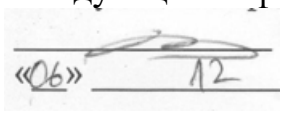
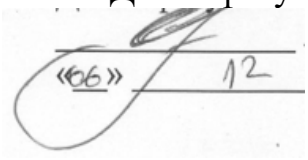
Учреждение образования
«Брестский государственный технический университет»

Факультет электронно-информационных систем
Кафедра «Интеллектуальных информационных технологий»

СОГЛАСОВАНО
Заведующий кафедрой

«06» 12 В.А.Головко
2024 г.

СОГЛАСОВАНО
Декан факультета

«06» 12 А.Н.Парфиевич
2024 г.

ЭЛЕКТРОННЫЙ УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС
ПО УЧЕБНОЙ ДИСЦИПЛИНЕ

«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ»

для специальности (направления специальности)

6-05-0612-03 Системы управления информацией

Составители: Михняев Андрей Леонидович, старший преподаватель
Яцук Татьяна Александровна, преподаватель-стажер

Рассмотрено и утверждено на заседании Научно-методического совета
27.12.2024 г., протокол № 2.

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Дисциплина «Объектно-ориентированное программирование» относится к государственному компоненту модуля «Программирование».

Подготовка современного специалиста требует уверенного владения возможностями, предоставляемыми компьютерными технологиями. Изучение настоящей дисциплины обеспечивает подготовку специалиста, владеющего фундаментальными знаниями и практическими навыками в области объектно-ориентированного анализа, программирования и элементов проектирования при решении практических задач.

Целью изучения курса является: углубленное обучение студентов технологическим основам и практическим навыкам проектирования, реализации и сопровождения больших программных систем современных ЭВМ на основе технологии объектно-ориентированного программирования.

Задачи изучения дисциплины:

- формирование представления об объектной технологии как примере использования системного подхода и реализации результатов системного анализа;
- приобретение знаний о возможностях, методах, моделях и средствах поддержки современных промышленных информационных технологий;
- приобретение навыков практической работы со средствами обеспечения жизненного цикла создания и эволюционного развития сложных программных систем.

В результате изучения учебной дисциплины «Объектно-ориентированное программирование» формируются следующие компетенции:

БПК-10. Использовать принципы объектно-ориентированного программирования для компьютерного моделирования реальных и концептуальных систем.

В результате изучения учебной дисциплины студент должен:

знать:

- принципы объектно-ориентированного программирования;
- способы реализации отношений между классами;
- использование свойств полиморфизма, наследования и инкапсуляции;
- возможности и ограничения абстрактных классов, интерфейсов и шаблонов;

уметь:

- создавать структурированные программы на основе объектных технологий в среде современных систем объектно-ориентированного проектирования;
- переходить из одной объектно-ориентированной платформы на другую;
- использовать возможности языка UML для представления проектных решений;

владеть:

- методами и инструментальными средствами и системами разработки объектно-ориентированных программ;

– техникой создания объектно-ориентированных программных компонент и организацией их взаимодействия в программных проектах.

Дисциплина базируется на знаниях, полученных студентами при изучении курсов «Информационные системы и технологии», «Архитектура ЭВМ», «Построение и анализ алгоритмов», «Основы алгоритмизации и программирования».

Структура электронного учебно-методического комплекса по дисциплине «Объектно-ориентированное программирование»

Теоретический раздел ЭУМК содержит материалы для теоретического изучения учебной дисциплины и представлен конспектом лекций. Практический раздел ЭУМК.

Практический раздел ЭУМК содержит материалы: для проведения лабораторных учебных занятий в виде методических указаний для выполнения лабораторных работ.

Раздел контроля знаний ЭУМК содержит материалы для итоговой аттестации (экзаменационные вопросы), позволяющие определить соответствие результатов учебной деятельности обучающихся требованиям образовательных стандартов высшего образования и учебно-программной документации образовательных программ высшего образования.

Вспомогательный раздел включает учебную программу учреждения высшего образования по учебной дисциплине «Объектно-ориентированное программирование».

Рекомендации по организации работы с ЭУМК:

- лекции проводятся с использованием представленных в ЭУМК учебных теоретических материалов; при подготовке к экзамену, лабораторным занятиям студенты могут использовать теоретический материал, представленный в данном ЭУМК;

- лабораторные занятия проводятся в компьютерной лаборатории с использованием: представленных в ЭУМК методических указаний; инструментальных программных средств – среду программирования (C++, Java, Python), MS Word;

- курсовое проектирование осуществляется на базе представленных в ЭУМК методических указаний, с использованием ЭВМ;

- экзаменационные материалы приведены в разделе контроля знаний.

ОГЛАВЛЕНИЕ

1 ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

ОГЛАВЛЕНИЕ

1 ВВЕДЕНИЕ В ООП

1.1 ИСТОРИЯ РАЗВИТИЯ

Хотя многие программисты не осознают этого, объектно-ориентированная разработка программного обеспечения существует с начала 1960-х годов. Только во второй половине 1990-х годов объектно-ориентированная парадигма начала набирать обороты, несмотря на тот факт, что популярные объектно-ориентированные языки программирования вроде Smalltalk и C++ уже широко использовались.

Расцвет объектно-ориентированных технологий совпал с началом применения Интернета в качестве платформы для бизнеса и развлечений. А после того, как стало очевидным, что Глобальная сеть активно проникает в жизнь людей, объектно-ориентированные технологии уже заняли удобную позицию для того, чтобы помочь в разработке новых веб-технологий.

Основа ООП была заложена в начале 1960-х годов. Прорыв в использовании экземпляров и объектов был достигнут в MIT с PDP-1, и первым языком программирования для работы с объектами стал *Simula 67*. Он был разработан *Кристен Найгаард* и *Оле-Джохан* Даль в Норвегии с целью создания симуляторов. Они работали над симуляциями взрыва кораблей и поняли, что могут сгруппировать корабли в различные категории. Каждому типу судна было решено присвоить свой собственный класс, который должен содержать в себе набор уникальных характеристик и данных. Таким образом, *Simula* не только ввела понятие класса, но и представила рабочую модель.

Термин «объектно-ориентированное программирование» был впервые использован компанией *Xerox PARC Аланом Кеем* и некоторыми другими учеными в языке программирования *Smalltalk*. Понятие ООП использовалось для обозначения процесса использования объектов в качестве основы для расчетов. Команда разработчиков была вдохновлена проектом Simula 67, но они спроектировали свой язык так, чтобы он был динамичным. В Smalltalk объекты могут быть изменены, созданы или удалены, что отличает его от статических систем, которые обычно используются. Этот язык программирования также был первым, использовавшим концепцию наследования. Именно эта особенность позволила Smalltalk превзойти как Simula 67, так и аналоговые системы программирования.

Simula 67 стала новаторской системой, которая впоследствии стала основой для создания большого количества других языков программирования, в том числе Pascal и Lisp. В 1980-х годах объектно-ориентированное программирование приобрело огромную популярность, и основным фактором в этом стало появление языка C++. Концепция ООП также имела важное значение для разработки графических пользовательских интерфейсов. В качестве одного из самых ярких примеров можно привести структуру Cocoa, существующую в Mac OS X.

Общие принципы модели стали применяться во многих современных языках программирования. Некоторые из них – Fortran, BASIC, Pascal. На тот момент многие программы не были разработаны с учетом ООП, что было причиной возникновения некоторых проблем совместимости. «Чистые» объектно-ориентированные языки

программирования не обладали многими функциями, необходимыми программистам. Для решения этих проблем ряд исследователей предложили несколько новых языков программирования, созданных на основе принципов ООП с сохранением других, необходимых программистам, функций. Среди наиболее ярких примеров можно выделить Eiffel, Java, .NET. Даже в серьезных веб-разработках используются языки программирования, основанные на принципах ООП - PHP, Python, Ruby. По мнению экспертов, в ближайшие несколько десятилетий именно объектно-ориентированный подход будет оставаться основной парадигмой в развитии программирования.

1.2 ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ В СРАВНЕНИИ С ОБЪЕКТНО-ОРИЕНТИРОВАННЫМ

Прежде чем мы углубимся в преимущества объектно-ориентированной разработки, рассмотрим более существенный вопрос: что именно такое объект? Это одновременно и сложный, и простой вопрос. Сложный он потому, что изучение любого метода разработки программного обеспечения не является тривиальным. А простой он в силу того, что люди уже мыслят объектно. Например, когда вы смотрите на какого-то человека, вы видите его как объект. При этом объект определяется двумя компонентами: *атрибутами* и *поведением*. У человека имеются такие атрибуты, как цвет глаз, возраст, вес и т. д. Человек также обладает поведением, то есть он ходит, говорит, дышит и т. д. В соответствии со своим базовым определением ***объект*** *– это сущность, одновременно содержащая данные и поведения*. Слово «одновременно» в данном случае определяет ключевую разницу между объектно-ориентированным программированием и другими методологиями программирования. Например, при процедурном программировании код размещается в полностью отдельных функциях или процедурах. В идеале, как показано на рис. 1.1, эти процедуры затем превращаются в «черные ящики», куда поступают входные данные и откуда потом выводятся выходные данные. Данные размещаются в отдельных структурах, а манипуляции с ними осуществляются с помощью этих функций или процедур.

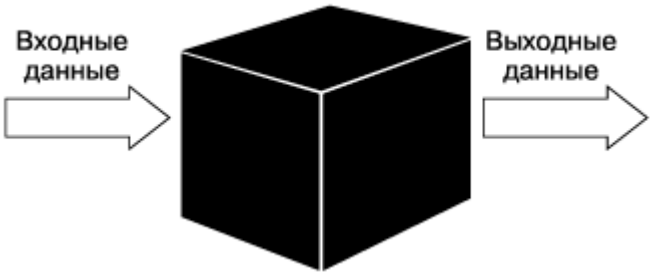


Рисунок 1.1. *Черный ящик*

При объектно-ориентированном проектировании атрибуты и поведения размещаются в рамках одного объекта, в то время как при процедурном или структурном проектировании атрибуты и поведения обычно разделяются.

Как показано на рис. 1.2, при структурном программировании данные зачастую отделяются от процедур и являются глобальными, благодаря чему их легко

модифицировать вне области видимости вашего кода. Это означает, что доступ к данным неконтролируемый и непредсказуемый (то есть у множества функций может быть доступ к глобальным данным). Во-вторых, поскольку у вас нет контроля над тем, кто сможет получить доступ к данным, тестирование и отладка намного усложняются. При работе с объектами эта проблема решается путем объединения данных и поведения в рамках одного элегантного полного пакета.

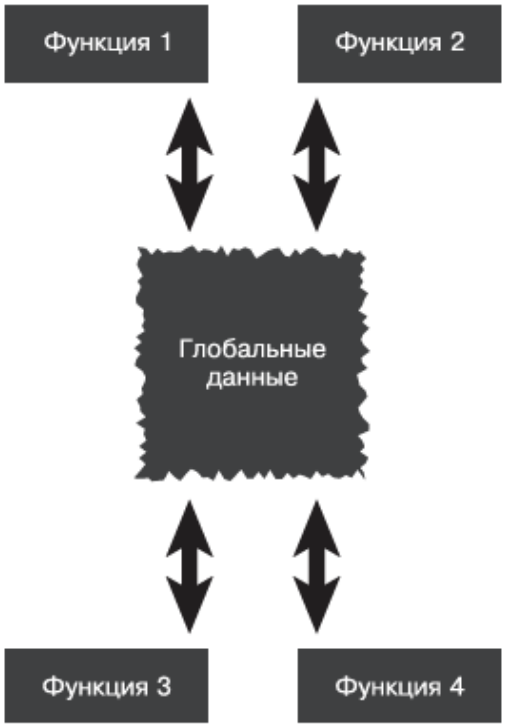


Рисунок 1.2. *Использование глобальных данных*

1.3 ОТ СТРУКТУР И ПОДПРОГРАММ К ОБЪЕКТАМ

Целями структурного программирования являлись структуризация данных и декомпозиция кода. Объединение данных в структуры позволяло, с одной стороны, более естественно описывать сущности реального мира, имеющие самые разнотипные наборы атрибутов. А, с другой стороны, делало процесс разработки программ более простым, поскольку однородные данные были сгруппированы в одном месте и под одним общим именем.

Декомпозиция кода заключалась в разделении исходного кода программы на отдельные подпрограммы. Подпрограммы снижали количество возможных связей между отдельными операторами (локализация кода) и, кроме того, позволяли устанавливать необходимую область видимости для переменных, используемых в подпрограммах (локализация данных). Локализация кода не позволяла произвольно переходить от одного оператора в одной подпрограмме к любому оператору в другой подпрограмме. Вызов подпрограммы приводил к передаче управления только в определенную точку входа, хотя некоторые языки допускали более одной точки входа в подпрограмму.

Такое решение не только повышало надежность программ, но и позволяло многократно использовать одни и те же подпрограммы в различных программах. Как следствие, стали появляться библиотеки подпрограмм, как самостоятельные программные продукты. Часть библиотек поставлялась вместе с компиляторами с языков программирования и операционными системами, другая часть распространялась свободно или на коммерческой основе. Применение библиотек существенно ускоряло процесс разработки программного обеспечения, так как подпрограммы, применяемые многократно, требовалось отлаживать только единожды.

Локализация данных имела ничуть не меньшее значение, чем локализация кода. Три вида данных в подпрограмме:

- *Локальные данные подпрограммы, которые автоматически создавались при ее вызове и уничтожались при возврате из подпрограммы;*
- *Локальные данные, сохраняемые между несколькими вызовами одной подпрограммы;*
- *Глобальные данные, доступные нескольким или всем подпрограммам, и доступ к которым осуществлялся прямо из подпрограммы;*
- *Фактические параметры, передаваемые подпрограммам, замещающие объявленные формальные параметры.*

Глобальные данные применялись в основном для организации связей между подпрограммами, но они существенно ограничивали гибкость подпрограмм. Действительно, если подпрограмма работала с какими-то глобальными данными, то эти данные должны были совпадать по имени, типу во всех программах, которые использовали данную подпрограмму.

В отличие от этого, локальные данные, наоборот, позволяли подпрограмме быть независимой от программ, использующих ее. Но они не обеспечивали передачу данных между различными подпрограммами. Для передачи данных или ссылок на них использовался механизм формальных/фактических параметров. При описании подпрограммы объявляются формальные параметры, которые она принимает при вызове. Когда реально происходит вызов подпрограммы, на место формальных параметров подставляются фактические параметры того же типа.

Образование библиотек подпрограмм происходило, как правило, по функциональному признаку. То есть, в одну библиотеку помещались подпрограммы, выполняющие близкую по составу работу. Например, библиотека подпрограмм, работающих с каким-то устройством. Библиотеки имели существенный недостаток. Результаты их работы могли быть неверны при подаче на вход не корректных данных. Однако данные располагались в программе за пределами библиотеки и могли меняться произвольно. Появилась реальная потребность объединить данные и код, который их обрабатывает. Эта задача была решена в модульном программировании. Модуль, в отличие от библиотеки подпрограмм, мог содержать не только подпрограммы, но и данные.

Данные, располагаемые в модуле, были глобальными и для модуля, и для программы, использующей модуль. Для увеличения безопасности данных модуль

разделили на несколько зон. Одна из зон обеспечивала взаимодействие между модулем и внешним программные обеспечением. Это так называемая зона интерфейса. Другая зона отвечала за реализацию модуля и была недоступна для внешнего программного обеспечения.

Соответственно, критичные к произвольным изменениям данные помещались в зону реализации и были недоступны для программы и других модулей. Про эти данные можно сказать, что они инкапсулированы модулем.

В интерфейсной зоне оставались, как правило, только те данные, с которыми активно взаимодействовало внешнее программное обеспечение. Нельзя сказать, что эти данные можно было произвольно изменять без ущерба для работоспособности программы. Практически все данные были инкапсулированы, то есть помещены в зону реализации, а для взаимодействия с ними в интерфейсную зону помещались объявления специальных интерфейсных программ. Только они могли изменять значения переменных внутри модуля или возвращать значения этих переменных.

Перенос данных в область реализации поставил еще одну проблему: данные необходимо было инициализировать перед началом работы и освобождать перед окончанием работы модуля. Например, перед началом работы модуля могла потребоваться динамически распределяемая область памяти, а в конце требовалось вернуть эту область памяти системе. Это привело к появлению в модулях областей инициализации, которые вызывались до начала работы модуля, и областей, которые автоматически вызывались перед тем, как модуль завершит работу.

Если теперь сложить все элементы полученной мозаики воедино, то получится знакомая картина: все подпрограммы модуля собраны вместе по функциональному признаку и работают над одними данными; модуль имеет области кода, которые вызываются при инициализации и окончании работы модуля. Такой модуль можно назвать прообразом объекта.

Мы вплотную подошли к возможности реализации объектной технологии. Фактически осталось только ввести поддержку полиморфизма и наследования, что сегодня, как правило, реализуется в рамках компиляторов. Важно понимать, что объектная технология, являясь прямым продолжением технологии структурной разработки программ, открыла новые принципиально иные подходы к проектированию сложного программного обеспечения.

1.4 ПРЕИМУЩЕСТВА ИСПОЛЬЗОВАНИЯ ООП

Основным достоинством ООП является то, что данная концепция позволяет значительно ускорить разработку новых программ и приложений, разделив общий объем работы между несколькими независимыми программистами или группами сотрудников.

Код строится таким образом, что его отдельные логические блоки работают изолированно друг от друга и не могут помешать выполнению других функций.

Среди ключевых аргументов в пользу использования объектно-ориентированного программирования выделяют:

- Программы, написанные с применением языков ООП, действительно *легко понять*.
- Поскольку *все рассматривается как объекты*, объектно-ориентированные языки позволяют смоделировать концепцию реального мира.
- Подход ООП предлагает возможность *многократного использования классов*. Вы можете повторно использовать уже созданные классы вместо того, чтобы записывать их снова.
- Кроме того, концепция ООП *допускает параллельную разработку* и использование нескольких классов. Больше усилий прилагается к объектно-ориентированному анализу и проектированию, что также снижает общие затраты на разработку ПО.
- *Удобство тестирования и обслуживания.* Поскольку конструкция кода является модульной, часть системы может быть обновлена в случае возникновения проблем без необходимости внесения масштабных изменений.
- Существующий код *легко поддерживать и менять*, поскольку новые объекты могут создаваться с небольшими отличиями от существующих.

Объектно-ориентированные языки программирования поставляются с богатыми библиотеками объектов, а код, разработанный в ходе реализации проекта, также может быть повторно использован в будущем при создании других программ.

Используя готовые библиотеки, вы можете еще больше ускорить процесс разработки, адаптируя и модифицируя для своих проектов уже существующие рабочие решения.

Это особенно полезно при разработке графических интерфейсов пользователя.

Поскольку объектные библиотеки содержат много полезных функций, разработчикам программного обеспечения не нужно «изобретать велосипед» так часто, что дает возможность максимально сосредоточиться на создании новой программы.

2 КЛАССЫ И ОБЪЕКТЫ

Исторически сложилось так, что объектно-ориентированные языки определяются следующими концепциями: *инкапсуляцией*, *наследованием* и *полиморфизмом*. Поэтому если тот или иной язык программирования не реализует все эти концепции, то он, как правило, не считается объектно-ориентированным.

