5.2.

Фрагмент взаимодействия "**цикл**" указывает, что события в его границах должны повторяться до тех пор, пока объект `Judge` определяет, что значение свойства `validPlay` равно `true`. Фрагмент взаимодействия "**выбор**" указывает, что одна из его альтернатив должна быть выполнена в зависимости от того, каким является текущее значение свойства `fromServer`: `true` или `false`.

И наконец, на данном этапе целесообразно (хотя это и не являются частью языка UML) познакомиться с использованием **CRC-карт** (*Class-Responsibility-Collaboration* — класс-ответственность-кооперация), поскольку этот метод играет важную роль при верификации объектно-ориентированных проектов. CRC-карта — это просто небольшая бумажная карточка с описанием объекта. Суть метода CRC-карт состоит в том, что разработчик программного обеспечения подготавливает подобные карточки для каждого объекта в проектируемой системе, а затем использует их для представления объектов в процессе имитации работы системы, — возможно, на рабочем столе или в "театрализованном" эксперименте, когда каждый член команды разработчиков держит CRC-карту и играет роль объекта таким образом, как это описано в его карте. Такое моделирование (часто называемое **сквозным структурным контролем**) показало себя как весьма полезный способ выявления недостатков в проекте еще до начала его реализации.

## Шаблоны проектирования

Для разработчиков программного обеспечения важным инструментом, возможности которого со временем только увеличиваются, является постоянно расширяющаяся коллекция шаблонов проектирования. **Шаблон проектирования** — это заранее разработанная архитектурная структура, предназначенная для решения регулярно возникающей проблемы при разработке программного обеспечения. Например, шаблон проектирования **`Адаптер`** предоставляет способ решения проблемы, которая часто возникает при создании программного обеспечения из готовых модулей. В частности, готовый модуль может иметь функциональные возможности, необходимые для решения рассматриваемой задачи, но при этом может не иметь интерфейса, совместимого с разрабатываемым приложением. В таких случаях шаблон **`Адаптер`** обеспечивает стандартный подход в виде "обертывания" требуемого модуля другим модулем, обеспечивающим трансляцию сигналов между интерфейсом требуемого модуля и внешним миром, что и позволяет использовать готовый модуль в приложении.

Другим известным шаблоном проектирования является шаблон **`Декоратор`**. Он предлагает способ разработки системы, которая выполняет различные комбинации одних и тех же действий в зависимости от текущей ситуации. Подобные системы могут требовать реализации бесчисленного множества вариантов

действий, что без тщательного проектирования часто приводит к созданию чрезвычайно сложного программного обеспечения. Однако использование шаблона **Декоратор** позволяет реализовать такую систему стандартным способом и получить полностью контролируемое решение.

Выявление повторяющихся проблем, а также создание и каталогизация шаблонов проектирования, предназначенных для их решения, является непрекращающимся процессом в области разработки программного обеспечения. Однако задача здесь состоит не в том, чтобы просто найти решения определенных проблем в проектировании, а в том, чтобы найти *высококачественные* решения, обеспечивающие необходимую гибкость на более поздних этапах жизненного цикла программного обеспечения. По этой причине соглашения о хороших принципах проектирования, таких как минимизация связывания и максимизация связности, играют важную роль в разработке шаблонов проектирования.

Результаты прогресса в разработке шаблонов проектирования находят свое отражение в библиотеках инструментов, доступных в современных пакетах разработки программного обеспечения, таких как среда программирования языка Java, предоставляемая корпорацией Oracle, или среда программирования .NET Framework, предоставляемая корпорацией Microsoft. В действительности многие из “шаблонов”, которые можно найти в подобных “наборах инструментов”, по сути являются каркасами шаблонов проектирования, обеспечивающих получение готовых высококачественных решений для различных задач проектирования.

## 7.5. Вопросы и упражнения

1. Нарисуйте диаграмму потоков данных, представляющую тот поток данных, который возникает, когда читатель берет книгу в библиотеке.
2. Нарисуйте диаграмму вариантов использования в системе ведения записей библиотеки.
3. Нарисуйте диаграмму классов, представляющую отношения между путешественниками и гостиницами, в которых они проживают.
4. Нарисуйте диаграмму классов, отражающую тот факт, что человек является обобщением работника. Включите некоторые атрибуты, которые могут принадлежать каждому.
5. Преобразуйте рис. 7.5 в полную диаграмму последовательности.
6. Какую роль в процессе разработки программного обеспечения играют шаблоны проектирования?

В заключение следует отметить, что появление шаблонов проектирования в области разработки программного обеспечения является примером того, как различные области человеческой деятельности могут оказывать плодотворное влияние друг на друга. Истоки идеи применения шаблонов проектирования лежат в исследованиях Кристофера Александера в области традиционной архитектуры. Его целью было выявление элементов, которые способствуют созданию высококачественных архитектурных проектов для зданий или комплексов зданий, с последующей разработкой шаблонов проектирования, включающих эти элементы. Сегодня многие из его идей приняты и в области разработки программного обеспечения, а его работа продолжает вдохновлять многих разработчиков в этой области.

## 7.6. Обеспечение качества программ

Частые случаи неправильного функционирования программного обеспечения, перерасхода средств и несоблюдения сроков реализации проектов требуют совершенствования методов контроля качества программного обеспечения. В этом разделе мы рассмотрим некоторые направления, активно изучаемые в этой области.

### Область применения средств обеспечения качества ПО

В начальный период развития вычислительной техники проблема создания качественного программного обеспечения воспринималась как устранение в программах ошибок, возникавших в процессе их реализации. Далее в этом разделе мы обсудим прогресс, достигнутый в этом направлении. Однако сегодня сфера обеспечения качества программного обеспечения вышла далеко за рамки процесса отладки и включает такие направления, как улучшение процедур разработки программного обеспечения, разработку учебных программ, которые во многих случаях приводят к сертификации, и установление стандартов, на которых строится разработка качественного программного обеспечения. В этой связи выше уже отмечалась важная роль таких организаций, как ISO, IEEE и ACM в повышении профессионализма и внедрении стандартов оценки контроля качества в компаниях, занимающихся разработкой программного обеспечения. Конкретным примером являются стандарты серии ISO 9000, касающиеся многих видов промышленной деятельности, таких как проектирование, производство, монтаж и обслуживание. Другим примером являются стандарты ISO/IEC 33001, представляющие собой серию стандартов, совместно разработанных ISO и Международной электротехнической комиссией (IEC — *International Electrotechnical Commission*).

Многие крупные поставщики программного обеспечения в настоящее время требуют, чтобы организации, которые они нанимают для разработки программного обеспечения, соответствовали таким стандартам. В результате компании-разработчики программного обеспечения создают **группы обеспечения качества** (SQA — *Software Quality Assurance*) программного обеспечения, в обязанности которых входит контроль и обеспечение соблюдения требований систем контроля качества, принятых в организации. Таким образом, в случае традиционной модели водопада группе SQA будет поручено утвердить спецификацию требований к программному обеспечению до начала этапа проектирования или утвердить разработанный проект и связанные с ним документы до начала реализации.