Основное преимущество объектно-ориентированного программирования заключается в том, что и данные, и операции (код), используемые для манипулирования ими, инкапсулируются в одном объекте. Например, при перемещении объекта по сети он передается целиком, включая данные и поведение.

Объекты – это строительные блоки объектно-ориентированных программ. Та или иная программа, которая задействует объектно-ориентированную технологию, по сути является набором объектов. В качестве наглядного примера рассмотрим корпоративную

систему, содержащую объекты, которые представляют собой работников соответствующей компании. Каждый из этих объектов состоит из данных и поведений.

2.1 Данные объектов

Данные, содержащиеся в объекте, представляют его состояние. В терминологии объектно-ориентированного программирования эти данные называются *атрибутами*. В нашем примере, как показано на рис. 2.1 атрибутами работника могут быть номер социального страхования, дата рождения, пол, номер телефона и т. д. Атрибуты включают информацию, которая разнится от одного объекта к другому (ими в данном случае являются работники).

Рисунок 2.1 *Атрибуты объекта* `Employee`

2.2 Поведения объектов

Поведение объекта представляет то, что он может сделать. В процедурных языках поведение определяется процедурами, функциями и подпрограммами. В терминологии объектно-ориентированного программирования поведения объектов содержатся в методах, а вызов метода осуществляется путем отправки ему сообщения. Примите по внимание, что в нашем примере с работниками одно из необходимых поведений объекта `Employee` заключается в задании и возврате значений различных атрибутов. Таким образом, у каждого атрибута будут иметься соответствующие методы, например `setGender()` и `getGender()`. В данном случае, когда другому объекту потребуется такая информация, он сможет отправить сообщение объекту `Employee` и узнать значение его атрибута `gender`. Помните, что одно из самых интересных преимуществ использования объектов заключается в том, что данные являются частью пакета — они не отделяются от кода.

2.3 Что такое класс

Если говорить просто, то *класс* – это «чертеж» объекта. При создании экземпляра объекта вы станете использовать класс как основу для того, как этот объект будет создаваться. Фактически попытка объяснить классы и объекты подобна стремлению

решить дилемму «что было раньше – курица или яйцо?». Трудно описать класс без использования термина «объект» и наоборот. Например, велосипед определенного человека – это объект. Однако для того, чтобы построить этот велосипед, кому-то сначала пришлось подготовить чертежи (то есть класс), по которым он затем был изготовлен.

В случае с объектно-ориентированным программным обеспечением, в отличие от дилеммы «что было раньше – курица или яйцо?», мы знаем, что первым был именно класс. Нельзя создать экземпляр объекта без класса.

Для объяснения классов и методов целесообразно использовать пример из сферы реляционных баз данных. Если говорить о таблице базы данных, то определением этой таблицы как таковой (полей, описания и использованных типов данных) был бы класс (метаданные), а объектами выступали бы строки таблицы (данные).

2.4 Создание объектов

Классы можно представлять себе как шаблоны или формочки для печенья для объектов, как показано на рис. 1.2. Класс используется для создания объекта.

Класс можно представлять себе как нечто вроде типа данных более высокого уровня. Например, точно таким же путем, каким вы создаете то, что относится к типу данных `int` или `float`:

```
int x; float y;
```

вы можете создать объект с использованием предопределенного класса:

```
myClass myObject;
```

В этом примере сами имена явно свидетельствуют о том, что `myClass` является классом, а `myObject` – объектом. Помните, что каждый объект содержит собственные атрибуты (данные) и поведения (функции или программы). Класс определяет атрибуты и поведения, которые будут принадлежать всем объектам, созданным с использованием этого класса. Классы – это фрагменты кода. Объекты, экземпляры которых созданы на основе классов, можно распространять по отдельности либо как часть библиотеки. Объекты создаются на основе классов, поэтому классы должны определять базовые строительные блоки объектов (атрибуты, поведения и сообщения). В общем, вам потребуется спроектировать класс прежде, чем вы сможете создать объект.

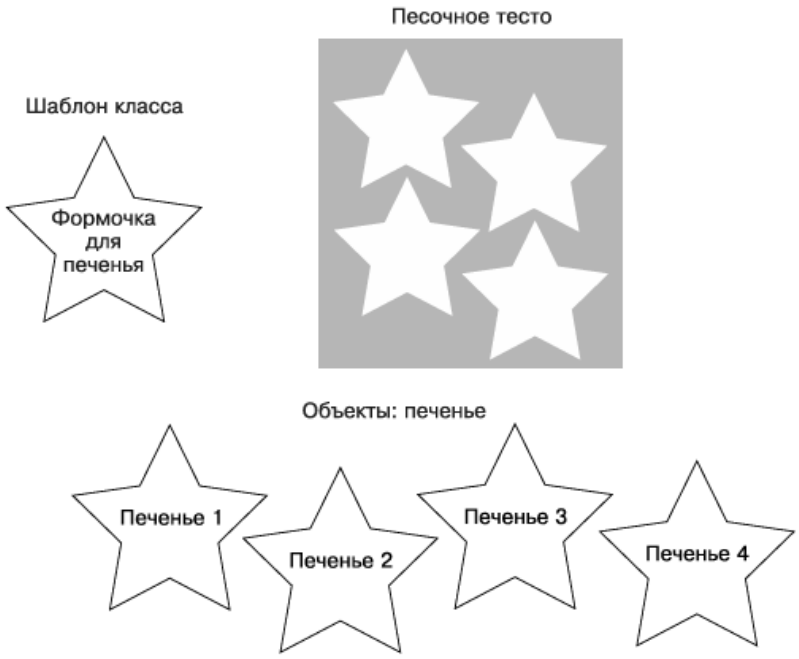


Рисунок 2.2 *Шаблон класса*

2.5 КЛАСС против ОБЪЕКТА

Основные различия между классом и объектом можно выразить в следующей таблице.

Класс	**Объект**
Класс – это шаблон для создания объектов в программе.	Объект является экземпляром класса.
Класс – это логическая сущность	Объект – физическое лицо
Класс не выделяет пространство памяти при создании.	Объект выделяет пространство памяти всякий раз, когда он создается.
Вы можете объявить класс только один раз. Пример: Автомобиль.	Используя класс, вы можете создать более одного объекта. ример: Jaguar, BMW, Tesla и т. д.
Класс генерирует объекты	Объекты обеспечивают жизнь классу.
Классами нельзя манипулировать, поскольку они недоступны в памяти.	Ими можно манипулировать.
У него нет значений, связанных с полями.	Каждый объект имеет свои собственные значения, связанные с полями.
Вы можете создать класс, используя ключевое слово «`class`».	Вы можете создать объект, используя ключевое слово «`new`» в Java

2. 6 Пример использования концепций классов и объектов

Давайте рассмотрим пример разработки системы управления домашними животными, специально предназначенной для собак. Вам понадобится различная информация о собаках, например, разные породы собак, возраст, размер и т. д.

Вам необходимо смоделировать реальных существ, например, собак, в программные объекты.

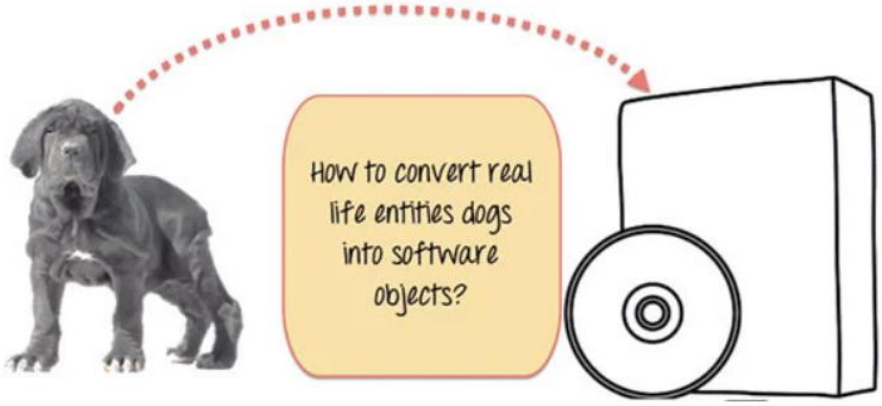


Рисунок 2.3

Каждая собака обладает некоторыми общими характеристиками, к ним можно отнести: порода, возраст, размер, цвет и т.д. – все эти характеристики могут формировать элементы данных для вашего объекта.

Рисунок 2.4 *Характеристики собак*

Помимо этого, у каждой собаки есть характерное поведение, например, возможность спать, сидеть, есть и т. д. Итак, это будут действия наших программных объектов.

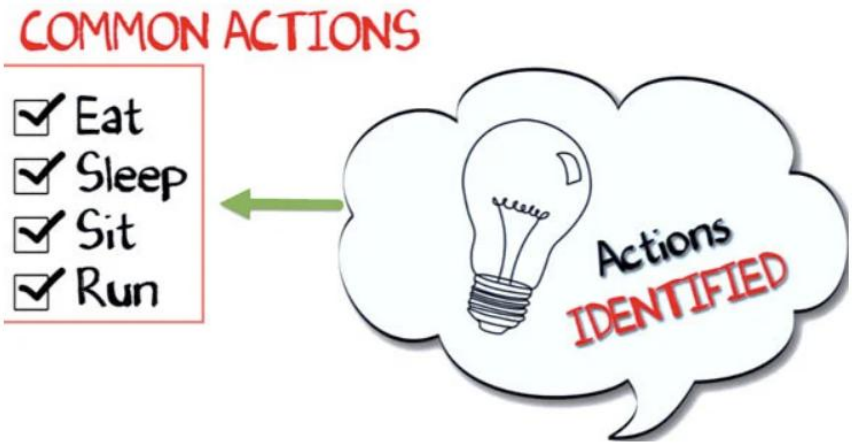


Рисунок 2.5 *Поведение собак*

До сих пор мы определили следующие вещи:

- **Класс**: Собаки
- **Члены данных объекта**: размер, возраст, окрас, порода и т. д.
- **методы**: есть, спать, сидеть и бегать.

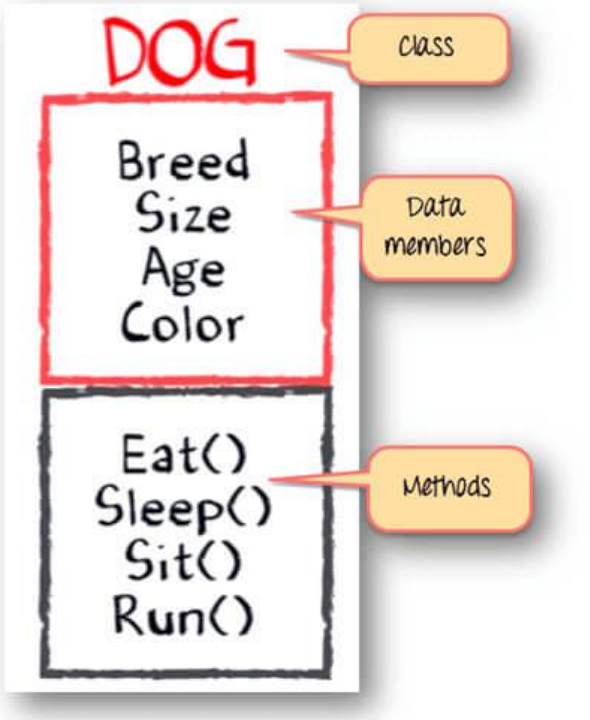


Рисунок 2.6 *Класс Dog*

Теперь для разных значений элементов данных (размера породы, возраста и окраса) в `Java` классе, вы получите разные предметы для собак.

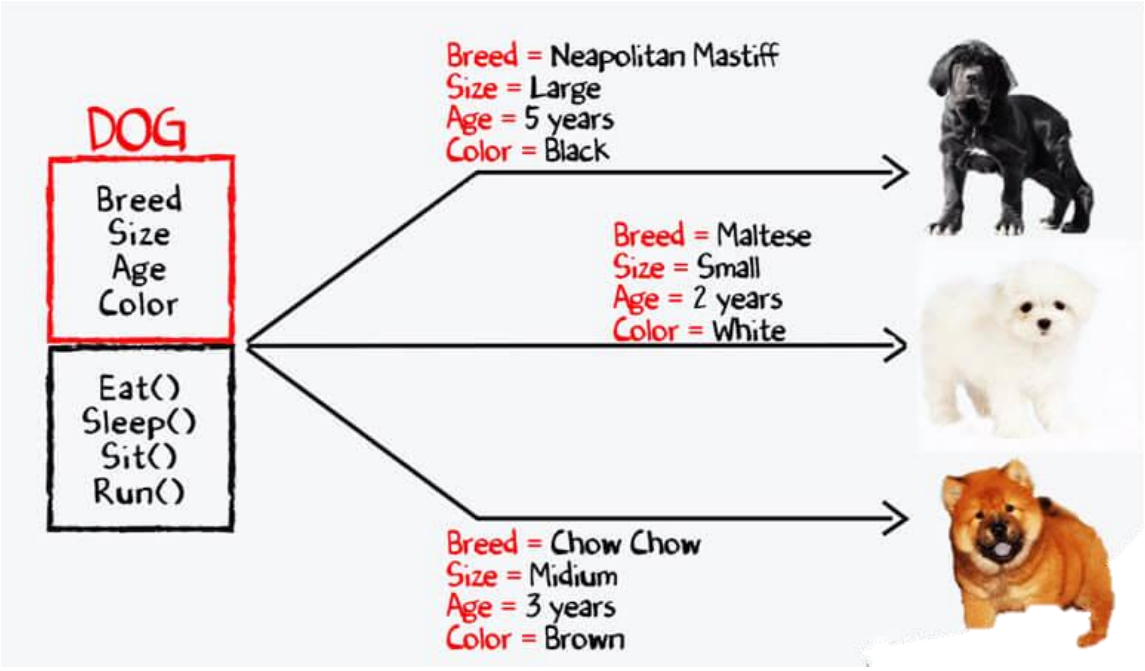


Рисунок 2.7

Приведем пример кода на языке Java.

Объявим класс Dog, а затем определим экземпляр этого класса под названием «мальтийский», используя ключевое слово new.

```
// Class Declaration
class Dog {
    // Instance Variables
    String breed;
    String size;
    int age;
    String color;

    // method 1
    public String getInfo() {
        return ("Breed is: "+breed+" Size is:"+size+" Age is:"+age+"
color is: "+color);
    }
}
public class Execute{
    public static void main(String[] args) {
        Dog maltese = new Dog();
        maltese.breed="Maltese";
        maltese.size="Small";
        maltese.age=2;
        maltese.color="white";
        System.out.println(maltese.getInfo());
    }
}
```

Результатом вывода данной программы будет

Порода: Мальтийская болонка Размер: Маленький Возраст:2 Цвет: белый.

Класс может использоваться в следующий вариантах:

- Класс используется для хранения как переменных данных, так и функций-членов.
- Он позволяет создавать объекты, определяемые пользователем.
- Класс предоставляет способ организации информации о данных.
- Вы можете использовать класс для наследования свойств другого класса.
- Классы могут использоваться для использования конструктора или деструктора.
- Его можно использовать для больших объемов данных и сложных приложений.

Объект, в свою очередь, используется:

- Это поможет вам узнать тип принятого сообщения и тип возвращенного ответа.
- Вы можете использовать объект для доступа к фрагменту памяти, используя переменную ссылки на объект.
- Он используется для манипулирования данными.
- Объекты представляют собой реальную проблему, для которой вы ищете решение.
- Это позволяет членам данных и функциям-членам выполнять желаемую задачу.

3 ОПРЕДЕЛЕНИЕ И СОЗДАНИЕ КЛАССОВ

3.1 Общий синтаксис

Общий синтаксис класса можно определить с помощью конструкции:

```
class имя_класса { список_компонентов };
```

- `имя_класса` - произвольно выбираемый идентификатор;
- `список_компонентов` - определения и описания типизированных данных и принадлежащих классу функций. Компонентами класса могут быть данные, функции, классы, перечисления, битовые поля и имена типов. Вначале для простоты будем считать, что компоненты класса – это типизированные данные (базовые и производные) и функции.
- Заключенный в фигурные скобки список компонентов называют *телом класса*.
- Телу класса предшествует заголовок. В простейшем случае заголовок класса включает слово `class` и имя.
- Определение класса всегда заканчивается точкой с запятой.

```
имя_класса имя_объекта;
```

Определение объекта класса предусматривает выделение участка памяти и деление этого участка на фрагменты, соответствующие отдельным элементам объекта.

Существует несколько уровней доступа к компонентам класса.

- спецификатор `public` делает члены открытыми;
- спецификатор `private` делает члены закрытыми;
- спецификатор `protected` открывает доступ к членам только для дружественных и дочерних классов

По умолчанию все переменные и функции, принадлежащие классу, определены как закрытые (`private`). Это означает, что они могут использоваться только внутри функций-членов самого класса. Для других частей программы, таких как функция `main()`, доступ к закрытым членам запрещен. Это, кстати, единственное отличие класса от структуры - в структуре все члены по умолчанию – `public`.

С использованием спецификатора доступа `public` можно создать открытый член класса, доступный для использования всеми функциями программы (как внутри класса, так и за его пределами).

```
class имя_класса
{
     закрытые переменный и функции;
     защищенные члены данных; защищенные конструкторы; защищенные
методы;
public:
     открытые переменные и функции;
     общедоступные свойства;
     общедоступные члены данных;
     общедоступные конструкторы;
     общедоступный деструктор;
     общедоступные методы;
} список имен объектов;
```

Синтаксис для доступа к данным конкретного объекта заданного класса (как и в случае структур), таков:

```
имя_объекта.имя_члена класса;
```

Рассмотрим пример:

```
# include <iostream>
using namespace std;
class Test{
    // так как спецификатор доступа не указан
    // данная переменная будет по умолчанию закрыта
    // для доступа вне класса (private)
    int one;

    // спецификатор доступа public все члены, идущие после него
    // будут открыты для доступа извне
public:

    // инициализировать переменные в классе
    // при создании запрещено, поэтому мы определяем
    // метод, реализующий данное действие
    void Initial(int o,int t){
        one=o;
        two=t;
    }
```

```
    // метод показывающий переменные класса на экран
    void Show(){
        cout<<"\n\n"<<one<<"\t"<<two<<"\n\n";
    }
    int two;
};
void main(){
    // создается объект с типом Test
    Test obj;
    // вызывается функция, инициализирующая его свойства
    obj.Initial(2,5);
    // показ на экран
    obj.Show(); // 2 5
    // прямая запись в открытую переменную two
    // с переменной one такая запись невозможна, так
    // как доступ к ней закрыт
    obj.two=45;
    // снова показ на экран
    obj.Show(); // 2 45

}
```

Отметим, что переменные класса нет необходимости передавать в методы класса в качестве параметров, так как они видны в них автоматически.

3.2 Оператор .

Внутри класса функции-члены могут вызывать друг друга и обращаться к переменным членам, не используя оператор точка(.) Этот оператор необходим, лишь когда обращение к функциям и переменным-членам осуществляется извне класса.

3.3 Оператор ->

Указатели на объекты. Доступ к членам объекта можно осуществлять и через указатель на объект. В этом случае применяется операция стрелка (→).