***Основные положения для запоминания***

- **Сотрудничество позволяет упростить поиск и исправление ошибок при разработке программ.**

Сегодня в основу прилагаемых усилий по контролю качества положено несколько направлений. Одним из них является учет. Крайне важно, чтобы каждый шаг в процессе разработки был точно задокументирован для использования в будущем. Однако эта цель не соответствует природе человека: всегда имеет место большой соблазн принимать или изменять решения без обновления соответствующих документов. В результате есть определенная вероятность того, что записи будут неправильными, а следовательно, их использование на будущих этапах может вводить в заблуждение. В этом заключается важное преимущество CASE-инструментов. Они значительно упрощают выполнение таких заданий, как перерисовка диаграмм или обновление словаря данных, в сравнении с ручными методами. Следовательно, обновления, скорее всего, будут сделаны, и окончательная документация, скорее всего, будет точной. (Этот пример является лишь одним из многих случаев, когда технология разработки программного обеспечения должна обеспечивать преодоление недостатков, свойственных природе человека. Другие такие недостатки включают личностные конфликты, ревность и столкновение интересов исполнителей, которые неизбежно возникают, когда люди работают вместе.)

Еще одна тема, связанная с обеспечением качества ПО, — это проведение аудитов, во время которых различные стороны, участвующие в проекте разработки программного обеспечения, собираются для обсуждения конкретной темы. Такие аудиты проводятся на протяжении всего процесса разработки программного обеспечения с целью анализа требований, анализа проектного решения и

анализа его реализации. Они могут иметь форму демонстрации и обсуждения прототипа на ранних этапах анализа требований, сквозного структурного контроля, осуществляемого силами членов команды разработчиков программного обеспечения, или координационных встреч между программистами, реализующими взаимосвязанные части проекта. Такие встречи и просмотры, проводимые на регулярной основе, обеспечивают каналы коммуникации, благодаря которым удается избежать недопонимания и исправить ошибки еще до того, как они приведут к катастрофическим последствиям. Значимость таких встреч подтверждается тем фактом, что они специально рассматриваются в стандарте IEEE по вопросу аудита программного обеспечения, известном как IEEE 1028.

Некоторые аудиты и просмотры имеют ключевое значение. Примером является встреча представителей заказчиков проекта и команды разработчиков программного обеспечения, на которой утверждается окончательная спецификация требований к программному обеспечению. И действительно, утверждение подобного документа знаменует собой конец фазы формального анализа требований и закладывает основу, на которой будет строиться вся дальнейшая работа над проектом. Тем не менее все проверки, аудиты и просмотры важны и для контроля качества обязательно должны быть задокументированы как часть постоянного процесса ведения документации.

## Тестирование программного обеспечения

Хотя обеспечение качества программного обеспечения в настоящее время признано в качестве предмета, пронизывающего весь процесс разработки ПО, тестирование и проверка самих программ также продолжает оставаться постоянным предметом исследований. В разделе 5.6 рассматривались методы верификации алгоритмов в строгом математическом смысле. Однако было сделано заключение, что б*о*льшая часть создаваемого сегодня программного обеспечения все еще “верифицируется” посредством тестирования. К сожалению, тестирование в лучшем случае можно расценивать как неточную науку. Проведя тестирование, мы не можем утверждать, что некоторая часть программы правильна, если не было выполнено достаточно проверок, чтобы исчерпать все возможные сценарии. Но даже для простых программ могут существовать миллиарды возможных путей ее выполнения. А это означает, что тестирование всех возможных путей выполнения достаточно сложной программы — задача просто невыполнимая.

С другой стороны, создатели программного обеспечения разработали методы тестирования программ, повышающие вероятность обнаружения существующих в них ошибок с использованием ограниченного количества тестов. Один из таких методов основан на наблюдении, что ошибки в программном

обеспечении имеют тенденцию собираться в группы. Иначе говоря, опыт показывает, что в крупной системе программного обеспечения всегда существует небольшое число модулей, являющихся более проблематичными, чем все остальные. Следовательно, обнаружив эти модули и проверив их более тщательно, можно выявить б*о*льшую часть существующих в системе ошибок, даже если все остальные модули будут протестированы обычным, менее тщательным образом. Это предположение часто называют **принципом Парето**, в честь экономиста и социолога Вильфредо Парето (Vilfredo Pareto, 1848–1923), который заметил, что малая часть населения Италии контролирует б*о*льшую часть богатств этой страны. В области разработки программного обеспечения принцип Парето утверждает, что желаемого результата часто можно достичь быстрее, если приложить больше усилий в областях концентрации ошибок.

Еще один метод тестирования ПО, называемый **тестированием основных путей**, состоит в разработке набора контрольных данных, гарантирующего, что каждая инструкция в программе будет выполнена хотя бы один раз. Для подготовки таких наборов были разработаны методы, построенные на основе математической теории графов. Поэтому, хотя и нельзя будет утверждать, что все возможные пути выполнения в системе программного обеспечения будут проверены, можно гарантировать, что в процессе тестирования каждая инструкция в программном коде системы будет выполнена как минимум один раз.

Методы, основанные на принципе Парето, и способ тестирования основных путей предполагают знание внутренней структуры тестируемого программного обеспечения. Следовательно, они относятся к категории тестирования по принципу "**прозрачного ящика**", при котором подразумевается, что внутреннее устройство программного обеспечения известно тестирующему и он использует это знание при разработке тестов. В противоположность этому при тестировании по принципу "**черного ящика**" выполняемая проверка не может основываться на знании внутренней структуры тестируемого программного обеспечения. Короче говоря, тестирование по принципу черного ящика выполняется с точки зрения пользователя системы. В этом случае анализируется не то, как именно программа функционирует при решении задачи, а исключительно то, насколько правильно она работает в смысле точности достигнутых результатов и скорости ее выполнения.

Один из методов, которые обычно относятся к концепции тестирования по принципу черного ящика, именуется **анализом граничных условий**. Он предполагает определение диапазонов данных, называемых **классами эквивалентности**, в которых программное обеспечение должно работать единообразно, и тестирование программного обеспечения на данных, близких к границам этих диапазонов. Например, если предполагается, что в программе допускается введение исходных значений только из конкретно заданного диапазона, работу

программы следует проверить при вводе наименьшего и наибольшего значений из допустимого диапазона. Если же программное обеспечение должно координировать множество различных действий, то его работу следует проверять с использованием множества действий, имеющих максимальные требования к системе. Положенная в основу этого метода теория заключается в том, что путем идентификации классов эквивалентности количество тестовых случаев можно минимизировать, поскольку правильная работа для нескольких примеров в классе эквивалентности может рассматриваться как успешная проверка программного обеспечения для всего класса. Более того, наилучший шанс идентифицировать ошибку в классе — использовать данные на границах класса.