3.4 Оператор : :

Оператор `::` называется *оператором разрешения области видимости* (`scope resolution operator`). Т.е. создается шаблон класса и в нем шаблоны методов класса, а сами методы описываются вне описания класса.

```
class stack {
    int stck[SIZE];
    int tos;
public:
    void init();
    void push(int i);
    int pop();
};
```

```
void stack::init()
{
    tos = 0;
}

void stack::push(int i)
{
    if(tos==SIZE) {
        cout << "Stack is full.\n";
        return;
    }
    stck[tos] = i;
    tos++;
}
```

3.5 Константный метод объекта

Говорят, что метод объекта обладает свойством неизменности (константности), если после его выполнения состояние объекта не изменяется. Если не контролировать свойство неизменности, то его обеспечение будет целиком зависеть от квалификации программиста. Если же «неизменный» метод в процессе выполнения будет производить посторонние эффекты, то результат может быть самым неожиданным, отлаживать и поддерживать такой код очень тяжело.

Язык C++ позволяет пометить метод как константный. При этом неконстантные методы объекта запрещается использовать в теле помеченного метода, и в контексте этого метода ссылки на сам объект и все его поля будут константны. Для обозначения константности, используется модификатор `const`.

Давайте, рассмотрим пример класса с константными методами:

```
# include <iostream>
# include <string.h>
using namespace std;
class Personal
{
public:
    // конструктор с параметрами
    // мы выделяем здесь память
    // однако в нашем примере нет
    // ни деструктора, ни конструктора
    // копирования - единственная цель,
    // которую мы преследуем показать
    // работу константного метода
    Personal(char*p,char*n,int a){
        name=new char[strlen(n)+1];
        if(!name){
            cout<<"Error!!!";
            exit(0);
        }
```

```
        picture_data=new char[strlen(n)+1];
        if(!picture_data){
            cout<<"Error!!!";
            exit(0);
        }
        strcpy(picture_data,p);
        strcpy(name,n);
        age=a;
    }

    // Группа константных методов
    // внутри них невозможно
    // изменить какое-то из свойств
    const char*Name()const{
        return name;
    }
    int Age()const{
        return age;
    }
    const char*Picture()const{
        return picture_data;
    }

    void SetName(const char*n){
        strcpy(name,n);
    }
    void SetAge(int a){
        age=a;
    }
    void SetPicture(const char*p){
        strcpy(picture_data,p);
    }

private:
    char*picture_data; // путь к фотографии
    char*name; // имя
    int age; // возраст
};

void main(){
    Personal A("C:\\Image\\","Ivan",23);
    cout<<"Name: "<<A.Name()<<"\n\n";
    cout<<"Age: "<<A.Age()<<"\n\n";
    cout<<"Path for picture: "<<A.Picture()<<"\n\n";
    A.SetPicture("C:\\Test\\");
    A.SetName("Leonid");
    A.SetAge(90);
    cout<<"Name: "<<A.Name()<<"\n\n";
    cout<<"Age: "<<A.Age()<<"\n\n";
    cout<<"Path for picture: "<<A.Picture()<<"\n\n";
}
```

В данном примере методы `Name`, `Age`, `Picture` объявлены константными. Кроме того, можно наблюдать и использование константных указателей: параметр методов `SetName` и `SetPicture`, возвращаемое значение методов `Name` и `Picture`. Компилятор обеспечит проверку того, что реализация константных методов не имеет побочных эффектов в виде изменения состояния объекта, реализующего класс `Personal`. Как только обнаружится попытка выполнить запрещенную операцию, компилятор сообщит об ошибке.

3.6 Конструкторы и деструкторы

Любой переменной, участвующей в работе программы, требуется память и некоторое начальное значение. Для переменных встроенных типов размещение в памяти обеспечивается компилятором. Для локальных переменных память выделяется из стека программы и занимается для хранения значения данной переменной до тех, пока не закончится время ее жизни. Сложные типы данных также должны размещаться в памяти и уничтожаться, когда их время жизни закончилось. Это осуществляется с использованием конструкторов и деструкторов.

Конструктор (`constructor`) – это функция-член, имя которой совпадает с именем класса, инициализирующая переменные-члены, распределяющая память для их хранения (`new`).

Рассмотрим класс date:

```
class date
{
    int day, month, year;
public:
    set(int, int, int);
};
```

Нигде не утверждается, что объект должен быть инициализирован, и программист может забыть инициализировать его или сделать это дважды.

ООП дает возможность программисту описать функцию, явно предназначенную для инициализации объектов. Поскольку такая функция конструирует значения данного типа, она называется конструктором.

Конструктор всегда имеет то же имя, что и сам класс и никогда не имеет возвращаемого значения. Когда класс имеет конструктор, все объекты этого класса будут проинициализированы.

```
class date {
    int day, month, year;
public:
    date(int, int, int); // конструктор
};
```

Если конструктор требует аргументы, их следует указать:

```
date today = date(6,4,2014); // полная форма
date xmas(25,12,0); // сокращенная форма
// date my_burthday; // недопустимо, опущена инициализация
```

Если необходимо обеспечить несколько способов инициализации объектов класса, задается несколько перегруженных конструкторов с разным набором аргументов:

```
class date {
    int month, day, year;
public:
    date(int, int, int); // день месяц год
    date(char*); // дата в строковом представлении
    date(); // дата по умолчанию: сегодня
};
```

Конструкторы подчиняются тем же правилам относительно типов параметров, что и перегруженные функции.

Если конструкторы различаются по типам своих параметров, то компилятор при каждом использовании должен выбрать соответствующий:

```
date july4("Февраль 27, 2014");
date guy(27, 2, 2014);
date now; // инициализируется по умолчанию
```

Одним из способов сократить количество перегруженных функций (в том числе и конструкторов) является использование значений по умолчанию.

3.7 Конструктор по умолчанию

Конструктор, не требующий параметров, называется конструктором по умолчанию. Это может быть конструктор с пустым списком параметров или конструктор, в котором все аргументы имеют значения по умолчанию.

Конструкторы могут быть перегруженными, но конструктор по умолчанию может быть только один.

```
class date
{
    int month, day, year;
public:
    date(int, int, int);
    date(char*);
    date(); // конструктор по умолчанию
};
```

При создании объекта вызывается конструктор, за исключением случая, когда объект создается как копия другого объекта этого же класса, например:

```
date date2 = date1;
```

Однако имеются случаи, в которых создание объекта без вызова конструктора осуществляется неявно:

- *формальный параметр* – объект, передаваемый по значению, создается в стеке в момент вызова функции и инициализируется копией фактического параметра;

- *результат функции* – объект, передаваемый по значению, в момент выполнения оператора `return` копируется во временный объект, сохраняющий результат функции.

Во всех этих случаях транслятор не вызывает конструктора для вновь создаваемого объекта:

- `date2` в приведенном определении;
- для создаваемого в стеке формального параметра;
- для временного объекта, сохраняющего значение, возвращаемое функцией.

Вместо этого в них копируется содержимое объекта-источника:

- `date1` в приведенном примере;
- фактического параметра;
- объекта-результата в операторе `return`.

3.8 Конструктор копии

Как правило, при создании нового объекта на базе уже существующего происходит поверхностное копирование, то есть копируются те данные, которые содержит объект-источник.

При этом если в объекте-источнике имеются указатели на динамические переменные и массивы, или ссылки, то создание копии объекта требует обязательного дублирования этих объектов во вновь создаваемом объекте.

С этой целью вводится конструктор копии, который автоматически вызывается во всех перечисленных случаях. Он имеет единственный параметр – ссылку на объект-источник.

В качестве примера рассмотрим класс строки `String`, содержащий указатель на строку и длину строки. Класс содержит конструктор по умолчанию и методы

- `set()` – для установки нового значения строки;
- `out()` – для вывода значения строки.

Проблема

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <stdlib.h>
using namespace std;
class String
{
    char* str = 0;
    int size = 0;
public:
    String(char* arg);  // Конструктор по умолчанию (аргумент задан по умолчанию)
    char* out() { return str; };
    void set(char* arg);   // установка нового значения строки
};
void String::set(char* arg)
{
```

```
    if (str != 0)    // освобождаем память если в строке что-то было
        delete[] str;
    size = strlen(arg) + 1;
    str = new char[size];
    strcpy(str, arg);
}
String::String(char* arg = (char*)"") // конструктор по умолчанию
{
    set(arg);
}
int main()
{
    system("chcp 1251");
    system("cls");
    cout << "До изменения" << endl;
    String s = (char*)"abc";
    String p = s;
    cout << "s: " << s.out() << endl;
    cout << "p: " << p.out() << endl;
    s.set((char*)"rty");
    cout << "После изменения" << endl;
    cout << "s: " << s.out() << endl;
    cout << "p: " << p.out() << endl;
}
```

Результат выполнения:

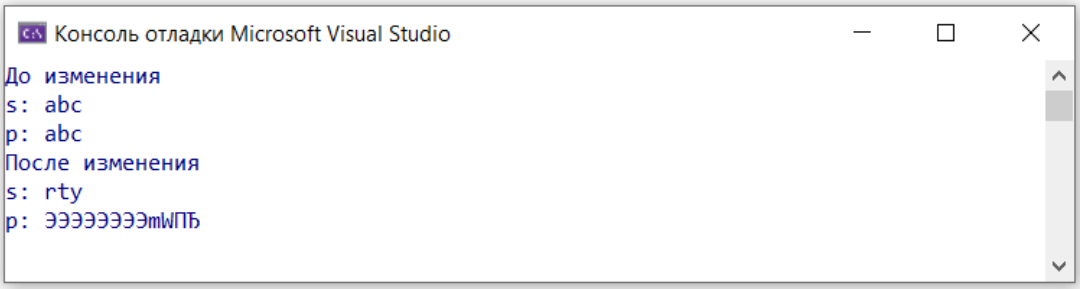


Рисунок 3.1

В результате того, что копирование строки (строка кода 32) было выполнено поверхностно, изменение поля `str` внутри объекта s приводит к разрушению поля `str` объекта `p`, поскольку функция `set()` перед переопределением строки освобождает память.

Решением этой проблемы станет использование конструктора копии.

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <stdlib.h>
using namespace std;
class String
{
    char* str = 0;
    int size = 0;
```

```
    public:
        String(char* arg);  // Конструктор по умолчанию (аргумент задан
по умолчанию)
        String(String&);  // Конструктор копии
        char* out() { return str; };
        void set(char* arg);   // установка нового значения строки
    };
    void String::set(char* arg)
    {
        if (str != 0)    // освобождаем память если в строке что-то было
            delete[] str;
        size = strlen(arg) + 1;
        str = new char[size];
        strcpy(str, arg);
    }
    String::String(char* arg = (char*)"")
    {
        set(arg);
    }
    String::String(String& right)  // тело конструктора копии
    {
        set(right.str);
    }
    int main()
    {
        system("chcp 1251");
        system("cls");
        cout << "До изменения" << endl;
        String s = (char*)"abc";
        String p = s;
        cout << "s: " << s.out() << endl;
        cout << "p: " << p.out() << endl;
        s.set((char*)"rty");
        cout << "После изменения" << endl;
        cout << "s: " << s.out() << endl;
        cout << "p: " << p.out() << endl;
    }
```

В коде функции `main()` ничего не поменялось.

В описание класса добавился конструктор копии.

Результат выполнения:

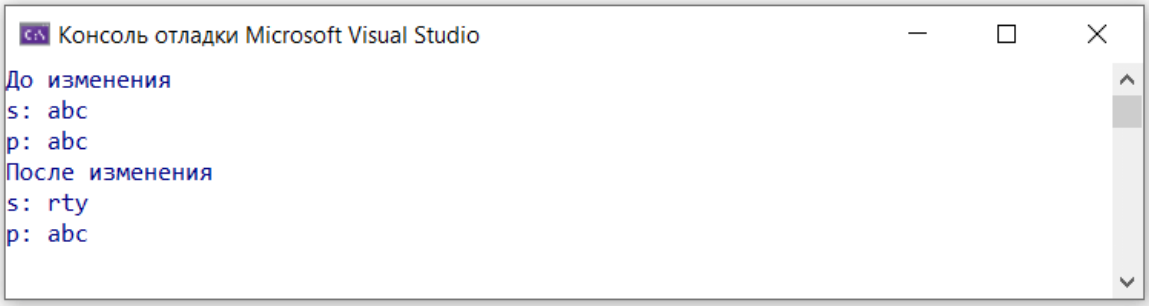


Рисунок 3.2

3.9 Деструктор

Определяемый пользователем класс имеет конструктор, который обеспечивает надлежащую инициализацию. Для многих типов также требуется обратное действие.

Деструктор обеспечивает соответствующую очистку объектов указанного типа. Имя деструктора представляет собой имя класса с предшествующим ему знаком «тильда» ~.

Так, для класса X деструктор будет иметь имя `~X()`.

Многие классы используют динамическую память, которая выделяется конструктором, а освобождается деструктором.

```
class date
{
    int day, year;
    char *month;
public:
    date(int d, char* m, int y)
    {
        day = d;
        month = new char[strlen(m)+1];
        strcpy_s(month, strlen(m)+1,m);
        year = y;
    }
    ~date() { delete[] month; } // деструктор
};
```

3. 10 Пример использования конструктора и деструктора

Пусть имеется класс `vect`, реализующий защищенный массив, и необходимо хранить несколько значений для каждого такого массива: возраст, вес и рост группы лиц. Группируем 3 массива внутри нового класса.

Конструктор нового класса имеет пустое тело и список вызываемых конструкторов класса `vect`, перечисленных после двоеточия `:` через запятую `,`. Они выполняются с целым аргументом `i`, создавая 3 объекта класса `vect: a, b, c.`

Конструкторы членов класса всегда выполняются до конструктора класса, в котором эти члены описаны. Порядок выполнения конструкторов для членов класса определяется порядком объявления членов класса. Если конструктору члена класса требуются аргументы, этот член с нужными аргументами указывается в списке инициализации.

Деструкторы вызываются в обратном порядке.

```
#include <iostream>
using namespace std;
class vect
{
    int size;
    int *array;
public:
```

```
    vect(int size)
    {
        this->size = size;
        array = new int;
        for (int i = 0; i < size; i++)
            array = 0;
    }
    int& element(int i) { return array; }
    int getSize() { return size; }
};
class multi_v
{
public:
    vect a;
    vect b;
    vect c;
    multi_v(int size): a(size), b(size), c(size) { }
    int getSize() { return a.getSize(); }
};
int main()
{
    system("chcp 1251");
    system("cls");
    multi_v f(3);
    for (int i = 0; i <= f.getSize(); i++)
    {
        f.a.element(i) = 10 + i;
        f.b.element(i) = 20 + 5 * i;
        f.c.element(i) = 120 + 5 * i;
    }
    for (int i = 0; i <= f.getSize(); i++)
    {
        cout << f.a.element(i) << "лет \t";
        cout << f.b.element(i) << "кг \t";
        cout << f.c.element(i) << "см" << endl;
    }
    cin.get();
    return 0;
}
```

При выполнении программы перед выходом из блока `main` для каждого члена `vect` будет вызываться индивидуальный деструктор.

Результат работы программы

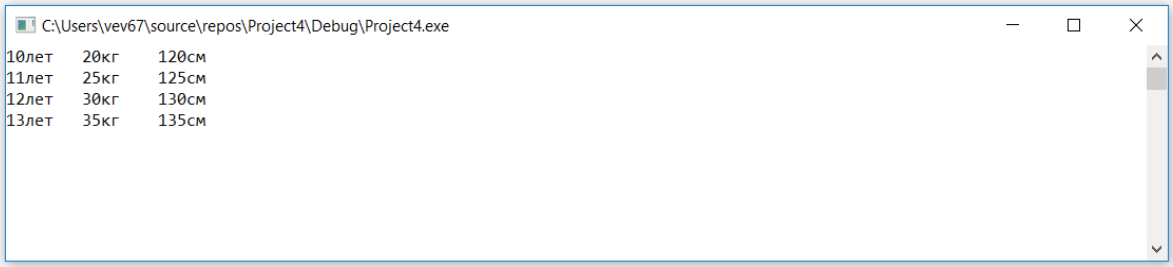


Рисунок 3.3

3.11 Статические члены класса

Кроме переменных и методов, которые относятся непосредственно к объекту, C++ позволяет определять переменные и методы, которые относятся непосредственно к классу или, иначе говоря, статические члены класса. Статические переменные и методы относят в целом ко всему классу. Для их определения используется ключевое слово `static`.

3.11.1 Статические поля

Статические переменные обычно применяются для хранения значений, специфичных для класса, для всех объектов класса в целом. То есть статические поля хранят состояние всего класса. Статическая переменная определяется только один раз и будет существовать, даже если объекты класса не были созданы.

Показательным примером статических переменных являются различные счетчики. Например, нам надо хранить количество созданных объектов. Такое количество относится классу, однако не зависит от конкретного объекта. Посмотрим на примере:

```
#include <iostream>

class Person
{
public:
    Person(std::string p_name, unsigned p_age)
    {
        ++count;   // при создании нового объекта увеличиваем счетчик
        name = p_name;
        age = p_age;
    }
    void print_count()
    {
        std::cout << "Created " << count << " objects" << std::endl;
    }
private:
    std::string name;
    unsigned age;
    static inline unsigned count{};  // статическое поле - счетчик
объектов Person
};
```

```
int main()
{
    Person tom{"Tom", 38};
    Person bob{"Bob", 42};
    Person sam{"Sam", 25};
    tom.print_count();
    bob.print_count();
    sam.print_count();
}
```

Здесь в классе `Person` определена статическая переменная `count` с помощью ключевого слова `static`:

```
static inline unsigned count{}; // инициализируем нулем
```

Обратите внимание, после `static` идет ключевое слово `inline`. Это ключевое слово в принципе необязательно для статических переменных и необходимо конкретно в данном случае для инициализации переменной `count`. В данном случае нулем.

При создании каждого нового объекта в конструкторе увеличиваем счетчик на единицу:

```
Person(std::string p_name, unsigned p_age)
{
    ++count;    // при создании нового объекта увеличиваем счетчик
}
```

В функциях класса `Person` мы можем обращаться к этой статической переменной. Например, в функции `print_count` выводим ее значение на консоль:

```
void print_count()
{
    std::cout << "Created " << count << " objects" << std::endl;
}
```

Для теста в функции `main` создаем три объекта `Person` и затем у каждого вызываем функцию `print_count`:

```
tom.print_count();
bob.print_count();
sam.print_count();
```

Но поскольку переменная `count` статическая и относится ко всему классу в целом и не зависит от конкретного объекта, во всех трех случаях будет выведено число `3`.