Еще одним методом тестирования по принципу черного ящика является **бета-тестирование**, при котором предварительная версия программного обеспечения предоставляется сегменту целевой аудитории с целью изучения работы этого ПО в реальных условиях, прежде чем окончательная версия данного программного продукта будет утверждена и выпущена на рынок. (Подобное тестирование, проводимое на сайте разработчика, называется **альфа-тестированием**.) Преимущества бета-тестирования выходят далеко за рамки традиционного обнаружения ошибок. Получение общих отзывов от потенциальных клиентов (как положительных, так и отрицательных) может оказать существенную помощь в уточнении рыночных стратегий. Более того, раннее распространение

## 7.6. Вопросы и упражнения

1. Какова роль группы обеспечения качества (SQA) в организации по разработке программного обеспечения?
2. Как человеческая природа работает против обеспечения качества ПО?
3. Определите два типа мероприятий, которые применяются в процессе разработки для повышения качества.
4. Какая проверка при тестировании программного обеспечения является успешной, нашедшая или не нашедшая ошибки?
5. Какие методы можно предложить для обнаружения в системе модулей, которые необходимо тестировать более тщательно, чем все остальные?
6. Что можно считать хорошим тестом для проверки работы пакета программного обеспечения, разработанного для сортировки списка, содержащего не более 100 элементов?

бета-версии программного обеспечения помогает сторонним разработчикам ПО в разработке своих совместимых продуктов. Например, в случае новой операционной системы для рынка настольных ПК распространение бета-версии стимулирует разработку разнообразного совместимого программного обеспечения, так что окончательная версия операционной системы появится на полках магазинов уже в окружении сопутствующих программных продуктов. Более того, наличие бета-тестирования помогает создать на рынке атмосферу предвкушения, которая повышает популярность продукта, а значит, увеличивает объем его продаж.

## 7.7. Документирование программного обеспечения

Программная система будет бесполезна, пока люди не научатся ее использовать и обслуживать. Следовательно, документация является важной частью конечного программного продукта, а ее разработка — важная тема в технологии разработки программного обеспечения.

Документация по программному обеспечению служит трем целям, что ведет к трем категориям создаваемых документов: пользовательская документация, системная документация и техническая документация. Назначение **пользовательской документации** состоит в объяснении возможностей программного обеспечения и описании того, как их использовать. Она предназначена для чтения пользователем программного обеспечения и поэтому выражается в терминологии приложения, т.е. должна носить нетехнический характер.

Сегодня документация пользователя считается важным инструментом маркетинга. Хорошая пользовательская документация в сочетании с хорошо разработанным интерфейсом пользователя делает программный пакет более доступным и, следовательно, способствует увеличению объема его продаж. Осознавая это, многие разработчики программного обеспечения для выполнения данной части проекта нанимают известных авторов технической литературы или представляют предварительные версии своих продуктов независимым авторам в надежде, что соответствующие книги уже поступят в продажу к тому моменту, когда само программное обеспечение появится на рынке.

Документация пользователя традиционно имеет форму учебного руководства в виде обычной книги или буклета, но все чаще доступна в электронном виде в Сети или как часть самого программного обеспечения. Это дает пользователю возможность обращаться к документации непосредственно при использовании программного обеспечения. В этом случае информация может быть разбита на небольшие блоки, иногда называемые страницами справки, которые могут автоматически отображаться на экране, если пользователь слишком долго раздумывает при переходе от одной команды к другой.

Назначение **системной документации** состоит в описании внутренней структуры программного обеспечения, чтобы обеспечить возможность сопровождения системы на более позднем этапе ее жизненного цикла. Основным компонентом системной документации является исходная версия всех программ системы. Очень важно, чтобы эти программы были представлены в читабельном виде, поэтому разработчики программного обеспечения используют хорошо разработанные языки программирования высокого уровня, сопровождают текст программы комментариями, а также применяют технологию модульного проектирования, что позволяет представить каждый модуль как отдельный, согласованно работающий элемент. На практике многие компании, выпускающие программное обеспечение, приняли ряд правил, которым должны следовать их работники при написании программ. Сюда входят соглашения об использовании отступов в исходных текстах программ; соглашения о присвоении имен, устанавливающие различия между именами переменных, констант, объектов, классов и т.д.; и соглашения по документированию, гарантирующие, что все написанные программы будут достаточно документированы. Подобные соглашения обеспечивают единообразие создаваемого программного обеспечения в рамках всей компании, что существенно упрощает процесс его сопровождения.

***Основные положения для запоминания***

- **Программная документация помогает программистам создавать программы и поддерживать их корректность, обеспечивая эффективное решение возникающих проблем.**

Другим компонентом системной документации является проектная документация, включая спецификацию требований к программному обеспечению и записи о том, как эти спецификации были получены во время проектирования. Эта информация будет полезна при модификации программного обеспечения, поскольку указывает, почему программное обеспечение было реализовано именно таким, каким оно есть, — эта информация позволяет уменьшить вероятность того, что изменения, внесенные во время модификации программы, нарушат целостность системы.

Назначением **технической документации** является описание того, как данную систему программного обеспечения следует устанавливать и обслуживать (например, настраивать параметры ее функционирования, устанавливать обновления и сообщать о проблемах разработчику этого ПО). Техническая документация программного обеспечения аналогична документации, предоставляемой механикам в автомобильной промышленности. В этой документации

не сообщается, как автомобиль был спроектирован и построен (подобно системной документации), и не объясняется, как управлять автомобилем и пользоваться его системой обогрева/охлаждения (подобно пользовательской документации). Вместо этого в ней описывается, как обслуживать компоненты автомобиля, например как заменить коробку передач или отследить возникающую время от времени проблему в электропроводке.

В сфере персональных компьютеров различия между технической документацией и пользовательской документацией довольно размыты, поскольку их пользователь часто является одновременно и тем человеком, который устанавливает и обслуживает программное обеспечение. Однако в многопользовательской среде это различие выражено заметнее. Здесь техническая документация предназначена прежде всего для системного администратора, который отвечает за обслуживание всего ПО, установленного в сфере его обслуживания, что обеспечивает пользователям возможность доступа к программным пакетам как к абстрактным инструментам.

### 7.7. Вопросы и упражнения

1. Какие существуют формы документации на программное обеспечение?
2. На какой фазе (или фазах) жизненного цикла программного обеспечения создают его документацию?
3. Что важнее, программа или ее документация?

## 7.8. Интерфейс “человек–машина”

Вспомним из материала раздела 7.2, что одной из задач при анализе требований является определение того, как создаваемая система программного обеспечения будет взаимодействовать со своей средой. В этом разделе мы рассмотрим темы, связанные с этим взаимодействием, когда оно включает в себя общение с людьми — вопрос, имеющий очень большое значение. В конце концов, люди должны получить возможность использовать программную систему как абстрактный инструмент. И этот инструмент должен быть прост в обращении и предназначен для минимизации (в идеале — для исключения) ошибок коммуникации между системой и ее пользователями-людьми. Это означает, что интерфейс системы должен быть спроектирован для удобства *людей*, а не просто для удобства разработки программной системы.

Важность хорошего дизайна интерфейса дополнительно подчеркивается тем фактом, что системный интерфейс может произвести на пользователя более сильное впечатление, чем любая другая характеристика системы. В конце концов, человек склонен рассматривать систему с точки зрения удобства ее использования, а не с точки зрения того, насколько умно она решает свои внутренние задачи. Для человека выбор между двумя конкурирующими системами, вероятнее всего, будет основан на особенностях интерфейсов этих систем. Следовательно, дизайн интерфейса системы может в конечном итоге стать определяющим фактором успеха или неудачи всего проекта разработки программного обеспечения.

По этим причинам интерфейс “человек–машина” стал важной проблемой на этапе разработки требований к проектам создания ПО и является постоянно расширяющейся областью в сфере разработки программного обеспечения. Фактически некоторые даже утверждают, что изучение интерфейсов “человек–машина” — это вообще отдельная область компьютерных наук.

Наибольшую выгоду в результате исследований, проведенных в этой области, получил интерфейс смартфона. Чтобы достичь поставленных целей и получить удобное устройство карманного размера, все элементы традиционного интерфейса “человек–машина” (полноразмерная клавиатура, мышь, полосы прокрутки, меню) были заменены новыми решениями, такими как жесты, выполняемые на сенсорном экране, голосовые команды и виртуальные клавиатуры с расширенным автозаполнением слов и фраз. Хотя все это в целом уже представляет собой значительный прогресс, большинство пользователей смартфонов утверждают, что есть еще много возможностей для дальнейших инноваций.

Исследования в области дизайна интерфейса “человек–машина” в значительной степени опираются на области инженерии, называемые **эргономикой**, которая отвечает за проектирование систем, соответствующих физическим способностям человека, и **когнетикой**, которая отвечает за проектирование систем, которые гармонируют с интеллектуальными способностями людей. Из этих двух эргономика понимается лучше, в основном потому, что люди веками физически взаимодействовали с машинами. Соответствующие примеры можно найти в древних инструментах, оружии и транспортных системах. Большая часть этой истории самоочевидна, однако иногда применение эргономики бывало и нелогичным. Часто цитируемым примером является конструкция клавиатуры пишущей машинки (которая теперь перевоплотилась в клавиатуру компьютера), в которой клавиши преднамеренно были расположены таким образом, чтобы уменьшить скорость работы машинистки, поскольку механическая система рычагов, использовавшаяся в ранних машинах, при увеличении скорости могла заедать.

Интеллектуальное (или ментальное) взаимодействие с машинами, напротив, является относительно новым явлением, и следовательно, именно когнетика предлагает более высокий потенциал для плодотворных исследований и проливающих свет идей. Зачастую находки в этой области оказываются интересны своей тонкостью. Например, люди формируют привычки — на первый взгляд, это хорошая черта, потому что она может способствовать повышению эффективности. Но привычки могут приводить к ошибкам, даже если дизайн интерфейса, безусловно, позволяет решить проблему. Рассмотрим процесс, когда человек просит типичную операционную систему удалить файл. Чтобы избежать непреднамеренного удаления, в большинстве системных интерфейсов в ответ на такой запрос пользователя просят подтвердить его намерения, например, с помощью сообщения “Вы действительно хотите удалить этот файл?” На первый взгляд, такое обязательное подтверждение требования пользователя, казалось бы, должно решить любую проблему непреднамеренного удаления файла. Однако после достаточно продолжительного периода использования системы у человека вырабатывается привычка автоматически отвечать на этот вопрос утвердительно. В результате задача удаления файла перестает быть двухэтапным процессом, состоящим из ввода команды удаления и предоставления обдуманного ответа на вопрос, заданный системой. Вместо этого он превращается в одноэтапный процесс “удалить–да”, означающий, что к тому времени, когда человек поймет, что он выдал ошибочную команду на удаление, запрос от системы уже будет подтвержден, а удаление выполнено.

Формирование привычек может также вызвать проблемы, если человек постоянно использует несколько пакетов прикладного программного обеспечения. Интерфейсы таких пакетов могут быть похожими, но все же разными. В результате одни и те же действия пользователя могут привести к различной реакции со стороны каждой из программ или, наоборот, одинаковые запросы от разных программ могут требовать различных действий пользователя. В подобных случаях привычки, выработанные в одном приложении, могут приводить к ошибочным результатам в других приложениях.

Другой человеческой характеристикой, которая привлекает внимание исследователей в области проектирования интерфейса “человек–машина”, является узость человеческого внимания, которое становиться все более сфокусированным при повышении сосредоточенности. По мере того, как человек оказывается все более поглощенным решаемой задачей, разрушить это фокусирование становится все труднее. В 1972 году коммерческий самолет потерпел крушение из-за того, что пилоты настолько погрузились в проблему с шасси (фактически речь идет о замене лампочки индикатора шасси), что позволили самолету врезаться в землю, даже несмотря на то, что в кабине громко звучал сигнал, предупреждающий их об опасности.

Менее критичные примеры часто имеют место в интерфейсах настольных компьютеров. Например, на большинстве клавиатур предусмотрен индикатор “Caps Lock”, указывающий, что клавиатура находится в режиме ввода прописными буквами (т.е. была нажата клавиша <Caps Lock>). Однако, случайно нажав эту клавишу, человек редко замечает изменение состояния соответствующего индикатора: он не реагирует на него, пока на экране не начнут появляться совсем не те символы, которые он ожидает. И даже в этом случае пользователь часто приходит в недоумение, не сразу осознав причину возникновения проблемы. В некотором смысле это не удивительно — подсвеченный индикатор клавиатуры, как правило, оказывается вне основного поля зрения пользователя. Тем не менее пользователи часто не замечают и те индикаторы, которые расположены у них непосредственно на виду. Так, пользователь может настолько увлечься выполнением текущей задачи, что просто не обратит внимания на изменение внешнего вида курсора на экране дисплея, даже если в этой задаче предполагается наблюдение за видом курсора.

Еще одна человеческая черта, которую следует учитывать при проектировании интерфейса, — это ограниченная способность сознания работать с несколькими фактами одновременно. В статье, опубликованной в журнале “Психологический обзор” в 1956 году, Джордж А. Миллер сообщил о проведенных им исследованиях, показавших, что человеческий разум способен удерживать внимание только примерно на семи элементах одновременно. Следовательно, когда требуется принятие решения, очень важно, чтобы интерфейс был спроектирован так, чтобы представлять пользователю всю необходимую информацию, не полагаясь на его память. В частности, следует считать очень плохим решением требовать от пользователя запоминания точных деталей из информации, выведенной на предыдущих экранах. Более того, когда интерфейс требует обширной навигации между многочисленными изображениями, человек может просто потеряться в лабиринте. А это означает, что назначение и расположение изображений на экране является важной проблемой дизайна.

Хотя требования эргономики и когнетики придают некоторые уникальные черты дизайну интерфейса человек-машина, эта область также охватывает многие более традиционные аспекты разработки программного обеспечения. В частности, поиск метрик в области дизайна интерфейса так же важен, как и в более традиционных областях разработки программного обеспечения. Характеристики интерфейса, для которых проводились измерения, включают время, необходимое для изучения интерфейса; время, необходимое для выполнения задачи через интерфейс; уровень ошибок интерфейса пользователя; степень, в которой пользователь сохраняет навыки работы с интерфейсом после

перерывов в работе с ним; и даже такие субъективные черты, как степень, в которой интерфейс нравится пользователям.