3.11.2 Статические функции

Статические функции также принадлежат классу в целом и не зависят от любого отдельного объекта класса. Обычно статические функции-члены используются для работы со статическими переменными. Например, в примере выше функция `print_count()` выводит значение статической переменной `count` и никак не зависит от конкретного объекта, не использует и не изменяет переменные и функции объектов. Поэтому такую функцию можно и даже лучше сделать статической:

```
#include <iostream>

class Person
{
public:
    Person(std::string p_name, unsigned p_age)
    {
        ++count;     // при создании нового объекта увеличиваем счетчик
        name = p_name;
        age = p_age;
    }
    // статическая функция
    static void print_count()
    {
        std::cout << "Created " << count << " objects" << std::endl;
    }
private:
    std::string name;
    unsigned age;
    static inline unsigned count{};  // статическое поле - счетчик объектов Person
};

int main()
{
    Person tom{"Tom", 38};
    Person bob{"Bob", 42};
    Person sam{"Sam", 25};
    tom.print_count();  // Created 3 objects
}
```

Для определения статической функции перед ней также указывается ключевое слово `static`:

```
static void print_count()
```

К подобным функциям также можно обращаться через имя объекта:

```
tom.print_count();
```

3.11.3 Обращение к статическим членам класса

Как выше было продемонстрировано, для обращения к статическим членам можно использовать имя любого объекта. Однако C++ также поддерживает и другой синтаксис:

```
класс::член_класса
```

После имени класса идет оператор `::` и имя статического компонента класса. Например:

```
#include <iostream>

class Person
{
public:
    static inline unsigned maxAge{120};     // статическая публичная
переменная
    Person(std::string p_name, unsigned p_age)
    {
        ++count;
        name = p_name;
        if(p_age < maxAge)  // если значение не больше максимального
            age = p_age;
    }
    // статическая функция
    static void print_count()
    {
        std::cout << "Created " << count << " objects" << std::endl;
    }
private:
    std::string name;
    unsigned age{1};
    static  inline  unsigned  count{};   //  статическая  приватная
переменная
};

int main()
{
    Person tom{"Tom", 38};
    Person bob{"Bob", 42};
    Person sam{"Sam", 25};

    // обращаемся к статической функции print_count
    Person::print_count();
    // обращаемся к статической переменной maxAge
    std::cout << "Max age: " << Person::maxAge << std::endl;
    // изменяем статическую переменную maxAge
    Person::maxAge = 110;
    std::cout << "Max age: " << Person::maxAge << std::endl;
}
```

Здесь добавлена публичная статическая переменная `maxAge`, которая представляет максимальный возраст. Поскольку этот показатель не зависит от определенного объекта и относится в целом к классу объектов `Person` (справедливо для всех людей), то определяем такую переменную как статическую. В конструкторе

используем эту переменную для верификации переданного в конструктор возраста. Если он больше максимального, то возраст получает значение по умолчанию – 1.

Далее в функции main мы можем по имени класса обратиться к статической функции print_count и переменной maxAge:

```
Person::print_count();
Person::maxAge
```

Консольный вывод программы:

```
Created 3 objects
Max age: 120
Max age: 110
```

3.11.4 Статические константы

Также можно определять статические константы. Так, в примере выше маловероятно, что значение maxAge будет меняться. Поэтому мы можем определить ее как константу. Ее значения нельзя будет изменить, а во остальном работа с ней идет также как и со статическими переменными:

```
#include <iostream>

class Person
{
public:
    static inline const unsigned maxAge{120};      // статическая константа
    Person(std::string p_name, unsigned p_age)
    {
        ++count;
        name = p_name;
        if(p_age < maxAge)  // если значение не больше максимального
            age = p_age;
    }
    // статическая функция
    static void print_count()
    {
        std::cout << "Created " << count << " objects" << std::endl;
    }
private:
    std::string name;
    unsigned age{1};
    static inline unsigned count{};   // статическая приватная переменная
};

int main()
{
    Person tom{"Tom", 38};
    Person bob{"Bob", 42};
```

```
    // обращаемся к статическим компонентам класса
    Person::print_count();
    std::cout << "Max age: " << Person::maxAge << std::endl;
}
```

4 ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

4.1 Инкапсуляция и скрытие данных

Одно из основных преимуществ использования объектов заключается в том, что объекту не нужно показывать все свои атрибуты и поведения. При хорошем объектно-ориентированном проектировании (по крайней мере, при таком, которое повсеместно считается хорошим) объект должен показывать только интерфейсы, необходимые другим объектам для взаимодействия с ним. Детали, не относящиеся к использованию объекта, должны быть скрыты от всех других объектов. Инкапсуляция определяется тем, что объекты содержат как атрибуты, так и поведения. Скрытие данных является основной частью инкапсуляции.

Например, объект, который применяется для вычисления квадратов чисел, должен обеспечивать интерфейс для получения результатов. Однако внутренние атрибуты и алгоритмы, используемые для вычисления квадратов чисел, не нужно делать доступными для запрашивающего объекта. Надежные классы проектируются с учетом инкапсуляции.

В объектно-ориентированном программировании **инкапсуляция** (или ***«сокрытие информации»***) – это процесс скрытого хранения деталей реализации объекта. Пользователи обращаются к объекту через открытый интерфейс.

В языке C++ инкапсуляция реализована через **спецификаторы доступа**. Как правило, все переменные-члены класса являются закрытыми (скрывая детали реализации), а большинство методов являются открытыми (с открытым интерфейсом для пользователя). Хотя требование к пользователям использовать публичный интерфейс может показаться более обременительным, нежели просто открыть доступ к переменным-членам, но на самом деле это предоставляет большое количество полезных преимуществ, которые улучшают возможность повторного использования кода и его поддержку.

4.1.1 Интерфейсы

Мы уже видели, что интерфейс определяет основные средства коммуникации между объектами. При проектировании любого класса предусматриваются интерфейсы для надлежащего создания экземпляров и эксплуатации объектов. Любое поведение, которое обеспечивается объектом, должно вызываться через сообщение, отправляемое с использованием одного из предоставленных интерфейсов. В случае с интерфейсом должно предусматриваться полное описание того, как пользователи соответствующего класса будут взаимодействовать с этим классом. В большинстве объектно-

ориентированных языков программирования методы, являющиеся частью интерфейсов, определяются как `public`.

Для того чтобы скрытие данных произошло, все атрибуты должны быть объявлены как `private`. Поэтому атрибуты никогда не являются частью интерфейсов. Частью интерфейсов классов могут быть только открытые методы. Объявление атрибута как `public` нарушает концепцию скрытия данных.

Взглянем на пример того, о чем совсем недавно шла речь: рассмотрим вычисление квадратов чисел. В таком примере интерфейс включал бы две составляющие:

- способ создать экземпляр объекта `Square`;
- способ отправить значение объекту и получить в ответ квадрат соответствующего числа.

Если пользователю потребуется доступ к атрибуту, то будет сгенерирован метод для возврата значения этого атрибута (*геттер*). Если затем пользователю понадобится получить значение атрибута, то будет вызван метод для возврата его значения. Таким образом, объект, содержащий атрибут, будет управлять доступом к нему. Это жизненно важно, особенно в плане безопасности, тестирования и сопровождения. Если вы контролируете доступ к атрибуту, то при возникновении проблемы вам не придется беспокоиться об отслеживании каждого фрагмента кода, который мог бы изменить значение соответствующего атрибута – оно может быть изменено только в одном месте (с помощью сеттера). С точки зрения безопасности вам не нужен неконтролируемый код для изменения или извлечения таких данных, как пароли и личная информация.

4.1.2 Реализации

Только открытые атрибуты и методы являются частью интерфейсов. Пользователи не должны видеть какую-либо часть внутренней реализации и могут взаимодействовать с объектами исключительно через интерфейсы классов. Таким образом, все определенное как `private` окажется недоступно пользователям и будет считаться частью внутренней реализации классов. В приводившемся ранее примере с классом `Employee` были скрыты только атрибуты. Во многих ситуациях будут попадаться методы, которые также должны быть скрыты и, таким образом, не являться частью интерфейса. В продолжение примера о вычислении квадратного корня, отметим при этом, что пользователям будет все равно, как вычисляется квадратный корень, при условии, что ответ окажется правильным. Таким образом, реализация может меняться, однако она не повлияет на пользовательский код. Например, компания, которая производит калькуляторы, может заменить алгоритм (возможно, потому что новый алгоритм оказался более эффективным), что не повлияет при этом на результаты.

4.1.3 Реальный пример парадигмы «интерфейс/реализация»

На рис. 4.1 проиллюстрирована парадигма «интерфейс/реализация» с использованием реальных объектов, а не кода. Тостеру для работы требуется электричество. Чтобы обеспечить подачу электричества, нужно вставить вилку шнура тостера в электрическую розетку, которая является интерфейсом. Для того чтобы получить требуемое электричество, тостеру нужно лишь «реализовать» шнур, который

соответствует техническим характеристикам электрической розетки; это и есть интерфейс между тостером и электроэнергетической компанией (в действительности – электроэнергетикой). Для тостера не имеет значения, что фактической реализацией является электростанция, работающая на угле. На самом деле для него важно лишь то, окажется реализацией атомная электростанция или же локальный электрогенератор. При такой модели любой электроприбор сможет получить электричество, если он соответствует спецификации интерфейса, как показано на рис. 4.1.

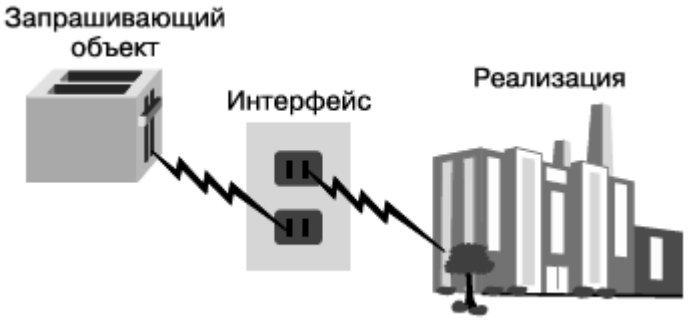


Рисунок 4.1 *Пример с электростанцией*

4.1.4 Геттеры и сеттеры: методы доступа к закрытым данным

В зависимости от класса, может быть уместным (в контексте того, что делает класс) иметь возможность получать/устанавливать значения закрытым переменным-членам класса.

Функция доступа – это короткая открытая функция, задачей которой является получение или изменение значения закрытой переменной-члена класса. Например:

```
class MyString
{
private:
    char *m_string; // динамически выделяем строку
    int m_length; // используем переменную для отслеживания длины строки

public:
    int getLength() { return m_length; } // функция доступа для получения значения m_length
};
```

Здесь `getLength()` является функцией доступа, которая просто возвращает значение `m_length`.

Функции доступа обычно бывают двух типов:

- **геттеры –** это функции, которые возвращают значения закрытых переменных-членов класса;
- **сеттеры –** это функции, которые позволяют присваивать значения закрытым переменным-членам класса.

Вот пример класса, который использует геттеры и сеттеры для всех своих закрытых переменных-членов:

```
class Date
{
private:
    int m_day;
    int m_month;
    int m_year;

public:
    int getDay() { return m_day; } // геттер для day
    void setDay(int day) { m_day = day; } // сеттер для day

    int getMonth() { return m_month; } // геттер для month
    void setMonth(int month) { m_month = month; } // сеттер для month

    int getYear() { return m_year; } // геттер для year
    void setYear(int year) { m_year = year; } // сеттер для year
};
```

В этом классе нет никаких проблем с тем, чтобы пользователь мог напрямую получать или присваивать значения закрытым переменным-членам этого класса, так как есть полный набор геттеров и сеттеров. В примере с классом `MyString` для переменной `m_length` не было предоставлено сеттера, так как не было необходимости в том, чтобы пользователь мог напрямую устанавливать длину.

Правило: *Предоставляйте функции доступа только в том случае, когда нужно, чтобы пользователь имел возможность получать или присваивать значения членам класса.*

Хотя иногда вы можете увидеть, что геттер возвращает **неконстантную ссылку** на переменную-член – этого следует избегать, так как в таком случае нарушается инкапсуляция, позволяя `caller`-у изменять внутреннее состояние класса вне этого же класса. Лучше, чтобы ваши геттеры использовали **тип возврата** по значению или по константной ссылке.

Правило: *Геттеры должны использовать тип возврата по значению или по константной ссылке. Не используйте для геттеров тип возврата по неконстантной ссылке.*

4.1.5 Преимущества инкапсуляции

1. ***Инкапсулированные классы проще в использовании и уменьшают сложность ваших программ.***

С полностью инкапсулированным классом вам нужно знать только то, какие методы являются доступными для использования, какие аргументы они принимают и какие значения возвращают. Не нужно знать, как класс реализован изнутри.

Все классы Стандартной библиотеки `C++` инкапсулированы. Представьте, насколько сложнее был бы процесс изучения языка `C++`, если бы вам нужно было знать

реализацию `std::string`, `std::vector` или `std::cout` (и других объектов) для того, чтобы их использовать!

2. ***Инкапсулированные классы помогают защитить ваши данные и предотвращают их неправильное использование.***

Глобальные переменные опасны, так как нет строгого контроля над тем, кто имеет к ним доступ и как их используют. Классы с открытыми членами имеют ту же проблему, только в меньших масштабах.

Например, рассмотрим класс с открытой переменной-членом в виде **массива**:

```
class IntArray
{
public:
    int m_array[10];
};
```

Если бы пользователи могли напрямую обращаться к массиву, то они могли бы использовать недопустимый индекс:

```
int main()
{
    IntArray array;
    array.m_array[16] = 2; // некорректный индекс, вследствие чего перезаписываем память, которой мы не владеем
}
```

Однако, если мы сделаем массив закрытым, то сможем заставить пользователя использовать функцию, которая первым делом проверяет корректность индекса:

```
class IntArray
{
private:
    int m_array[10]; // пользователь не имеет прямого доступа к этому члену

public:
    void setValue(int index, int value)
    {
        // Если индекс недействителен, то не делаем ничего
        if (index < 0 || index >= 10)
            return;

        m_array[index] = value;
    }
};
```

3. ***Инкапсулированные классы легче изменить. Инкапсуляция предоставляет возможность изменения способа реализации классов, не нарушая при этом работу всех программ, которые их используют.***

4. С инкапсулированными классами легче проводить отладку.

И, наконец, инкапсуляция помогает проводить отладку программ, когда что-то идет не по плану. Часто причиной неправильной работы программы является

некорректное значение одной из переменных. Если каждый объект имеет прямой доступ к переменной, то отследить часть кода, которая изменила переменную, может быть довольно-таки трудно. Однако, если для изменения значения нужно вызывать один и тот же метод, вы можете просто использовать точку останова для этого метода и посмотреть, как каждый вызывающий объект изменяет значение, пока не увидите что-то странное.

4.2 Наследование

Одной из наиболее сильных сторон объектно-ориентированного программирования, пожалуй, является повторное использование кода. При структурном проектировании повторное использование кода допускается в известной мере: вы можете написать процедуру, а затем применять ее столько раз, сколько пожелаете. Однако объектно-ориентированное проектирование делает важный шаг вперед, позволяя вам определять отношения между классами, которые не только облегчают повторное использование кода, но и способствуют созданию лучшей общей конструкции путем упорядочения и учета общности разнообразных классов. Основное средство обеспечения такой функциональности – *наследование*.

Наследование позволяет классу наследовать атрибуты и методы другого класса. Это дает возможность создавать абсолютно новые классы путем абстрагирования общих атрибутов и поведений. Одна из основных задач проектирования при объектно-ориентированном программировании заключается в выделении общности разнообразных классов. До пустим, у вас есть класс `Dog` и класс `Cat`, каждый из которых будет содержать атрибут `eyeColor`. При процедурной модели код как для `Dog`, так и для `Cat` включал бы этот атрибут. При объектно-ориентированном проектировании атрибут, связанный с цветом, можно перенести в класс с именем `Mammal` наряду со всеми прочими общими атрибутами и методами. В данном случае оба класса – `Dog` и `Cat` – будут наследовать от класса `Mammal`.

Итак, оба класса наследуют от `Mammal`. Это означает, что в итоге класс `Dog` будет содержать следующие атрибуты:

```
eyeColor // унаследован от Mammal
barkFrequency // определен только для Dog
```

В том же духе объект `Dog` будет содержать следующие методы:

```
getEyeColor // унаследован от Mammal
bark // определен только для Dog
```

Создаваемый экземпляр объекта `Dog` или `Cat` будет содержать все, что есть в его собственном классе, а также все имеющееся в родительском классе. Таким образом, `Dog` будет включать все свойства своего определения класса, а также свойства, унаследованные от класса `Mammal`.

4.2.1 Суперклассы и подклассы

Суперкласс, или *родительский класс* (иногда называемый *базовым*), содержит все атрибуты и поведения, общие для классов, которые наследуют от него. Например, в случае с классом `Mammal` все классы млекопитающих содержат аналогичные атрибуты,

такие как eyeColor и hairColor, а также поведения вроде generateInternalHeat и growHair. Все классы млекопитающих включают эти атрибуты и поведения, поэтому нет необходимости дублировать их, спускаясь по дереву наследования, для каждого типа млекопитающих. Дублирование потребует много дополнительной работы, и, пожалуй, что вызывает наибольшее беспокойство, оно может привести к ошибкам и несоответствиям.

Подкласс, или *дочерний класс* (иногда называемый *производным*), представляет собой расширение суперкласса. Таким образом, классы Dog и Cat наследуют все общие атрибуты и поведения от класса Mammal. Класс Mammal считается суперклассом подклассов, или дочерних классов, Dog и Cat.

Наследование обеспечивает большое количество преимуществ в плане проектирования. При проектировании класса Cat класс Mammal предоставляет значительную часть требуемой функциональности. Наследуя от объекта Mammal, Cat уже содержит все атрибуты и поведения, которые делают его настоящим классом млекопитающих. Точнее говоря, являясь классом млекопитающих такого типа, как кошки, Cat должен включать любые атрибуты и поведения, которые свойственны исключительно кошкам.

4.2.2 Абстрагирование

Дерево наследования может разрастись довольно сильно. Когда классы Mammal и Cat будут готовы, добавить другие классы млекопитающих, например собак (или львов, тигров и медведей), не составит особого труда. Класс Cat также может выступать в роли суперкласса. Например, может потребоваться дополнительно абстрагировать Cat, чтобы обеспечить классы для персидских, сиамских кошек и т. д. Точно так же, как и Cat, класс Dog может выступать в роли родительского класса для других классов, например GermanShepherd и Poodle. Мощь наследования заключается в его методиках абстрагирования и организации.