Модель GOMS (*Goals, Operators, Methods and Selection* — цели, операторы, методы, правила выбора), впервые представленная в 1954 году, является результатом исследований по отысканию метрик в области проектирования интерфейса “человек–машина”. Основная методология модели заключается в анализе задач с точки зрения целей пользователя (таких, как удаление слова из текста), операторов (таких, как нажатие кнопки мыши), методов (таких, как двойной щелчок кнопкой мыши или нажатие клавиши удаления) и правил выбора (например, выбора между двумя способами достижения одной и той же цели). Все упомянутое, по сути, и является источником аббревиатуры GOMS — цели, операторы, методы и правила выбора. Коротко говоря, GOMS — это методология, позволяющая анализировать действия использующего интерфейс человека, представленные в виде последовательности элементарных этапов (нажать клавишу, переместить мышь, принять решение). Выполнению каждого элементарного этапа назначается точный период времени, и, таким образом, суммируя время, назначенное всем этапам выполнения задачи, методология GOMS предоставляет средство сравнения различных предлагаемых интерфейсов с точки зрения времени, которое потребуется в каждом из них для выполнения одной и той же задачи.

Однако изучение технических деталей таких систем, как GOMS, вовсе не является целью нашего текущего обсуждения. Эта методология упоминается здесь потому, что она основана на особенностях человеческого поведения — движение рук, принятие решений и т.д. На самом деле методология GOMS изначально вообще рассматривалась как тема из области психологии. И это еще раз подчеркивает ту роль, которую особенности человеческого поведения и восприятия играют в области проектирования интерфейса “человек–машина”, даже в таких темах, которые были перенесены из области традиционной разработки программного обеспечения.

Дизайн интерфейсов “человек–машина” обещает оставаться активной областью исследований и в обозримом будущем. Многие проблемы, связанные с современными графическими интерфейсами, все еще не решены, и множество новых сложнейших проблем возникнет при переходе к трехмерным интерфейсам, разработка и использование которых сейчас уже не является далекой перспективой. И действительно, поскольку в этих интерфейсах предполагается объединение аудио- и тактильного взаимодействия с трехмерным изображением, объем потенциальных проблем здесь просто огромен.

### 7.8. Вопросы и упражнения

1. **а.** Приведите примеры возможности применения эргономики в области проектирования интерфейса “человек-машина”.

   **б.** Приведите примеры возможности применения когнетики в области проектирования интерфейса “человек–машина”.

2. Заметным отличием интерфейса “человек–машина” смартфона от интерфейса настольного компьютера являются методы, используемые для прокрутки изображения на экране. В случае настольных компьютеров прокрутка обычно осуществляется перетаскиванием мышью ползунков специальных полос прокрутки, отображаемых справа и внизу в прокручиваем окне, либо с помощью колесика прокрутки, встроенного непосредственно в мышь. С другой стороны, на смартфонах полосы прокрутки чаще всего не используются. (А если они используются, то обычно отображаются в виде тонких линий, предназначенных лишь для указания, какая часть просматриваемого окна видна на экране в данный момент.) Прокрутка в данном случае осуществляется особым жестом “смахивания” — скользящего касания вверх или вниз по экрану дисплея.

   **а.** Какие аргументы могут быть приведены в поддержку этого различия из соображений эргономики?

   **б.** Какие аргументы могут быть приведены в поддержку этого различия из соображений когнетики?

3. Что отличает область проектирования интерфейса “человек–машина” от более традиционной области разработки программного обеспечения?

4. Укажите три характеристики человека, которые следует учитывать при разработке интерфейса “человек–машина”.

## 7.9. Право собственности и ответственность за создаваемое программное обеспечение

Не вызывает сомнения, что компания или отдельный человек должен иметь возможность возмещать затраты и получать доходы от тех инвестиций, которые потребовались для разработки качественного программного обеспечения. В противном случае маловероятно, что многие захотят взять на себя решение

задачи создания программного обеспечения, необходимого нашему обществу. Короче говоря, разработчикам программного обеспечения необходим определенный уровень прав владения на то программное обеспечение, которое они производят.

Юридические усилия по обеспечению такого права собственности подпадают под категорию прав **интеллектуальной собственности**, б*о*льшая часть которого основана на устоявшихся принципах авторского права и патентного права. Действительно, цель авторского права или патента состоит в том, чтобы позволить разработчику "продукта" реализовать этот продукт (или его части) заинтересованным сторонам с необходимой защитой своих прав собственности. По существу, разработчик продукта (будь то физическое или юридическое лицо) будет отстаивать свое право собственности путем включения заявления об авторских правах во все выполненные им работы, включая спецификации требований, проектную документацию, исходный код, план тестирования и конечный продукт — в каком-либо видимом его месте. В уведомлении об авторских правах четко указываются право собственности, лица, которым разрешено использовать произведение, и другие ограничения. Кроме того, права разработчика формально выражены в юридических терминах в **лицензии на программное обеспечение**.

Лицензия на программное обеспечение — это юридическое соглашение между владельцем и пользователем программного продукта, предоставляющее пользователю определенные разрешения на использование продукта без передачи прав собственности на интеллектуальную собственность. В этих соглашениях подробно изложены права и обязанности обеих сторон. Следовательно, важно внимательно прочитать и понять условия лицензии на программное обеспечение — до того, как оно будет установлено и начнется его использование.

Хотя авторские права и лицензионные соглашения на программное обеспечение предоставляют юридические возможности для запрета прямого копирования и несанкционированного использования программного обеспечения, их, как правило, недостаточно, чтобы помешать другой стороне самостоятельно разработать продукт с почти идентичной функцией. Печально, но за прошедшие годы было много случаев, когда разработчик действительно революционного программного продукта так и не смог полностью воспользоваться своим изобретением (два примечательных примера — электронные таблицы и веб-браузеры). В большинстве случаев другой компании удавалось разработать конкурентоспособный продукт, обеспечивавший ей доминирующую долю на рынке. Правовой путь предотвращения такого вторжения со стороны конкурента находится в патентном праве.

Патентные законы были установлены, чтобы позволить изобретателю получить коммерческую выгоду от изобретения. Чтобы получить патент,

изобретатель должен раскрыть подробности изобретения и продемонстрировать, что оно является новым, полезным и неочевидным для других с аналогичным опытом (требование, которое может оказаться достаточно сложным в случае программного обеспечения). Если патент выдан, изобретателю предоставляется право запретить другим лицам создавать, использовать, продавать или импортировать изобретение в течение ограниченного периода времени, который обычно составляет двадцать лет с даты подачи заявки на патент.

Одним из недостатков использования патентов является то, что процесс получения патента является дорогим и трудоемким (часто он растягивается на несколько лет). В течение этого времени программный продукт может устареть, а до выдачи патента заявитель имеет только сомнительные полномочия для предотвращения присвоения продукта другими лицами.

Важность признания авторских прав, лицензий на программное обеспечение и патентов имеет первостепенное значение в процессе разработки программного обеспечения. При разработке программного продукта инженеры-программисты часто решают включить в него программное обеспечение из других продуктов, будь то весь продукт, подмножество его компонентов или даже части исходного кода, загруженные через Интернет. Однако несоблюдение прав интеллектуальной собственности в таких случаях может привести к огромным денежным штрафам и другим последствиям. Например, в 2004 году малоизвестная компания NPT, Inc. успешно выиграла иск против компании Research In Motion (компания RIM — создатель смартфонов марки BlackBerry), выдвинув ей обвинение в нарушении патента на несколько ключевых технологий, встроенных в системы электронной почты RIM. В судебное постановление было включено решение о приостановке работы служб электронной почты для всех пользователей смартфонов BlackBerry в Соединенных Штатах! В конечном итоге компания RIM достигла компромиссного соглашения о выплате компании NPT компенсации на сумму 612,5 млн. долларов США с целью отменить это решение о приостановке.

И наконец, необходимо также рассмотреть вопрос об ответственности. Чтобы защитить себя от ответственности, разработчики программного обеспечения часто включают в лицензии на программное обеспечение отказы от ответственности, в которых указаны те или иные ограничения в отношении их ответственности за созданный продукт. Такие заявления, как “Компания *X* ни в коем случае не будет нести ответственность за любые убытки, возникшие в результате использования этого программного обеспечения”, встречаются очень часто. Однако суды редко признают подобный отказ от ответственности, если истец сможет доказать проявление небрежности со стороны ответчика. Таким образом, дела об ответственности обычно фокусируются на том, проявил ли ответчик уровень тщательности, соответствующий выпущенному им продукту.

Понятно, что уровень тщательности, который будет считаться приемлемым в случае разработки системы обработки текста, может оказаться недостаточным и будет воспринят как недопустимая небрежность в случае разработки программного обеспечения для управления ядерным реактором. Следовательно, одним из лучших способов защиты от претензий по поводу ответственности за программное обеспечение является применение строгих принципов разработки программного обеспечения в процессе разработки создаваемого ПО и обеспечение уровня тщательности разработки, соответствующего области применения создаваемого приложения, а также создание и ведение записей, подтверждающих соблюдение всех этих условий.

### 7.9. Вопросы и упражнения

1. В чем заключается значение уведомления об авторских правах в технических требованиях, проектной документации, исходном коде и конечном продукте?
2. Каким образом закон об авторском праве и патентное право служат на благо обществу?
3. В каких случаях оговорки об отказе от ответственности не принимаются во внимание судами?

## ЗАДАНИЯ ПО МАТЕРИАЛУ ГЛАВЫ

1. Приведите пример, каким образом усилия, затраченные при разработке программного обеспечения, могут окупиться позднее, при сопровождении программы.
2. Что такое пошаговая модель?
3. Объясните, как на технологию разработки программного обеспечения влияет отсутствие метрик для измерения точных характеристик программного обеспечения.
4. Ожидаете ли вы, что метрика измерения сложности программной системы будет кумулятивной в том смысле, что сложность полной системы будет суммой сложностей ее частей? Поясните свой ответ.
5. Ожидаете ли вы, что метрика измерения сложности программной системы будет коммутативной в том смысле, что сложность полной системы

будет одной и той же, если она изначально была разработана с функцией *X*, после чего к ней добавили функцию *Y*, либо если она изначально была разработана с функцией *Y*, после чего к ней добавили функцию *X*? Поясните свой ответ.

**6.** Чем технологии разработки программного обеспечения отличаются от технологий традиционных технических областей, таких как электротехника и машиностроение?

**7.** **а.** В чем заключаются недостатки использования традиционной модели водопада при разработке программного обеспечения?

**б.** В чем заключаются преимущества использования традиционной модели водопада при разработке программного обеспечения?

**8.** Является ли разработка с открытым исходным кодом методологией типа "сверху вниз" или "снизу вверх"? Поясните свой ответ.

**9.** Опишите, как использование констант вместо литералов может упростить процесс модификации программного обеспечения.

**10.** В чем заключается различие между связанностью и связностью модулей? Что следует минимизировать, а что максимизировать и почему?

**11.** Выберите объект из повседневной жизни и проанализируйте его компоненты с точки зрения функциональной или логической связности.

**12.** Сравните связанность между двумя программными единицами, достигаемую с помощью команды `goto`, со связанностью, получаемой при использовании механизма вызова процедур.

**13.** В главе 6 объяснялось, что параметры могут быть переданы в функции по значению или по ссылке. Какой вариант обеспечивает более сложную форму связанности по данным? Поясните свой ответ.

**14.** Какие проблемы могут возникнуть при модификации ПО, если большая программная система была спроектирована таким образом, что все ее элементы данных являются глобальными?

**15.** Что в объектно-ориентированной программе означает объявление переменной экземпляра как открытой или закрытой в отношении связанности по данным? Что может служить основанием для предпочтения объявления переменной экземпляра как закрытой?

**16.** Укажите проблему в отношении связанности по данным, которая может возникнуть в контексте параллельной обработки.

**17.** Ответьте на следующие вопросы, пользуясь приведенной ниже структурной схемой.

**а.** Какому модулю возвращает управление модуль *Y*?

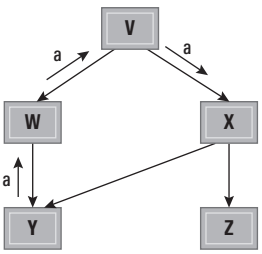
**б.** Какому модулю возвращает управление модуль *Z*?

**в.** Являются ли модули *W* и *X* связанными по управлению?

**г.** Обладают ли модули *W* и *X* связанностью по данным?

**д.** Какие данные совместно используются модулями *W* и *Y*?

**е.** В каких отношениях находятся модули *W* и *Z*?




**18.** Воспользовавшись структурной схемой, представьте процедурную структуру простой системы инвентаризации/учета для небольшого магазина (например, частного магазина сувениров в курортном поселке). Какие модули в вашей системе потребуется модифицировать вследствие внесения изменений в законы о налогах с продаж? Какие модули потребуется изменить, если будет принято решение вести учет клиентов таким образом, чтобы впоследствии им можно было отправлять рекламные материалы по почте?

**19.** Используя диаграмму классов, разработайте объектно-ориентированное решение для предыдущей задачи.

**20.** Начертите простую диаграмму классов, представляющую взаимоотношения между издателями, журналами и подписчиками. Достаточно указать только имя класса в каждом поле, представляющем класс.

**21.** Что такое UML и для чего он используется? Уточните слово, соответствующее в этой аббревиатуре букве “M”.

**22.** Нарисуйте простую диаграмму вариантов использования, представляющую способы, которыми читатель может использовать библиотеку.

**23.** Нарисуйте диаграмму последовательности, отражающую ту серию взаимодействий, которая возникает, когда коммунальная компания отправляет клиенту счет на оплату услуг.

**24.** Нарисуйте простую диаграмму потока данных, отображающую поток данных, который происходит в автоматизированной системе складского учета при продаже изделий.

**25.** Сравните информацию, представленную в диаграмме классов, с информацией, представленной в диаграмме последовательности.

**26.** В чем разница между отношением “один-ко-многим” и отношением “многие-ко-многим”?

**27.** Приведите пример отношения “один-ко-многим”, который не упомянут в этой главе. Приведите пример отношения “многие-ко-многим”, который не упомянут в этой главе.

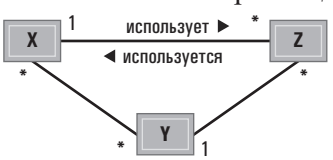
**28.** Исходя из информации, представленной на рис. 7.10, представьте серию взаимодействий, которая может происходить между врачом и пациентом во время их встречи при обращении пациента за консультацией. Нарисуйте диаграмму последовательности, представляющую эти взаимодействия.