В большинстве объектно-ориентированных языков программирования (например, Java, .NET и Objective C) у класса может иметься только один родительский, но много дочерних классов. А в некоторых языках программирования, например C++, у одного класса может быть несколько родительских классов. В первом случае наследование называется *простым*, а во втором – *множественным.*

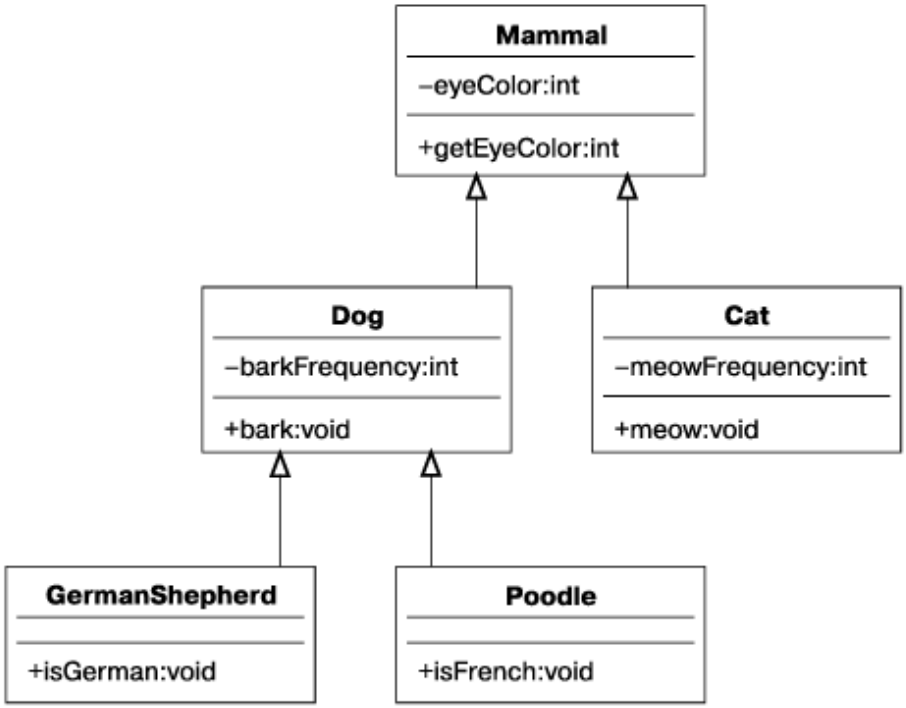


Рисунок 4.2 *UML-диаграмма классов млекопитающих*

Обратите внимание, что оба класса – `GermanShepherd` и `Poodle` – наследуют от `Dog` – каждый содержит только один метод. Однако, поскольку они наследуют от `Dog`, они также наследуют от `Mammal`. Таким образом, классы `GermanShepherd` и `Poodle` включают в себя все атрибуты и методы, содержащиеся в `Dog` и `Mammal`, а также свои собственные (рис. 4.3).

Рисунок 4.3 *Иерархия млекопитающих*

4.2.3 Отношения «является экземпляром»

Рассмотрим пример, в котором `Circle`, `Square` и `Star` наследуют от `Shape`. Это отношение часто называется отношением *«является экземпляром»*, поскольку круг – это форма, как и квадрат. Когда подкласс наследует от суперкласса, он получает все возможности, которыми обладает этот суперкласс. Таким образом, `Circle`, `Square` и `Star` являются расширениями `Shape`. На рис. 4.4 имя каждого из объектов

представляет метод Draw для Circle, Star и Square соответственно. При проектировании системы Shape очень полезно было бы стандартизировать то, как мы используем разнообразные формы. Так мы могли бы решить, что, если нам потребуется нарисовать фигуру любой формы, мы вызовем метод с именем Draw. Если мы станем придерживаться этого решения всякий раз, когда нам нужно будет нарисовать фигуру, то потребуется вызывать только метод Draw, независимо от того, какой она будет формы. В этом заключается фундаментальная концепция *полиморфизма* — на индивидуальный объект, будь то Circle, Star или Square, возлагается обязанность по рисованию фигуры, которая ему соответствует. Это общая концепция во многих современных приложениях, например предназначенных для рисования и обработки текста.

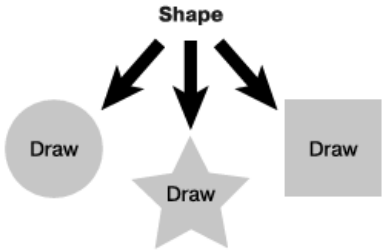


Рисунок 4.4 *Иерархия Shape*

4.2.4 Создание подклассов и суперклассов

Рассмотрим небольшую ситуацию, допустим, у нас есть классы, которые представляют человека и сотрудника компании:

```
class Person
{
public:
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << 
std::endl;
    }
    std::string name;       //  имя
    unsigned age;           // возраст
};
class Employee
{
public:
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << 
std::endl;
    }
    std::string name;       // имя
    unsigned age;           // возраст
    std::string company;    // компания
};
```

В данном случае класс Employee фактически содержит функционал класса Person: свойства name и age и функцию print. В целях демонстрации все переменные здесь определены как публичные. И здесь, с одной стороны, мы сталкиваемся с повторением функционала в двух классах. С другой стороны, мы также сталкиваемся с отношением **is** («является»). То есть мы можем сказать, что сотрудник компании ЯВЛЯЕТСЯ человеком. Так как сотрудник компании имеет в принципе все те же признаки, что и человек (имя, возраст), а также добавляет какие-то свои (компанию). Поэтому в этом случае лучше использовать механизм наследования. Унаследуем класс Employee от класса Person:

```
class Person
{
public:
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    std::string name;       //  имя
    unsigned age;           // возраст
};
class Employee : public Person
{
public:
    std::string company;    // компания
};
```

Для установки отношения наследования после названия класса ставится двоеточие, затем идет спецификатор доступа и название класса, от которого мы хотим унаследовать функциональность.

Спецификатор доступа позволяет указать, к каким членам класса производный класс будет иметь доступ. В данном случае используется спецификатор **public**:

```
public:
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    std::string name;       //  имя
    unsigned age;           // возраст
```

который позволяет использовать в производном классе все публичные члены базового класса. Если мы не используем модификатор доступа, то класс Employee ничего не будет знать о переменных name и age и функции print.

После установки наследования мы можем убрать из класса Employee те переменные, которые уже определены в классе Person. Используем оба класса:

```
#include <iostream>
```

```
class Person
{
public:
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    std::string name;       //  имя
    unsigned age;           // возраст
};
class Employee : public Person
{
public:
    std::string company;    // компания
};

int main()
{
    Person tom;
    tom.name = "Tom";
    tom.age = 23;
    tom.print();    // Name: Tom       Age: 23

    Employee bob;
    bob.name = "Bob";
    bob.age = 31;
    bob.company = "Microsoft";
    bob.print();    // Name: Bob       Age: 31
}
```

4.2.5 Конструкторы

Но теперь сделаем все переменные приватными, а для их инициализации добавим конструкторы. И тут стоит учитывать, что конструкторы при наследовании **не наследуются**. И если базовый класс содержит только конструкторы с параметрами, то производный класс должен вызывать в своем конструкторе один из конструкторов базового класса:

```
#include <iostream>

class Person
{
public:
    Person(std::string name, unsigned age)
    {
        this->name = name;
        this->age = age;
    }
    void print() const
```

```
        {
            std::cout << "Name: " << name << "\tAge: " << age << std::endl;
        }
    private:
        std::string name;       //  имя
        unsigned age;           // возраст
    };
    class Employee: public Person
    {
    public:
        Employee(std::string name, unsigned age, std::string company): Person(name, age)
        {
            this->company = company;
        }
    private:
        std::string company;    // компания
    };

    int main()
    {
        Person person {"Tom", 38};
        person.print();     // Name: Tom       Age: 38

        Employee employee {"Bob", 42, "Microsoft"};
        employee.print();   // Name: Bob       Age: 42
    }
```

После списка параметров конструктора производного класса через двоеточие идет вызов конструктора базового класса, в который передаются значения параметров `n` и `a`.

```
    Employee(std::string name, unsigned age, std::string company): Person(name, age)
    {
        this->company = company;
    }
```

Если бы мы не вызвали конструктор базового класса, то это было бы ошибкой.

Консольный вывод программы:

```
Name: Tom       Age: 38
Name: Bob       Age: 42
```

Таким образом, в строке

```
Employee employee {"Bob", 42, "Microsoft"};
```

Вначале будет вызываться конструктор базового класса `Person`, в который будут передаваться значения `"Bob"` и `42`. И таким образом будут установлены имя и возраст. Затем будет выполняться собственно конструктор `Employee`, который установит компанию.

Также мы могли бы определить конструктор Employee следующим образом, используя списки инициализации:

```
    Employee(std::string name, unsigned age, std::string company): 
Person(name, age), company(company)
    {
    }
```

4.2.6 Подключение конструктора базового класса

В примерах выше конструктор Employee отличается от конструктора Person одним параметром – company. Все остальные параметры из Employee передаются в Person. Однако, если бы у нас было бы полное соответствие по параметрам между двумя классами, то мы могли бы и не определять отдельный конструктор для Employee, а подключить конструктор базового класса:

```
    #include <iostream>

    class Person
    {
    public:
        Person(std::string name, unsigned age)
        {
            this->name = name;
            this->age = age;
        }
        void print() const
        {
            std::cout << "Name: " << name << "\tAge: " << age << 
std::endl;
        }
    private:
        std::string name;       //  имя
        unsigned age;           // возраст
    };
    class Employee: public Person
    {
    public:
        using Person::Person;   // подключаем конструктор базового класса
    };

    int main()
    {
        Person person {"Tom", 38};
        person.print();     // Name: Tom       Age: 38

        Employee employee {"Bob", 42};
        employee.print();   // Name: Bob       Age: 42
    }
```

Здесь в классе Employee подключаем конструктор базового класса с помощью ключевого слова **using**:

```
using Person::Person;
```

Таким образом, класс Employee фактически будет иметь тот же конструктор, что и Person с теми же двумя параметрами. И этот конструктор мы также можем вызвать для создания объекта Employee:

```
Employee employee {"Bob", 42};
```

4.2.7 Определение конструкторов копирования

При определении конструктора копирования в производном классе следует вызывать в нем конструктор копирования базового класса. Например, добавим в классы Person и Employee конструкторы копирования:

```
#include <iostream>

class Person
{
public:
    // конструктор копирования класса Person
    Person(const Person& person)
    {
        name = person.name;
        age = person.age;
    }
    Person(std::string name, unsigned age)
    {
        this->name = name;
        this->age = age;
    }
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
private:
    std::string name;
    unsigned age;
};
class Employee: public Person
{
public:
    Employee(std::string name, unsigned age, std::string company): Person(name, age)
    {
        this->company = company;
    }
    // конструктор копирования класса Employee
    // вызываем конструктор копирования базового класса
```

```
    Employee(const Employee& employee): Person(employee)
    {
        company=employee.company;
    }
private:
    std::string company;
};

int main()
{
    Employee tom{"Tom", 38, "Google"};
    Employee tomas{tom};    // вызываем конструктор копирования
    tomas.print();  // Name: Tom       Age: 38
}
```

В конструкторе копирования производного класса `Employee` вызываем конструктор копирования базового класса `Person`:

```
Employee(const Employee& employee): Person(employee)
{
    company=employee.company;
}
```

При этом в конструктор копирования `Person` передается объект `employee`, где будут установлены переменные `name` и `age`. В самом же конструкторе класса `Employee` лишь устанавливается переменная `company`.

4.2.8 Наследование деструкторов

Уничтожение объекта производного класса может вовлекать как собственно деструктор производного класса, так и деструктор базового класса. Например, определим в обоих классах деструкторы

```
~Person()
{
    std::cout << "Person deleted" << std::endl;
}
```

и

```
~Employee()
{
    std::cout << "Employee deleted" << std::endl;
}
```

В обоих классах деструктор просто выводит некоторое сообщение. В функции `main` создается один объект `Employee`, однако при завершении программы будет вызываться деструктор как из производного, так и из базового класса:

```
Person created
Employee created
Name: Tom       Age: 38
```

```
Employee deleted
Person deleted
```

По консольному выводу мы видим, что при создании объекта `Employee` сначала вызывается конструктор базового класса `Person` и затем собственно конструктор `Employee`. А при удалении объекта `Employee` процесс идет в обратном порядке - сначала вызывается деструктор производного класса и затем деструктор базового класса. Соответственно, если в деструкторе базового класса идет освобождение памяти, то оно в любом случае будет выполнено при удалении объекта производного класса.

4.2.9 Запрет наследования

Иногда наследование от класса может быть нежелательно. И с помощью спецификатора **`final`** мы можем запретить наследование:

```
class Person final
{
};
```

После этого мы не сможем унаследовать другие классы от класса `Person`. И, например, если мы попробуем написать, как в случае ниже, то мы столкнемся с ошибкой:

```
class Employee : public Person
{
};
```

4.3 Полиморфизм

Полиморфизм – это греческое слово, буквально означающее множественность форм. Несмотря на то, что полиморфизм тесно связан с наследованием, он часто упоминается отдельно от него как одно из наиболее весомых преимуществ объектно-ориентированных технологий. Если потребуется отправить сообщение объекту, он должен располагать методом, определенным для ответа на это со общение. В иерархии наследования все подклассы наследуют от своих суперклассов. Однако, поскольку каждый подкласс представляет собой отдельную сущность, каждому из них может потребоваться дать отдельный ответ на одно и то же сообщение.

Возьмем, к примеру, класс `Shape` и поведение с именем `Draw`. Когда вы попросите кого-то нарисовать фигуру, первый вопрос вам будет звучать так: «Какой формы?» Никто не сможет нарисовать требуемую фигуру, не зная формы, которая является абстрактной концепцией (кстати, метод `Draw()` в коде `Shape` не содержит реализации). Вы должны указать конкретную форму. Для этого потребуется обеспечить фактическую реализацию в `Circle`. Несмотря на то что `Shape` содержит метод `Draw`, `Circle` переопределит этот метод и обеспечит собственный метод `Draw()`. Переопределение, в сущности, означает замену реализации родительского класса на реализацию из дочернего класса.

Допустим, у вас имеется массив из трех форм – `Circle`, `Square` и `Star`. Даже если вы будете рассматривать их все как объекты `Shape` и отправите сообщение `Draw` каждому объекту `Shape`, то конечный результат для каждого из них будет разным,

поскольку Circle, Square и Star обеспечивают фактические реализации. Одним словом, каждый класс способен реагировать на один и тот же метод Draw не так, как другие, и рисовать соответствующую фигуру. Это и понимается под полиморфизмом.

Взгляните на следующий класс Shape:

```
public abstract class Shape{
  private double area;
  public abstract double getArea();
}
```

Класс Shape включает атрибут с именем area, который содержит значение площади фигуры. Метод getArea() включает идентификатор с именем abstract. Когда метод определяется как abstract, подкласс должен обеспечивать реализацию для этого метода; в данном случае Shape требует, чтобы подклассы обеспечивали реализацию getArea(). А теперь создадим класс с именем Circle, который будет наследовать от Shape (ключевое слово extends будет указывать на то, что Circle на следует от Shape):

```
public class Circle extends Shape{
  double radius;
  public Circle(double r) {
    radius = r;
  }
  public double getArea() {
    area = 3.14*(radius*radius);
    return (area);
  }
 }
```

Здесь мы познакомимся с новой концепцией под названием *«конструктор»*. Класс Circle содержит метод с таким же именем – Circle. Если имя метода оказывается аналогичным имени класса и при этом не предусматривается возвращаемого типа, то это особый метод, называемый *конструктором*. Считайте конструктор точкой входа для класса, где создается объект. Конструктор хорошо подходит для выполнения инициализаций и задач, связанных с запуском.

Конструктор Circle принимает один параметр, представляющий радиус, и присваивает его атрибуту radius класса Circle.

Класс Circle также обеспечивает реализацию для метода getArea, изначально определенного как abstract в классе Shape.

Мы можем создать похожий класс с именем Rectangle:

```
public class Rectangle extends Shape{
  double length;
  double width;
  public Rectangle(double l, double w){
    length = l;
    width = w;
  }
```

```
  public double getArea() {
    area = length*width;
    return (area);
  }
}
```

Теперь мы можем создавать любое количество классов прямоугольников, кругов и т. д. и вызывать их метод `getArea()`. Ведь мы знаем, что все классы прямоугольников и кругов наследуют от `Shape`, а все классы `Shape` содержат метод `getArea()`. Если подкласс наследует абстрактный метод от суперкласса, то он должен обеспечивать конкретную реализацию этого метода, поскольку иначе он сам будет абстрактным классом (см. рис. 4.5, где приведена UML-диаграмма). Этот подход также обеспечивает механизм для довольно легкого создания других, новых классов.

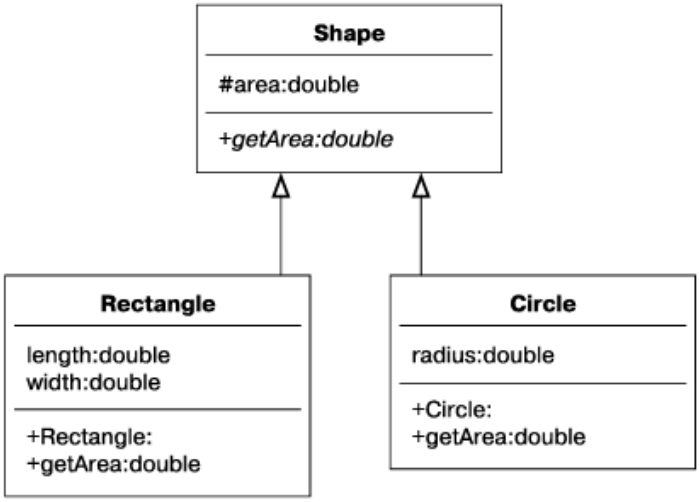


Рисунок 4.5 *UML-диаграмма Shape*

Этот подход направлен на обеспечение стандартизации определенного интерфейса среди классов, а также приложений. Представьте себе приложение из офисного пакета, которое позволяет обрабатывать текст, и приложение для работы с электронными таблицами. Предположим, что они оба включают класс с именем `Office`, который содержит интерфейс с именем `print()`. Этот `print()` необходим всем классам, являющимся частью офисного пакета. Любопытно, но несмотря на то, что текстовый процессор и табличная программа вызывают интерфейс `print()`, они делают разные вещи: один выводит текстовый документ, а другая – документ с электронными таблицами.

Позднее и раннее связывание

С понятием полиморфизма тесно связано понятие *виртуальная функция*. Это специальным образом оформленная функция, которая может быть в так называемом полиморфном состоянии – состоянии, при котором вызов нужной функции из набора виртуальных формируется на этапе *позднего связывания*. Понятие позднее связывание означает, что код вызова нужной функции формируется при выполнении программы. Иными словами, в исходном коде вызов функции только обозначается без точного

указания того, какая именно функция должна быть вызвана. Объект, для которого вызывается виртуальная функция, имеет общее значение. Конкретный объект и соответствующая ему функция будут сформированы на этапе выполнения программы.

Механизм виртуальных функций реализует основополагающий *принцип полиморфизма*: *«один интерфейс, несколько реализаций»* или *«один интерфейс, несколько методов»*.

Как известно, существует также и *раннее связывание*. При раннем связывании известно, какие объекты используются при вызове функции в каждом случае. Это накладывает ограничения на возможности программного кода. Изменение объектов для функций с одинаковыми именами невозможно в процессе выполнения программы, это изменение нужно программировать каждый раз вручную (это и есть ограничение кода).

Не все задачи нуждаются в использовании позднего связывания. Выбор того, какой вид связывания будет использоваться в программе, зависит от специфики решаемой задачи.

Для реализации позднего связывания требуется следующее:

- классы обязаны образовывать иерархию с помощью механизма наследования;
- в иерархии классов были определены функции, имеющие одинаковое имя и список параметров;
- функции с одинаковым именем и параметрами должны быть отмечены как виртуальные (с ключевым словом `virtual`).

Относительно парадигмы классов и объектов для полиморфизма можно выделить следующие характерные особенности:

- полиморфизм не характеризует объект;
- реализацию полиморфизма определяет архитектура класса;
- полиморфизм является характеристикой функций-членов класса;
- весь класс не может быть полиморфным, полиморфными могут быть только функции члены класса.

Статический и динамический полиморфизм

В языке C++ есть возможность реализовывать два вида полиморфизма:

- *статический*. Этот вид полиморфизма достигается путём использования перегруженных функций (раннее связывание), шаблонов классов и перегрузки операторов.
- *динамический*. В этом случае используется наследование в сочетании с виртуальными функциями (позднее связывание).