**29.** Нарисуйте диаграмму классов, представляющую отношения между посетителями и официантами в ресторане.

**30.** Нарисуйте диаграмму классов, представляющую отношения между журналами, издателями журналов и подписчиками журналов. Для каждого класса укажите необходимый набор переменных экземпляра и методов.

**31.** Расширьте диаграмму последовательности, представленную на рис. 7.5, чтобы показать последовательность взаимодействий, которая произойдет, если класс `PlayerA` успешно вернет мяч классу `PlayerB`, а класс `PlayerB` не сможет успешно отбить этот мяч.

**32.** Ответьте на следующие вопросы, основываясь на прилагаемой диаграмме классов, которая представляет связи между инструментами, их пользователями и их производителями.




**а.** Какие классы (`X`, `Y` и `Z`) представляют на диаграмме инструменты, пользователей и производителей? Обоснуйте свой ответ.

**б.** Может ли инструмент использоваться более чем одним пользователем?

**в.** Может ли инструмент быть изготовлен несколькими производителями?

**г.** Каждый пользователь использует инструменты, изготовленные многими производителями?

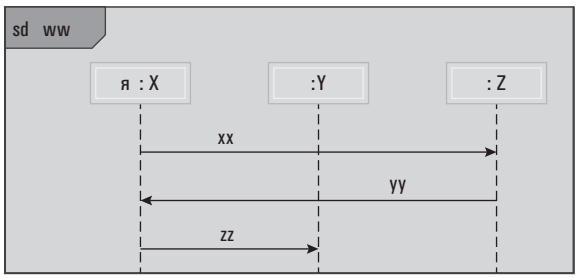
**33.** В каждом из следующих случаев определите, относится ли действие к диаграмме последовательности, диаграмме вариантов использования или диаграмме классов.

**а.** Представляет способ взаимодействия пользователей с системой.

**б.** Представляет отношения между классами в системе.

**в.** Представляет способ взаимодействия объектов для выполнения некоторой задачи.

**34.** Ответьте на следующие вопросы, основываясь на прилагаемой диаграмме последовательности.




**а.** Какой класс содержит метод с именем `ww()`?

**б.** Какой класс содержит метод с именем `xx()`?

**в.** В этой последовательности объект "типа" `Z` когда-либо непосредственно связывается с объектом "типа" `Y`?

**35.** Нарисуйте диаграмму последовательности, показывающую, что объект `A` вызывает метод `bb()` в объекте `B`, объект `B` выполняет запрошенное действие и возвращает управление объекту `A`, а затем объект `A` вызывает метод `cc()` в объекте `B`.

**36.** Расширьте решение предыдущей задачи таким образом, чтобы указать, что объект `A` вызывает метод `bb()` только в том случае, если переменная `continue` имеет значение `true` и продолжает вызывать метод `bb()` до тех пор, пока в переменной `continue` сохраняется значение `true`, и после того, как объект `B` возвращает управление.

**37.** Нарисуйте диаграмму классов, отображающую тот факт, что классы `Truck` и `Automobile` являются обобщениями класса `Vehicle`.

**38.** Исходя из информации, представленной на рис. 7.12, укажите, какие дополнительные переменные экземпляра будут содержаться в объекте "типа" `SurgicalRecord`? А в объекте "типа" `OfficeVisitRecord`?

**39.** Объясните, почему наследование — не всегда лучшее средство реализации обобщений классов.

**40.** Укажите какие-либо шаблоны проектирования в других областях, помимо разработки программного обеспечения.

**41.** Обобщите роль, которую шаблоны проектирования играют в разработке программного обеспечения.

**42.** В какой степени управляющие структуры, представленные в типичном высокоуровневом языке программирования (`if-else`, `while` и т.д.), можно рассматривать как шаблоны проектирования небольшого масштаба?

**43.** Какие из приведенных ниже высказываний соответствуют принципу Парето? Объясните свой ответ.

**а.** Один неприятный человек может испортить вечеринку всем присутствующим.

**б.** Каждая радиостанция концентрируется на определенном формате, таком как хард-рок, классическая музыка или ток-шоу.

**в.** На выборах кандидаты проявят мудрость, если сосредоточат свои кампании на том сегменте электората, который голосовал в прошлом.

**44.** Ожидают ли разработчики программного обеспечения, что большие программные системы будут однородными (либо неоднородными) относительно содержащихся в них ошибок? Поясните свой ответ.

**45.** В чем различие между тестированиями по принципам “черного ящика” и “прозрачного ящика”?

**46.** Приведите некоторые аналогии тестирования по принципам “черного ящика” и “прозрачного ящика” в других областях, отличных от разработки программного обеспечения.

**47.** Чем разработка с открытым исходным кодом отличается от бета-тестирования? (Сравните вариант тестирования по принципу “прозрачного ящика” с тестированием по принципу “черного ящика”.)

**48.** Предположим, что перед окончательным тестированием крупной системы программного обеспечения в нее было намеренно внесено 100 ошибок. Далее допустим, что во время этого тестирования было обнаружено и исправлено 200 ошибок, из которых 50 оказались из группы намеренно внесенных в систему. Если исправить оставшиеся 50 известных ошибок, сколько невыявленных ошибок, по-вашему, еще останется в системе? Объясните, почему.

**49.** Что такое GOMS?

**50.** Что такое эргономика? Что такое когнетика?

**51.** Одно из различий между интерфейсами “человек–компьютер” смартфона и настольного компьютера заключается в методике, используемой для изменения масштаба изображения на экране с целью получения более или менее детального представления данных (процесс, называемый масштабированием). На рабочем столе масштабирование обычно достигается путем перетаскивания ползунка, отдельного от отображаемой области, или с помощью элемента меню или панели инструментов. На смартфоне масштабирование выполняется путем одновременного касания экрана дисплея большим и указательным пальцами, а затем изменения расстояния

между двумя точками касания (процесс, называемый растягиванием для увеличения масштаба или стягиванием — для его уменьшения).

**а.** Исходя из положений эргономики, какие аргументы могут быть сделаны в поддержку этого различия?

**б.** Исходя из положений когнетики, какие аргументы могут быть сделаны в поддержку этого различия?

**52.** В каких случаях существующие законы о защите авторских прав не смогут защитить инвестиции разработчиков программного обеспечения?

**53.** В каких случаях разработчику программного обеспечения может быть отказано в получении патента?

## ОБЩЕСТВЕННЫЕ И СОЦИАЛЬНЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники, а также в ваших собственных воззрениях и тех принципах, на которых они основаны. Задача не сводится к тому, чтобы просто дать ответы на предложенные вопросы. Вы должны также разобраться, почему вы ответили именно так, а не иначе, и насколько ваши суждения по различным вопросам согласуются между собой.

**1. а.** Исполнителю была поставлена задача разработать систему для занесения медицинских записей в машину, соединенную с большой сетью. Его предложения относительно требований безопасности были отклонены по финансовым соображениям, и исполнителю было приказано продолжить работу над проектом, применяя систему безопасности, которую он считал недостаточной. Что ему следует делать и почему?

**б.** Предположим, что упомянутый выше исполнитель разработал систему так, как ему было приказано, и теперь опасается, что медицинские записи могут подвергаться несанкционированному просмотру. Что ему следует делать? В какой степени он несет ответственность за возможные нарушения безопасности?

**в.** Допустим, вместо того чтобы подчиниться руководству, исполнитель отказывается от работы с системой и поднимает шум, разглашая недостатки проекта, что приводит к финансовым затруднениям в компании и потере работы многими ни в чем не повинными сотрудниками. Корректны ли действия этого исполнителя? Что если, будучи всего лишь одним из рядовых членов группы разработчиков, он не знал о том, что

в другом подразделении компании предпринимались усилия по разработке действенной системы безопасности, которую предполагалось применить и к его системе. Как это изменит ваше отношение к действиям данного исполнителя? (Помните, что взгляд исполнителя на ситуацию остается таким же, как и раньше.)

2. Как должна распределяться ответственность, если крупная система программного обеспечения разрабатывается многими людьми? Существует ли иерархия ответственности? Существуют ли степени юридической ответственности?
3. Как мы знаем, крупные и сложные системы программного обеспечения часто разрабатываются многими людьми, однако лишь некоторые из них имеют полное представление о существе проекта. Допустимо ли работнику принимать участие в проекте, не имея полных знаний о его назначении и свойствах?
4. До какой степени каждый несет ответственность за то, как его достижения в конечном счете применяются другими людьми?
5. С точки зрения отношений между специалистом в области компьютеров и его клиентом должен ли первый просто реализовывать желания клиента или же скорее направлять его желания? Что если специалист предвидит, что требования клиента могут привести к нежелательным последствиям? Например, клиент может пожелать внести в систему некоторые упрощения для повышения ее эффективности, но специалист может предвидеть, что такие упрощения могут стать потенциальным источником появления ошибочных результатов или злоупотреблений в системе. Если клиент все же настаивает на своем решении, может ли специалист считать себя свободным от ответственности?
6. Что произойдет, если технологии начнут развиваться так быстро, что новые изобретения будут вытеснены новейшими еще до того, как изобретатели успеют извлечь выгоду из своего изобретения? Нужна ли прибыль для мотивации изобретателей? Как с вашим ответом соотнести возможный успех разработки с открытым исходным кодом? Является ли качественное бесплатное программное обеспечение устойчивым элементом реальности?
7. Внесла ли компьютерная революция заметный вклад или хотя бы оказала определенную помощь в решении энергетических проблем в мире? А как насчет других масштабных проблем, таких как голод или бедность?
8. Будут ли достижения в технологии продолжаться бесконечно? В любом случае что могло бы изменить зависимость общества от технологий? Что

может оказаться конечным итогом развития общества, бесконечно продолжающего продвижение технологии?

9. Если бы у вас была машина времени, в каком историческом периоде вы хотели бы жить? Есть ли современные технологии, которые вы хотели бы взять с собой? Можно ли отделить одну технологию от другой? Реально ли протестовать против глобального потепления, но принимать современный уровень медицины и применяемые ею методы лечения?
10. Многие приложения на смартфоне автоматически интегрируются с услугами, предоставляемыми другими приложениями. Эта интеграция может делиться информацией, введенной в одно приложение, с другим. Каковы преимущества этой интеграции? Возможны ли какие-либо проблемы в связи со “слишком большой” интеграцией?

## РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА


1. Alexander C., Ishikawa S., Silverstein M. *A Pattern Language*. — New York: Oxford University Press, 1977.
2. Beck K. *Extreme Programming Explained: Embrace Change*, 2nd ed. — Boston, MA: Addison-Wesley, 2004.
3. Bowman D.A., Kruijff E., LaViola J.J., Jr., Poupyrev I. *3D User Interfaces Theory and Practice*. — Boston, MA: Addison-Wesley, 2005.
4. Braude E. *Software Design: From Programming to Architecture.* — New York: Wiley, 2004.
5. Bruegge B., Dutoit A. *Object-Oriented Software Engineering Using UML, Patterns, and Java*, 3rd ed. — Boston, MA: Addison-Wesley, 2010.
6. Cockburn A. *Agile Software Development: The Cooperative Game*, 2nd ed. — Boston, MA: Addison-Wesley, 2006.
7. Fox C. *Introduction to Software Engineering Design: Processes, Principles and Patterns with UML2*. — Boston, MA: Addison-Wesley, 2007.
8. Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. — Boston, MA: Addison-Wesley, 1995.
9. Maurer P.M. *Component-Level Programming*. — Upper Saddle River, NJ: Prentice-Hall, 2003.
10. Pfleeger S.L., Atlee J.M. *Software Engineering: Theory and Practice*, 4th ed. — Upper Saddle River, NJ: Prentice-Hall, 2010.
11. Pilone D., Pitman N. *UML 2.0 in a Nutshell*. — Cambridge, MA: O’Reilly Media, 2005.
12. Pressman R.S., Maxim B. *Software Engineering: A Practitioner’s Approach*, 8th ed. — New York: McGraw-Hill, 2014.

**13.** Schach S.R. *Classical and Object-Oriented Software Engineering*, 8th ed. — New York: McGraw-Hill, 2010.
**14.** Shalloway A., Trott J.R. *Design Patterns Explained*, 2nd ed. — Boston, MA: Addison-Wesley, 2005. (Имеется русский перевод первого издания этой книги: Шаллоуей А., Тротт Дж.Р. *Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию.* — М.: Издательский дом "Вильямс", 2002.)
**15.** Shneiderman B., Plaisant C., Cohen M., Jacobs S., Elmqvist N., Diakopoulos N. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 5th ed. — Boston, MA: Addison-Wesley, 2009.
**16.** Sommerville I. *Software Engineering*, 9th ed. — Boston, MA: Addison-Wesley, 2010. (Имеется русский перевод 6-го издания этой книги: Соммервил Я. *Инженерия программного обеспечения*. 6-е изд. — М.: Издательский дом "Вильямс", 2002.)

## ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

**1.** Элиенс А. *Принципы разработки объектно-ориентированных программ*, 2-е изд. — М.: Издательский дом "Вильямс", 2001.
**2.** Астелс Д., Миллер Г., Новак М. *Практическое руководство по экстремальному программированию*. — М.: Издательский дом "Вильямс", 2002.
**3.** Скотт К. *UML. Основные концепции*. — М.: Издательский дом "Вильямс", 2002.
**4.** Леффингуэлл Д., Уидриг Д. *Принципы работы с требованиями к программному обеспечению. Унифицированный подход*. — М.: Издательский дом "Вильямс", 2002.
**5.** Тамре Л. *Введение в тестирование программного обеспечения*. — М.: Издательский дом "Вильямс", 2003.
**6.** Грехэм Я. *Объектно-ориентированные методы*. Принципы и практика, 3-е изд. — М.: Издательский дом "Вильямс", 2004.
**7.** Ларман К. *Применение UML 2.0 и шаблонов проектирования*, 3-е изд. — М.: Издательский дом "Вильямс", 2006.
**8.** Майерс Г., Баджетт Т., Сандлер К. *Искусство тестирования программ*, 3-е изд. — М.: ООО "Вильямс", 2012.
**9.** Смит Дж. *Элементарные шаблоны проектирования*. — М.: ООО "Вильямс", 2012.
**10.** Мартин Р.С. *Гибкая разработка программ на Java и C++: принципы, паттерны и методики.* — М.: ООО "Вильямс", 2017.