Виртуальная функция – это функция, объявляемая в базовом классе и переопределяемая в производном классе. Производный класс по своему усмотрению реализует виртуальную функцию. Чтобы объявить виртуальную функцию, используется ключевое слово `virtual`.

Сигнатура виртуальной функции, объявленная в базовом классе, определяет вид интерфейса, реализуемого этой функцией. Интерфейс определяет способ вызова виртуальной функции. Для каждого конкретного класса виртуальная функция имеет свою реализацию, обеспечивающую выполнение действий, свойственных только этому

классу. Таким образом, виртуальная функция для конкретного класса является неким уникальным (конкретным) методом (specific method).

В наиболее упрощенном виде объявление виртуальной функции в классе может быть следующим:

```
class BaseClass
{
  virtual return_type FuncNameVirtual(list_of_parameters)
  {
    // ...
  }
};
```

здесь

- FuncNameVirtual() – имя виртуальной функции;
- return_type – тип, возвращаемый функцией;
- list_of_parameters – список параметров, которые получает функция.

В унаследованном классе виртуальная функция FuncNameVirtual() продолжает цепочку виртуальных функций для классов низших уровней. Для этого не обязательно указывать ключевое слово virtual, поскольку это слово уже указано в базовом классе BaseClass.

```
class DerivedClass
{
  return_type FuncNameVirtual(list_of_parameters)
  {
    // Это также виртуальная функция, которая переопределяет функцию базового класса.
    // ...
  }
}
```

Возможны ситуации, когда в производном классе объявлена функция, которая может восприниматься как виртуальная, однако она не есть виртуальной. Примеры таких ситуаций:

- функция с такой же сигнатурой как в базовом классе, но объявленная как константная (const);
- функция, которой передаются аргументы типа, совместимого с аргументами функции базового класса.

В языке C++ в унаследованном классе, виртуальная функция, которая переопределяет одноименную функцию базового класса, может быть объявлена с спецификатором override. Хотя этот спецификатор не обязателен, объявление нужно для лучшей информативности. При беглом осмотре унаследованного класса сразу видно виртуальные функции (override). Исходя из этого, приблизительный вид унаследованного класса может быть примерно следующим

```
class DerivedClass
{
```

```
  return_type FuncNameVirtual(list_of_parameters) override
  {
    // Для цепочки виртуальных функций так нужно делать всегда
    // ...
  }
}
```

После указания спецификатора `override` программист имеет лучшую информативность о том, что эта функция виртуальна и она переопределяет одноименную виртуальную функцию базового класса.

Если для функции, объявленной со спецификатором `override` в производном классе `DerivedClass`, нет подходящей виртуальной функции в базовом классе `BaseClass`, компилятор сгенерирует ошибку.

Чистые виртуальные функции и абстрактные классы

Иногда возникает необходимость определить класс, который не предполагает создания конкретных объектов. Например, класс фигуры. В реальности есть конкретные фигуры: квадрат, прямоугольник, треугольник, круг и так далее. Однако абстрактной фигуры самой по себе не существует. В то же время может потребоваться определить для всех фигур какой-то общий класс, который будет содержать общую для всех функциональность. И для описания подобных сущностей используются абстрактные классы.

Абстрактные классы – это классы, которые содержат или наследуют без переопределения хотя бы одну чистую виртуальную функцию. Абстрактный класс определяет интерфейс для переопределения производными классами.

Что такое **чистые виртуальные функции** (`pure virtual functions`)? Это функции, которые не имеют определения. Цель подобных функций – просто определить функционал без реализации, а реализацию определят производные классы. Чтобы определить виртуальную функцию как чистую, ее объявление завершается значением «`=0`». Например, определим абстрактный класс, который представляет геометрическую фигуру:

```
class Shape
{
public:
    virtual double getSquare() const = 0;      // площадь фигуры
    virtual double getPerimeter() const = 0;   // периметр фигуры
};
```

Класс `Shape` является абстрактным, потому что он содержит как минимум одну чистую виртуальную функцию. А в данном случае даже две таких функции – для вычисления площади и периметра фигуры. И ни одна из функций не имеет никакой реализации. В данном случае обе функции являются константными, но это необязательно. Главное, чтобы любой производный класс от `Shape` должен будет предоставить для этих функций свою реализацию.

При этом мы не можем создать объект абстрактного класса:

```
Shape shape{};
```

Для применения абстрактного класса определим следующую программу:

```
#include <iostream>

class Shape
{
public:
    virtual double getSquare() const = 0;      // площадь фигуры
    virtual double getPerimeter() const = 0;  // периметр фигуры
};
class Rectangle : public Shape  // класс прямоугольника
{
public:
    Rectangle(double w, double h) : width(w), height(h)
    { }
    double getSquare() const override
    {
        return width * height;
    }
    double getPerimeter() const override
    {
        return width * 2 + height * 2;
    }
private:
    double width;   // ширина
    double height;  // высота
};
class Circle : public Shape     // круг
{
public:
    Circle(double r) : radius(r)
    { }
    double getSquare() const override
    {
        return radius * radius * 3.14;
    }
    double getPerimeter() const override
    {
        return 2 * 3.14 * radius;
    }
private:
    double radius;  // радиус круга
};

int main()
{
    Rectangle rect{30, 50};
    Circle circle{30};
```

```
        std::cout << "Rectangle square: " << rect.getSquare() <<
std::endl;
        std::cout << "Rectangle perimeter: " << rect.getPerimeter() <<
std::endl;
        std::cout << "Circle square: " << circle.getSquare() <<
std::endl;
        std::cout << "Circle perimeter: " << circle.getPerimeter() <<
std::endl;
    }
```

Здесь определены два класса-наследника от абстрактного класса `Shape` – `Rectangle` (прямоугольник) и `Circle` (круг). При создании классов-наследников все они должны либо определить для чистых виртуальных функций конкретную реализацию, либо повторить объявление чистой виртуальной функции. Во втором случае производные классы также будут абстрактными.

В данном же случае и `Circle`, и `Rectangle` являются конкретными классами и реализуют все виртуальные функции.

Консольный вывод программы:

```
Rectangle square: 1500
Rectangle perimeter: 160
Circle square: 2826
Circle perimeter: 188.4
```

Стоит отметить, что абстрактный класс может определять и обычные функции и переменные, может иметь несколько конструкторов, но при этом нельзя создавать объекты этого абстрактного класса.

Примеры полиморфизм

В примере демонстрируется механизм полиморфизма, заключающийся в передаче некоторой функции указателя на базовый класс. Базовый класс и его подкласс (класс, унаследованный от базового) содержат виртуальную функцию `PrintInfo()` без параметров, которая выводит информационное сообщение о данном классе.

```
#include <iostream>
using namespace std;

// Базовый класс
class Base
{
public:
  // virtual - признак виртуальной функции
  virtual void PrintInfo()
  {
    cout << "Base." << endl;
  }
};

// Класс, унаследованный от класса Base,
// важно: здесь должен быть модификатор public
class Derived : public Base
```

```
{
public:
  virtual void PrintInfo() override // спецификатор override желательно указывать
  {
    cout << "Derived." << endl;
  }
};

void main()
{
  // 1. Создать экземпляры базового и производного класса
  Base obj1;
  Derived obj2;

  // 2. Объявить указатель на базовый класс
  Base* p;

  // 3. Использовать правило: указатель на базовый класс может указывать
  //    на любой экземпляр базового и производного от него класса.
  //    Ниже демонстрируется полиморфизм.
  // 3.1. Установить указатель p на экземпляр базового класса obj1
  //    и вызвать PrintInfo()
  p = &obj1;
  p->PrintInfo(); // Base

  // 3.2. Установить указатель p на экземпляр производного класса
  //      и вызвать PrintInfo()
  p = &obj2;
  p->PrintInfo(); // Derived - это есть полиморфизм (слово virtual)
}
```

Результат выполнения программы

```
Base.
Derived.
```

Проанализируем вышеприведенный код.

В примере объявляются два класса `Base` и `Derived`, образующие иерархию с помощью механизма наследования.

Для обеспечения полиморфизма используется правило: для классов, образующих иерархию, указатель на базовый класс может ссылаться на экземпляр базового класса и любого унаследованного класса из этой иерархии. Поэтому в программе объявляется строка

```
...

// 3. Объявить указатель на базовый класс
Base* p;
```

```
...
```

Теперь указателю **p** можно присваивать адрес любого экземпляра классов `Base` и `Derived`. Сначала присваивается адрес экземпляра `obj1` типа `Base` и вызывается метод `PrintInfo()`

```
p = &obj1;
p->PrintInfo(); // Base
```

Вывод будет прогнозированным – слово «`Base`».

Затем указателю `p` присваивается адрес экземпляра `obj2` типа `Base` и вызывается метод `PrintInfo()`

```
p = &obj2;
p->PrintInfo();
```

Вывод будет «`Derived`». То есть будет вызван метод `PrintInfo()` производного класса, что нам и нужно. Вызов этого метода обеспечивает ключевое слово `virtual` в объявлении функции `PrintInfo()` базового класса `Base`.

Если в классе `Base` перед объявление функции `PrintInfo()` убрать ключевое слово `virtual`, то в следующем коде

```
p = &obj2;
p->PrintInfo();
```

будет вызван метод `PrintInfo()` класса `Base`, а не класса `Derived`. Это означает, что полиморфизм не будет поддерживаться, а всегда будет вызываться функция базового класса. В результате программа выведет

```
Base.
Base.
```

Таким образом, функция `PrintInfo()` класса `Derived` для указателя `p` на базовый класс `Base` будет недоступна.

Вывод. *Полиморфизм реализует правило «один интерфейс, много реализаций». В нашем случае интерфейс один – это объявление и вызов функции* `PrintInfo()`

```
p->PrintInfo();
```

Но в зависимости от того, на какой объект указывает указатель **p**, будет вызван соответствующий метод `PrintInfo()` – это и есть много реализаций.

4.4 Композиция

Вполне естественно представлять себе, что одни объекты содержат другие объекты. У телевизора есть тюнер и экран. У компьютера есть видеокарта, клавиатура и жест кий диск. Хотя компьютер сам по себе можно считать объектом, его жесткий диск тоже считается полноценным объектом.

Фактически вы могли бы открыть системный блок компьютера, достать жесткий диск и подержать его в руке. Как компьютер, так и его жесткий диск считаются объектами. Просто компьютер содержит другие объекты, например жесткий диск.

Таким образом, объекты зачастую формируются или состоят из других объектов – это и есть композиция.

4.4.1 Абстрагирование

Точно так же, как и наследование, композиция обеспечивает механизм для создания объектов. Фактически, есть только два способа создания классов из других классов: *наследование* и *композиция*. Как мы уже видели, наследование позволяет одному классу наследовать от другого. Поэтому мы можем абстрагировать атрибуты и поведения для общих классов. Например, как собаки, так и кошки относятся к млекопитающим, поскольку собака *является экземпляром* млекопитающего так же, как и кошка. Благодаря композиции мы к тому же можем создавать классы, вкладывая одни классы в другие.

Взглянем на отношение между автомобилем и двигателем. Преимущества раз деления двигателя и автомобиля очевидны. Создавая двигатель отдельно, мы сможем использовать его в разных автомобилях – не говоря уже о других преимуществах. Однако мы не можем сказать, что двигатель *является* экземпляром автомобиля. Это будет просто неправильно звучать, если так выразиться (а поскольку мы моделируем реальные системы, это нам и нужно). Вместо этого для описания отношений композиции мы используем словосочетание *«содержит как часть»*. Автомобиль *содержит* как часть двигатель.

5 ОБРАБОТКА ИСКЛЮЧЕНИЙ

В процессе работы программы могут возникать различные ошибки. Например, при передаче файла по сети оборвется сетевое подключение или будут введены некорректные и недопустимые данные, которые вызовут падение программы. Такие ошибки еще называются исключениями. Исключение представляет временный объект любого типа, который используется для сигнализации об ошибке. Цель объекта-исключения состоит в том, чтобы передать информацию из точки, в которой произошла ошибка, в код, который должен ее обработать. Если исключение не обработано, то при его возникновении программа прекращает свою работу.

В C++ различают ошибки времени компиляции и ошибки времени выполнения. Ошибки первого типа обнаруживает компилятор до запуска программы. К ним относятся, например, синтаксические ошибки в коде. Ошибки второго типа проявляются при запуске программы. Примеры ошибок времени выполнения: ввод некорректных данных, некорректная работа с памятью, недостаток места на диске и т. д. Часто такие ошибки могут привести к неопределённому поведению программы.

Некоторые ошибки времени выполнения можно обнаружить заранее с помощью проверок в коде. Например, такими могут быть ошибки, нарушающие инвариант класса в конструкторе. Обычно, если ошибка обнаружена, то дальнейшее выполнение функции не имеет смысла, и нужно сообщить об ошибке в то место кода, откуда эта функция была вызвана. Для этого предназначен механизм исключений.

5.1 Коды возврата и исключения

Рассмотрим функцию, которая считывает со стандартного потока возраст и возвращает его вызывающей стороне. Добавим в функцию проверку корректности возраста: он должен находиться в диапазоне от `0` до `128` лет. Предположим, что повторный ввод возраста в случае ошибки не предусмотрен.

```
int ReadAge() {
    int age;
    std::cin >> age;
    if (age < 0 || age >= 128) {
        // Что вернуть в этом случае?
    }
    return age;
}
```

Что вернуть в случае некорректного возраста? Можно было бы, например, договориться, что в этом случае функция возвращает ноль. Но тогда похожая проверка должна быть и в месте вызова функции:

```
int main() {
    if (int age = ReadAge(); age == 0) {
        // Произошла ошибка
    } else {
        // Работаем с возрастом age
    }
}
```

Такая проверка неудобна. Более того, нет никакой гарантии, что в вызывающей функции программист вообще её напишет. Фактически мы тут выбрали некоторое значение функции (ноль), обозначающее ошибку. Это пример подхода к обработке ошибок через коды возврата. Другим примером такого подхода является хорошо знакомая нам функция `main`. Только она должна возвращать ноль при успешном завершении и что-либо ненулевое в случае ошибки.

Другим способом сообщить об обнаруженной ошибке являются исключения. С каждым сгенерированным исключением связан некоторый объект, который как-то описывает ошибку. Таким объектом может быть что угодно – даже целое число или строка. Но обычно для описания ошибки заводят специальный класс и генерируют объект этого класса:

```
#include <iostream>

struct WrongAgeException {
    int age;
};

int ReadAge() {
    int age;
    std::cin >> age;
    if (age < 0 || age >= 128) {
```

```
            throw WrongAgeException(age);
        }
        return age;
    }
```

Здесь в случае ошибки оператор `throw` генерирует исключение, которое представлено временным объектом типа `WrongAgeException`. В этом объекте сохранён для контекста текущий неправильный возраст `age`. Функция досрочно завершает работу: у неё нет возможности обработать эту ошибку, и она должна сообщить о ней наружу. Поток управления возвращается в то место, откуда функция была вызвана. Там исключение может быть перехвачено и обработано.

Для обработки исключений применяется конструкция `try...catch`. Она имеет следующую форму:

```
try
{
    инструкции, которые могут вызвать исключение
}
catch(объявление_исключения)
{
    обработка исключения
}
```

В блок кода после ключевого слова `try` помещается код, который потенциально может сгенерировать исключение.

После ключевого слова `catch` в скобках идет параметр, который передает информацию об исключении. Затем в блоке производится собственно обработка исключения.

```
int main() {
    try {
        age = ReadAge();  // может сгенерировать исключение
        // Работаем с возрастом age
    } catch (const WrongAgeException& ex) {    // ловим объект
исключения
        std::cerr << "Age is not correct: " << ex.age << "\n";
        return 1;   // выходим из функции main с ненулевым кодом
возврата
    }
    // ...
}
```

Мы знаем заранее, что функция `ReadAge` может сгенерировать исключение типа `WrongAgeException`. Поэтому мы оборачиваем вызов этой функции в блок `try`. Если происходит исключение, для него подбирается подходящий `catch`-обработчик. Таких обработчиков может быть несколько. Можно смотреть на них как на набор перегруженных функций от одного аргумента – объекта исключения. Выбирается первый подходящий по типу обработчик и выполняется его код. Если же ни один обработчик не подходит по типу, то исключение считается необработанным. В этом

случае оно пробрасывается дальше по стеку – туда, откуда была вызвана текущая функция. А если обработчик не найдётся даже в функции `main`, то программа аварийно завершается.

Усложним немного наш пример, чтобы из функции `ReadAge` могли вылетать исключения разных типов. Сейчас мы проверяем только значение возраста, считая, что на вход поступило число. Но предположим, что поток ввода досрочно оборвался, или на входе была строка вместо числа. В таком случае конструкция `std::cin >> age` никак не изменит переменную `age`, а лишь возведёт специальный флаг ошибки в объекте `std::cin`. Наша переменная `age` останется непроинициализированной: в ней будет лежать неопределённый мусор. Можно было бы явно проверить этот флаг в объекте `std::cin`, но мы вместо этого включим режим генерации исключений при таких ошибках ввода:

```
int ReadAge() {
    std::cin.exceptions(std::istream::failbit);
    int age;
    std::cin >> age;
    if (age < 0 || age >= 128) {
        throw WrongAgeException(age);
    }
    return age;
}
```

Теперь ошибка чтения в операторе `>>` у потока ввода будет приводить к исключению типа `std::istream::failure`. Функция `ReadAge` его не обрабатывает. Поэтому такое исключение покинет пределы этой функции. Поймаем его в функции `main`:

```
int main() {
    try {
        age = ReadAge();  // может сгенерировать исключения разных
типов
        // Работаем с возрастом age
    } catch (const WrongAgeException& ex) {
        std::cerr << "Age is not correct: " << ex.age << "\n";
        return 1;
    } catch (const std::istream::failure& ex) {
        std::cerr << "Failed to read age: " << ex.what() << "\n";
        return 1;
    } catch (...) {
        std::cerr << "Some other exception\n";
        return 1;
    }
    // ...
}
```

При обработке мы воспользовались функцией `ex.what` у исключения типа `std::istream::failure`. Такие функции есть у всех исключений стандартной библиотеки: они возвращают текстовое описание ошибки.

Обратите внимание на третий `catch` с многоточием. Такой блок, если он присутствует, будет перехватывать любые исключения, не перехваченные ранее.

5.2 try-catch и деструкторы

Стоит отметить, что, если в блоке `try` создаются некоторые объекты, то при возникновении исключения у них вызываются деструкторы. Например:

```
#include <iostream>

class Person
{
public:
    Person(std::string name) :name{ name }
    {
        std::cout << "Person " << name << " created" << std::endl;
    }
    ~Person()
    {
        std::cout << "Person " << name << " deleted" << std::endl;
    }
    void print()
    {
        throw "Print Error";
    }
private:
    std::string name;
};

int main()
{
    try
    {
        Person tom{ "Tom" };
        tom.print();    // Здесь генерируется ошибка
    }
    catch (const char* error)
    {
        std::cerr << error << std::endl;
    }
}
```

В классе `Person` определяет деструктор, который выводит сообщение на консоль. В функции `print` просто генерируем исключение.

В функции `main` в блоке `try` создаем один объект `Person` и вызываем у него функцию `print`, что естественно приведет к генерарции исключения и переходу управления программы в блок **`catch`**. И если мы посмотрим на консольный вывод

```
Person Tom created
Person Tom deleted
Print Error
```

то мы увидим, что, прежде чем, начнется обработка исключения в блоке `catch`, будет вызван деструктор объекта `Person`.

5.3 Создание своих типов исключений

Для каких-то специфичных задач мы можем создавать свои типы исключений, что позволяет нам передавать более структурированную и комплексную информацию об ошибке, нежели примитивные типы. Например, рассмотрим следующую программу:

```
#include <iostream>

class AgeException
{
public:
    AgeException(std::string message): message{message}{}
    std::string getMessage() const {return message;}
private:
    std::string message;
};

class Person
{
public:
    Person(std::string name, unsigned age)
    {
        if(!age||age>110)   // если возраст равен 0 или больше 110
        {
            throw AgeException{"Invalid age"};
        }
        this->name = name;
        this->age = age;
    }
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << 
std::endl;
    }
private:
    std::string name;
    unsigned age;
};

int main()
{
    try
    {
        Person tom{"Tom", 38};      // Корректные данные
        tom.print();
```

```
        Person bob{"Bob", 1500};     // Некорректные данные
        bob.print();
    }
    catch (const AgeException& ex)
    {
        std::cout << ex.getMessage() << std::endl;
    }
}
```

Здесь определяется класс `Person`, в конструктор которого передается имя и возраст пользователя. Однако нам необходимо, чтобы возраст был в некотором разумном диапазоне, например, от `1` до `110`. И в этом случае в конструкторе класса проверяем переданное значение возраста. Если оно выходит за допустимые пределы, с помощью оператора `throw` генерируем исключение класса `AgeException`, который чуть выше определен.

```
throw AgeException{"Invalid age"};
```

Класс `AgeException` специально создан, чтобы инкапсулировать исключение, связанное с возрастом человека. Этот класс просто хранит сообщение об ошибке и определяет метод `getMessage` для доступа к нему.
В конструкции `try-catch` для теста определяем пару объектов `Person`. При передаче некорректного возраста:

```
Person bob{"Bob", 1500};
```

будет генерироваться исключение `AgeException`, и управление перейдем в блок `catch`, который обрабатывает данный тип исключений:

```
catch (const AgeException& ex)
{
    std::cout << ex.getMessage() << std::endl;
}
```

Чтобы не происходило ненужного копирования объекта исключения, в блок `catch` объект исключения передается по ссылке.

Соответственно консольный вывод программы будет следующим

```
Name: Tom       Age: 38
Invalid age
```

5.4 Создание производных классов исключений

Все исключения в языке `C++` описываются типом **`exception`**, который определен в заголовочном файле **`<exception>`**. И при обработке исключений мы также можем использовать данный класс, интерфейс которого выглядит следующим образом

```
namespace std
{
    class exception
```

```
        {
        public:
            exception() noexcept;
            exception(const exception&) noexcept;
            exception& operator=(const exception&) noexcept;
            virtual ~exception(); // Destructor
            virtual const char* what() const noexcept; // возвращает
сообщение об исключении
        };
    }
```

Используем данный тип для обработки исключения:

```
#include <iostream>

double divide(int, int);

int main()
{
    int x {500};
    int y{};
    try
    {
        double z = divide(x, y);
        std::cout << z << std::endl;
    }
    catch (const std::exception& err)
    {
        std::cout << "Error!!!" << std::endl;
    }
    std::cout << "The End..." << std::endl;
}
double divide(int a, int b)
{
    if (!b)
        throw std::exception();
    return a / b;
}
```

Прежде всего, оператору `throw` передается объект типа **`std::exception`**

```
throw std::exception();
```

Если мы хотим отловить исключения типа `exception`, то нам надо в выражении `catch` определить переменную этого типа:

```
catch (const std::exception& err)
```

То есть здесь `err` представляет константную ссылку на объект `exception`. Если мы не собираемся использовать эту переменную в блоке `catch`, то можно указать просто тип исключения:

```
catch (std::exception)
```

```
    {
        std::cout << "Error!!!" << std::endl;
    }
```

С помощью функции **what()** можно получить сообщение об ошибке в виде строки в C-стиле. Однако непосредственно для типа `std::exception` он имеет мало смысла, поскольку просто выводит название класса. Но в производных классах он может использоваться для вывода сообщения об ошибке.

На основе класса `std::exception` мы можем создавать свои собственные типы исключений. Например:

```
#include <iostream>

class person_error: public std::exception
{
public:
    person_error(const std::string& message): message{message}
    {}
    const char* what() const noexcept override
    {
        return message.c_str();     // получаем из std::string строку
const char*
    }
private:
    std::string message;    // сообщение об ошибке
};

class Person
{
public:
    Person(std::string name, unsigned age)
    {
        if(!age || age > 110)    // если age==0 или age > 110
            throw person_error("Invalid age");
        this->name = name;
        this->age = age;
    }
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age <<
std::endl;
    }
private:
    std::string name;
    unsigned age;
};

void testPerson(std::string name, unsigned age)
{
    try
```

```
        {
            Person person{name, age};   // создаем один объект Person
            person.print();
        }
        catch (const person_error& err) // обработка ошибок, сязанных с
Person
        {
            std::cout << "Person error: " << err.what() << std::endl;
        }
        catch (const std::exception&)   // обработка остальных исключений
        {
            std::cout << "Something wrong"<< std::endl;
        }
    }
    int main()
    {
        testPerson("Tom", 38);  // Name: Tom       Age: 38
        testPerson("Sam", 250);  // Person error: Invalid age
    }
```

Здесь определен класс `Person`, который представляет пользователя. В конструктор класса передается имя и возраст. Однако передаваемое число может превышать разумный возраст или быть равно нулю. В этом случае мы генерируем исключение типа `person_error`:

```
Person(std::string name, unsigned age)
{
    if(!age || age > 110)    // если age==0 или age > 110
        throw person_error("Invalid age");
```

Класс `person_error` унаследован от `std::exception`, через конструктор получает сообщение об ошибке и хранит его в переменной `message`:

```
class person_error: public std::exception
{
public:
    person_error(const std::string& message): message{message}
    {}
    const char* what() const noexcept override
    {
        return message.c_str();     // получаем из std::string строку
const char*
    }
private:
    std::string message;    // сообщение об ошибке
};
```

Для возвращения сообщения нам надо переопределить виртуальную функцию `what()`. Но проблема заключается в том, что функция возвращает строку `const char*`, но класс хранит сообщение в виде строки `std::string`. И чтобы получить из `std::string` значение `const char*`, у строки вызываем функцию `c_str()`

```
    return message.c_str();     // получаем из std::string строку
const char*
```

Для теста определена функция `testPerson`, в которой в блоке `try` создается объект `Person`. Конструкция `try..catch` использует два блока `catch` для обработки исключений. Первый блок обрабатывает исключения производного типа - класса `person_error`, а последний блок представляет базовый тип `exception`

```
    catch (const person_error& err) // обработка ошибок, сязанных с
                                    // Person
    {
        std::cout << "Person error: " << err.what() << std::endl;
    }
    catch (const std::exception&)   // обработка остальных исключений
    {
        std::cout << "Something wrong"<< std::endl;
    }
```

И в данном случае программа выдаст следующий результат:

```
Name: Tom       Age: 38
Person error: Invalid age
```

6 ВИЗУАЛЬНОЕ МОДЕЛИРОВАНИЕ

Под моделью ПО в общем случае понимается формализованное описание системы ПО на определенном уровне абстракции. Каждая модель определяет конкретный аспект системы, использует набор диаграмм и документов заданного формата, а также отражает точку зрения и является объектом деятельности различных людей с конкретными интересами, ролями или задачами.

Графические (визуальные) модели представляют собой средства для визуализации, описания, проектирования и документирования архитектуры системы. Разработка модели системы ПО промышленного характера в такой же мере необходима, как и наличие проекта при строительстве большого здания. Хорошие модели являются основой взаимодействия участников проекта и гарантируют корректность архитектуры. Поскольку сложность систем повышается, важно располагать хорошими методами моделирования. Хотя имеется много других факторов, от которых зависит успех проекта, но наличие строгого стандарта языка моделирования является весьма существенным.

Состав моделей, используемых в каждом конкретном проекте, и степень их детальности в общем случае зависят от следующих факторов:

- сложности проектируемой системы;
- необходимой полноты ее описания;
- знаний и навыков участников проекта;
- времени, отведенного на проектирование.
- Визуальное моделирование оказало большое влияние на развитие ПО вообще и CASE-средств в частности.

Понятие CASE (Computer Aided Software Engineering) используется в настоящее время в весьма широком смысле. Первоначальное значение этого понятия, ограниченное только задачами автоматизации разработки ПО, в настоящее время приобрело новый смысл, охватывающий большинство процессов жизненного цикла ПО. CASE-технология представляет собой совокупность методов проектирования ПО, а также набор инструментальных средств, позволяющих в наглядной форме моделировать предметную область, анализировать эту модель на всех стадиях разработки и сопровождения ПО и разрабатывать приложения в соответствии с информационными потребностями пользователей. Большинство существующих CASE-средств основано на методах структурного или объектно-ориентированного анализа и проектирования, использующих спецификации в виде диаграмм или текстов для описания внешних требований, связей между моделями системы, динамики поведения системы и архитектуры программных средств.

Методы структурного анализа и проектирования ПО

В структурном анализе и проектировании используются различные модели, описывающие:

1. Функциональную структуру системы;
2. Последовательность выполняемых действий;
3. Передачу информации между функциональными процессами;
4. Отношения между данными.

Наиболее распространенными моделями первых трех групп являются:

- *функциональная модель SADT (Structured Analysis and Design Technique);*
- *модель IDEF0, IDEF3;*
- *DFD (Data Flow Diagrams) - диаграммы потоков данных.*

В настоящее время производство ПО достигло огромного размаха. В программировании задействованы тысячи специалистов в различных странах, ПО входит в состав самых разных промышленных продуктов, является основой многочисленных сервисов и технологий в бизнесе, образовании, в социальной инфраструктуре и бытовой жизни. Рост производства в этой области продолжается высокими темпами.

В конце 60-х годов прошлого века исследователи в поисках **способов упорядочивания и стандартизации процесса создания ПО** обратились к другим, уже устоявшимся промышленным областям. Было замечено, что в строительстве, машиностроении, электротехнике и т. д. работы по созданию новых систем разбиваются на два основных этапа – **проектирование и реализацию**. Проектирование осуществляют архитекторы, конструкторы, инженеры, а изготовляют систему рабочие, строители, монтеры. Результаты проектирования фиксируются с помощью чертежей – схематичных изображений создаваемой системы. Эти чертежи служат хорошим интерфейсом между проектировщиками и теми, кто, собственно, создает, делает саму систему. Чертежи обязывают последних строго следовать принятым решениям, избавляя от необходимости думать над принципиальными вопросами. Главное уже продумано и решено, и все, что нужно – это разобраться в чертежах системы, понять, как и что нужно сделать, и сделать это. Ситуации, когда по ходу разработки приходится менять

проектные решения, как правило, случаются крайне редко. Таким образом, чертежи сыграли важную роль в становлении современной промышленности, позволяя эффективно разделить труд между квалифицированными инженерами и обычными рабочими. Аналогичным образом хотелось бы применять чертежи и в программировании. Однако здесь существуют некоторые особенности, которые не позволили использовать чертежное проектирование **«as is»**. ПО и другие инженерные объекты. Инженерные дисциплины, успешно использующие чертежи, занимаются разработкой материальных, видимых объектов.

Существует большое количество различных метафор для изображения ПО. На рис. 6.1 представлен пример, изображающий условное предложение `if B then S1 else S2; S3` с использованием графического языка `VIPR` (`VIsual Imperative Programming`). Однако очевидно, что большие программы так изображать неудобно

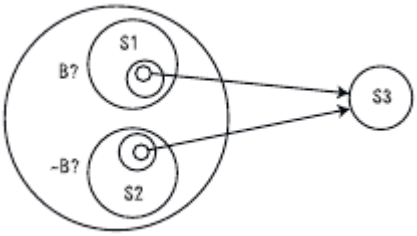


Рисунок 6.1 *Условное предложение в нотации языка VIPR*

Графовая метафора. Среди различных метафор визуализации ПО выделяются математические графы – вершины, изображаемые по-разному, и ребра – стрелки, связи, зависимости и т. д. На рис. 6.2 приводится несколько типов диаграмм, используемых на практике при проектировании ПО.

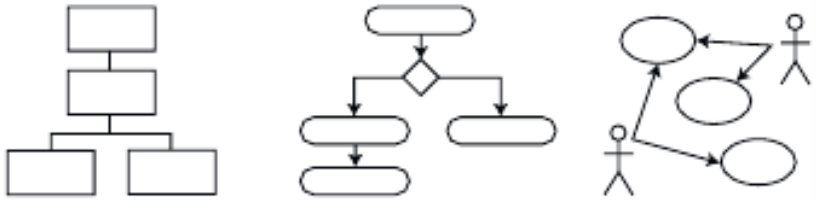


Рис. 6.2 *Примеры разных графов, используемых в визуальном моделировании*

Очевидно, что на этом рисунке изображены разные графы. На настоящий момент, несмотря на многочисленные попытки, другой общеупотребительной метафоры визуализации ПО не создано. Однако не все виды диаграмм, применяемые в рамках визуального моделирования, являются графами, например, **диаграммы последовательностей (sequence diagrams**) или **временные диаграммы (timing diagrams**) **UML**. Однако из тринадцати видов этих диаграмм UML 2.0 только два не являются графами.

6.1 Определение визуального моделирования.

Итак, **визуальное моделирование (visual modeling)** является методом, применяемым в разработке ПО, который:

• использует графовые модели для визуализации ПО;

• предлагает моделировать ПО с разных точек зрения;

• может применяться в разработки и эволюции ПО, а также в различных видах деятельности по его созданию.

Принципиально, что в одном проекте используются разные визуальные модели ПО, созданные с разных точек зрения. **Визуальные модели, как правило, не составляют «сплошных»** спецификаций, подобно программам, но **часто являются, скорее, фрагментами**, формально не связанными друг с другом. Эти модели описывают отдельные аспекты ПО, которые нужно прояснить в определенной ситуации для той или иной категории лиц, участвующих в проекте или как-либо с ним связанных. В целом визуальное моделирование служит **для повышения понимаемости решений проекта людьми** – разными категориями задействованных в проекте специалистов (инженеров-электронщиков, менеджеров, заказчика и т. д.). Визуальное моделирование может применяться как при разработке, так и при сопровождении ПО.

При разработке – главным образом при проектировании и анализе системы, которые предшествуют непосредственному программированию.

При сопровождении – когда новые разработчики изучают доставшееся им ПО. Визуальное моделирование может также использоваться в разных видах деятельности процесса разработки ПО: главным образом при анализе и проектировании, но также и при документировании, тестировании, разработке требований и т. д. Средства визуального моделирования. Визуальное моделирование применяется на практике с помощью методов, языков и соответствующих программных инструментов (см. рис. 6.3).

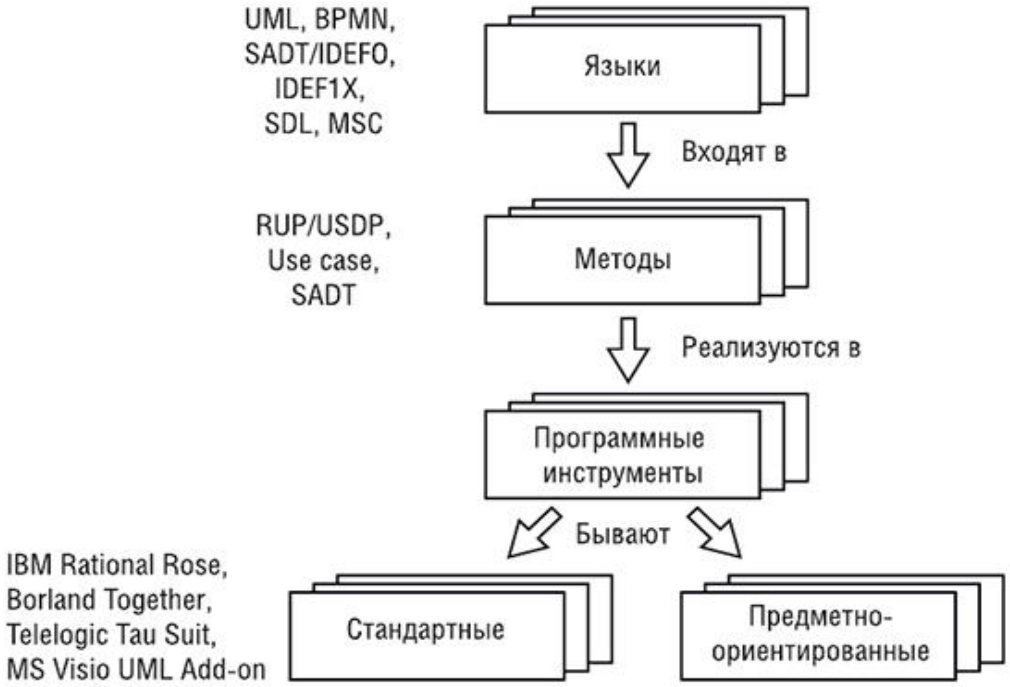


Рисунок 6.3 *Визуальное моделирование: языки, методы, программные средства*

Языки визуального моделирования (или визуальные языки) – это формализованные наборы графических символов и правила построения из них визуальных моделей. Сейчас известны и активно используются на практике такие языки визуального моделирования, как UML (Unified Modeling Language) и BPMN (Business Process Model and Notation). Однако существуют и более старые языки: SDL и MSC для моделирования телекоммуникационных систем, SADT/IDEF0 для моделирования бизнес-процессов, IDEF1x для моделирования баз данных и некоторые другие. Кроме того, в исследовательской среде создано множество других визуальных языков, например, язык WebML для моделирования web-приложений. Методы использования визуального моделирования предписывают правила применения визуальных языков для решения тех или иных задач процесса разработки ПО.

В качестве примера кратко рассмотрим метод SADT (Structured Analysis and Design Technique). Этот метод предназначен для структурного анализа создаваемой или модифицируемой системы и является способом уменьшить количество дорогостоящих ошибок за счет структуризации знаний о системе на ранних этапах ее разработки, улучшения взаимодействия разработчиков и пользователей/заказчиков, а также сглаживания перехода от анализа к проектированию. Он включает в себя визуальный язык, а также подробно описанные принципы и технологию использования этого языка.

Термин «структурный анализ» был введен в обиход *Дугласом Россом (Douglas Ross)* – главным автором SADT – в конце 60-х годов. Коротко историю развития SADT можно представить следующим образом:

- 60-е годы – группа ученых из MIT (Massachusetts Institute of Technology) под руководством Дугласа Росса создала метод иерархической модульной декомпозиции программных систем под названием SADT;
- в 1969 авторы SADT основали компанию SoftTech, которая стала развивать и коммерциализировать этот метод;
- 1973 год – первая масштабная апробация SADT – проект по созданию завода будущего;
- конец 70-х годов – SADT был использован в программе интегрированной компьютеризации производства ICAM (Integrated ComputerAided Manufacturing) военно-воздушных сил США, что привело к стандартизации части SADT под названием IDEF0 и широкому распространению этого стандарта в военной промышленности США. В настоящее время при разработке ПО SADT не используется, но активно применяется при моделировании бизнес-процессов.

С точки зрения SADT модель **может основываться либо на функциях системы, либо на ее предметах** (планах, данных, оборудовании, информации и т.д.). Соответствующие модели принято называть **функциональными моделями** и **моделями данных**. Функциональная модель представляет с нужной степенью подробности систему активностей, которые в свою очередь отражают свои взаимоотношения через предметы системы. Модели данных дуальны к

функциональным моделям и представляют собой подробное описание предметов системы. Полная методология SADT заключается в построении моделей обеих типов для более точного описания сложной системы. **Однако, в настоящее время широкое применение нашли только функциональные модели.**

Методология SADT представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели объекта какой-либо предметной области. Функциональная модель SADT отображает функциональную структуру объекта, т.е. производимые им действия и связи между этими действиями. Основные элементы этой методологии основываются на следующих концепциях:

- графическое представление блочного моделирования. Графика блоков и дуг SADT-диаграммы отображает функцию в виде блока, а интерфейсы входа/выхода представляются дугами, соответственно входящими в блок и выходящими из него. Взаимодействие блоков друг с другом описываются посредством интерфейсных дуг, выражающих "ограничения", которые в свою очередь определяют, когда и каким образом функции выполняются и управляются;
- строгость и точность. Выполнение правил SADT требует достаточной строгости и точности, не накладывая в то же время чрезмерных ограничений на действия аналитика. Правила SADT включают:
- ограничение количества блоков на каждом уровне декомпозиции (правило 3-6 блоков);
- связность диаграмм (номера блоков);
- уникальность меток и наименований (отсутствие повторяющихся имен);
- синтаксические правила для графики (блоков и дуг);
- разделение входов и управлений (правило определения роли данных).
- отделение организации от функции, т.е. исключение влияния организационной структуры на функциональную модель.

Методология SADT может использоваться для моделирования широкого круга систем и определения требований и функций, а затем для разработки системы, которая удовлетворяет этим требованиям и реализует эти функции. Для уже существующих систем SADT может быть использована для анализа функций, выполняемых системой, а также для указания механизмов, посредством которых они осуществляются.

Результатом применения методологии SADT является модель, которая состоит из диаграмм, фрагментов текстов и глоссария, имеющих ссылки друг на друга. Диаграммы - главные компоненты модели, все функции ИС и интерфейсы на них представлены как блоки и дуги. Место соединения дуги с блоком определяет тип интерфейса. Управляющая информация входит в блок сверху, в то время как информация, которая подвергается обработке, показана с левой стороны блока, а результаты выхода показаны с правой стороны. Механизм (человек или автоматизированная система), который осуществляет операцию, представляется дугой, входящей в блок снизу (рисунок 6.4).

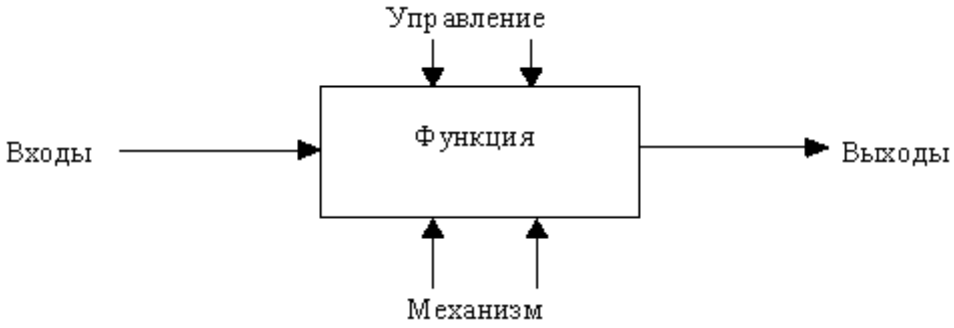


Рисунок 6.4 *Функциональный блок и интерфейсные дуги*

Одной из наиболее важных особенностей методологии SADT является постепенное введение все больших уровней детализации по мере создания диаграмм, отображающих модель.

На рисунке 6.5, где приведены четыре диаграммы и их взаимосвязи, показана структура SADT-модели. Каждый компонент модели может быть декомпозирован на другой диаграмме. Каждая диаграмма иллюстрирует «внутреннее строение» блока на родительской диаграмме.

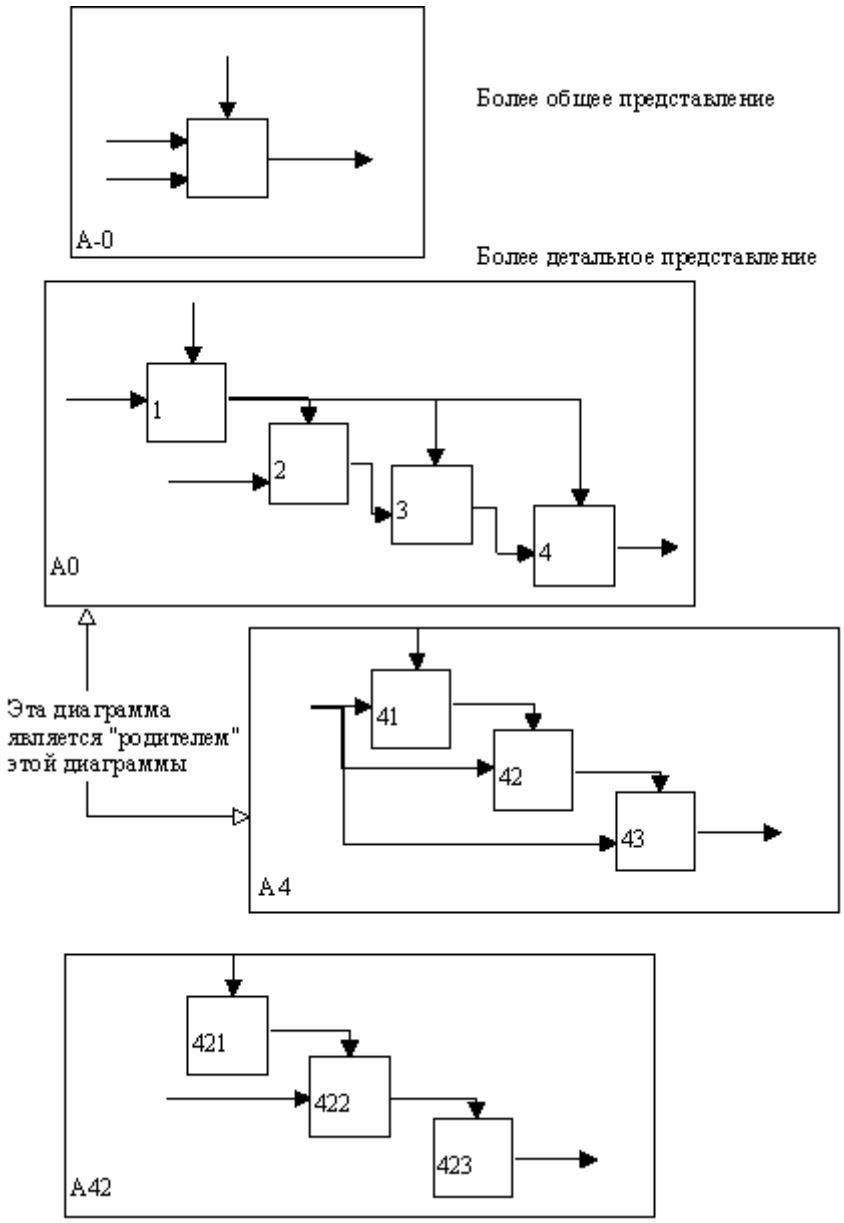


Рисунок 6.5 *Структура SADT-модели. Декомпозиция диаграмм*

6.2 ИЕРАРХИЯ ДИАГРАММ

Построение SADT-модели начинается с представления всей системы в виде простейшей компоненты – одного блока и дуг, изображающих интерфейсы с функциями вне системы. Поскольку единственный блок представляет всю систему как единое целое, имя, указанное в блоке, является общим. Это верно и для интерфейсных дуг – они также представляют полный набор внешних интерфейсов системы в целом.

Затем блок, который представляет систему в качестве единого модуля, детализируется на другой диаграмме с помощью нескольких блоков, соединенных интерфейсными дугами. Эти блоки представляют основные подфункции исходной функции. Данная декомпозиция выявляет полный набор подфункций, каждая из которых представлена как блок, границы которого определены интерфейсными дугами. Каждая

из этих подфункций может быть декомпозирована подобным образом для более детального представления.

Во всех случаях каждая подфункция может содержать только те элементы, которые входят в исходную функцию. Кроме того, модель не может опустить какие-либо элементы, т.е., как уже отмечалось, родительский блок и его интерфейсы обеспечивают контекст. К нему нельзя ничего добавить, и из него не может быть ничего удалено.

Модель SADT представляет собой серию диаграмм с сопроводительной документацией, разбивающих сложный объект на составные части, которые представлены в виде блоков. Детали каждого из основных блоков показаны в виде блоков на других диаграммах. Каждая детальная диаграмма является декомпозицией блока из более общей диаграммы. На каждом шаге декомпозиции более общая диаграмма называется родительской для более детальной диаграммы.

Дуги, входящие в блок и выходящие из него на диаграмме верхнего уровня, являются точно теми же самыми, что и дуги, входящие в диаграмму нижнего уровня и выходящие из нее, потому что блок и диаграмма представляют одну и ту же часть системы.

На рисунках 6.6 – 6.8 представлены различные варианты выполнения функций и соединения дуг с блоками.

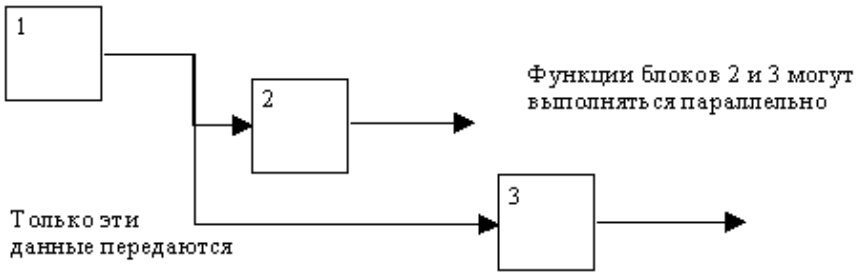
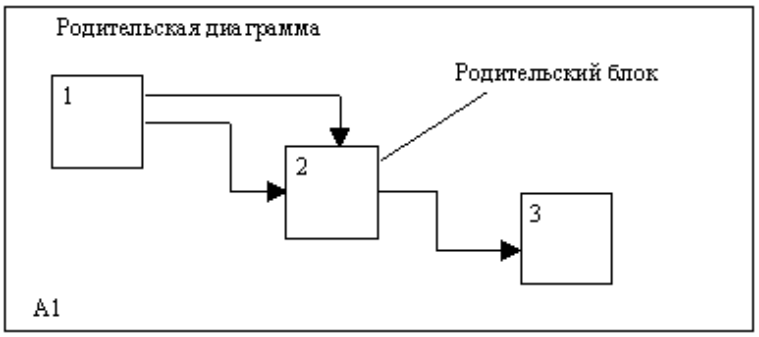


Рисунок 6.6. *Одновременное выполнение*



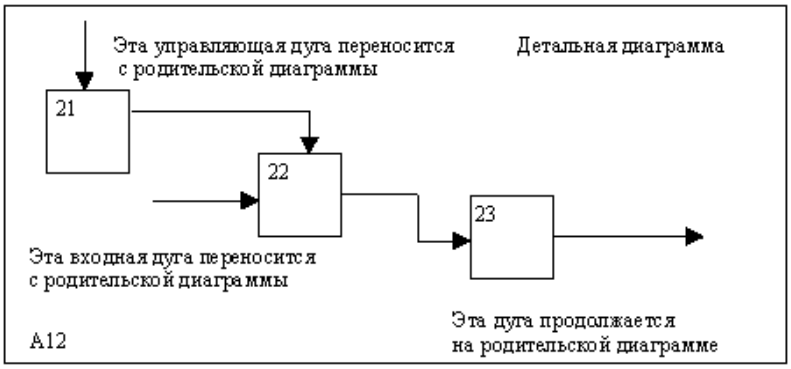


Рисунок 6.7 *Соответствие должно быть полным и непротиворечивым*

Некоторые дуги присоединены к блокам диаграммы обоими концами, у других же один конец остается неприсоединенным. Неприсоединенные дуги соответствуют входам, управлениям и выходам родительского блока. Источник или получатель этих пограничных дуг может быть обнаружен только на родительской диаграмме. Неприсоединенные концы должны соответствовать дугам на исходной диаграмме. Все граничные дуги должны продолжаться на родительской диаграмме, чтобы она была полной и непротиворечивой.

На SADT-диаграммах не указаны явно ни последовательность, ни время. Обратные связи, итерации, продолжающиеся процессы и перекрывающиеся (по времени) функции могут быть изображены с помощью дуг. Обратные связи могут выступать в виде комментариев, замечаний, исправлений и т.д. (рисунок 6.8).

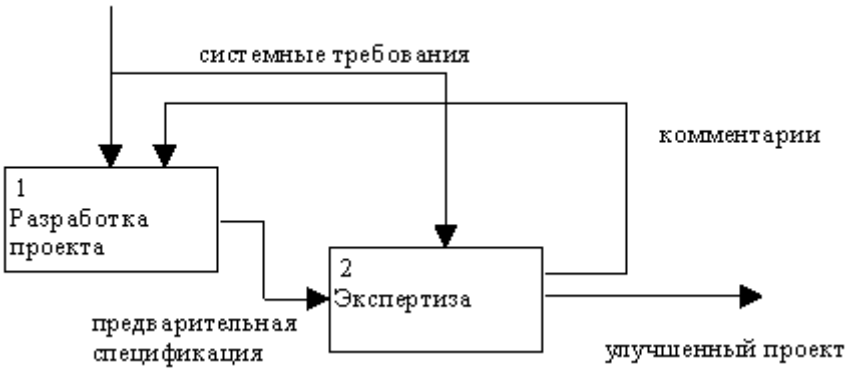


Рисунок 6.8 Пример обратной связи

Как было отмечено, механизмы (дуги с нижней стороны) показывают средства, с помощью которых осуществляется выполнение функций. Механизм может быть человеком, компьютером или любым другим устройством, которое помогает выполнять данную функцию (рисунок 6.9).

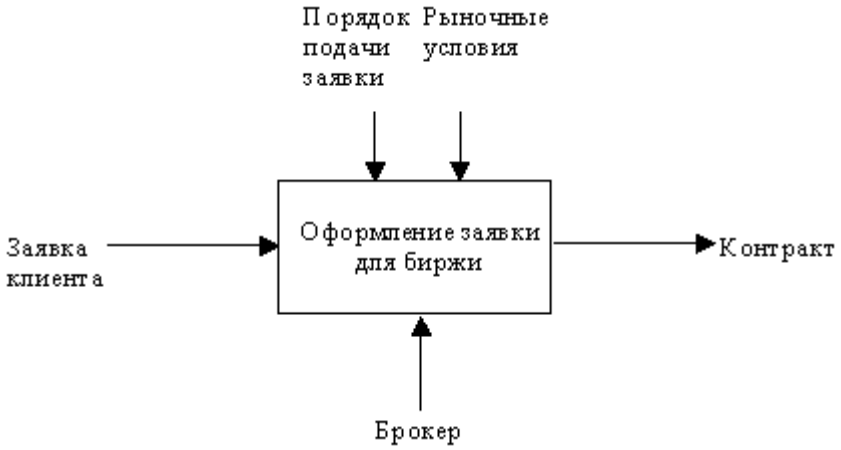


Рисунок 6.9 *Пример механизма*

Каждый блок на диаграмме имеет свой номер. Блок любой диаграммы может быть далее описан диаграммой нижнего уровня, которая, в свою очередь, может быть далее детализирована с помощью необходимого числа диаграмм. Таким образом, формируется иерархия диаграмм.

Для того, чтобы указать положение любой диаграммы или блока в иерархии, используются номера диаграмм. Например, `A21` является диаграммой, которая детализирует блок `1` на диаграмме `A2`. Аналогично, `A2` детализирует блок `2` на диаграмме A0, которая является самой верхней диаграммой модели. На рисунке 6.10 показано типичное дерево диаграмм.

Рисунок 6.10 *Иерархия диаграмм*

6.3 СИНТАКСИС ДИАГРАММ

Как уже отмечалось, каждая SADT-диаграмма содержит блоки и дуги. Блоки изображают функции системы, дуги связывают блоки и отображают взаимодействия и взаимосвязи между ними. Диаграмме дается название, которое располагается внизу.

Функциональные блоки изображаются прямоугольниками. Поскольку блоки отображают функции системы, то в названии блоков используют глаголы или глагольные обороты (рассчитать, выполнить задание и т.д.).

Блоки на SADT-диаграмме никогда не размещаются случайным образом. Они размещаются по степени важности, как ее понимает автор диаграммы. В SADT этот относительный порядок называется ***доминированием***. Доминирование понимается как влияние, которое один блок оказывает на другие блоки диаграммы. Например, самым доминирующим блоком диаграммы может быть первый из требуемой последовательности функций. Наиболее доминирующий блок обычно располагается в верхнем левом углу диаграммы, а наименее доминирующий – в правом нижнем углу диаграммы. В результате имеем ступенчатую схему.

Блоки в SADT должны быть пронумерованы. Номера блоков служат однозначными идентификаторами для системных функций и автоматически организуют эти функции в иерархию модели.

Дуги на SADT-диаграмме изображаются одинарными линиями со стрелками на концах. Дуги изображают объекты (или данные), поэтому они описываются существительными или существительными с определениями (набор инструментов, чертеж и т.д.)

Между объектами и функциями возможны 4 отношения: *вход, управление, выход и механизм*. Каждое из этих отношений изображается дугой и связано с определенной стороной блока (левая сторона – входные дуги, правая сторона – выходные дуги, верхняя сторона – управляющие дуги, нижняя сторона – дуги механизма).

Входные дуги изображают объекты, используемые и преобразуемые функциями. Управленческие дуги представляют информацию, управляющую действиями функций. Обычно управляющие дуги несут информацию, которая указывает, что должна выполнять функция. Выходные дуги изображают объекты, в которые преобразуются входы. Дуги механизмов отражают, по крайней мере частично, как функции реализуются, с помощью чего или кого.

Таким образом, SADT-диаграмма составлена из блоков, связанных дугами, которые определяют, как блоки влияют друг на друга. Это влияние может выражаться либо в передаче выходной информации к другой функции для дальнейшего преобразования, либо в выработке управляющей информации, предписывающей, что именно должна выполнять другая функция. Поэтому SADT-диаграммы нельзя отнести к блок-схемам или диаграммам потоков данных. Скорее всего, это предписывающие диаграммы, представляющие входные-выходные преобразования и указывающие правила этих преобразований.

В методологии SADT требуется только пять типов взаимосвязей между блоками для описания их отношений: *управление, вход, обратная связь по управлению, обратная связь по входу, выход-механизм*. Рассмотрим на примере.

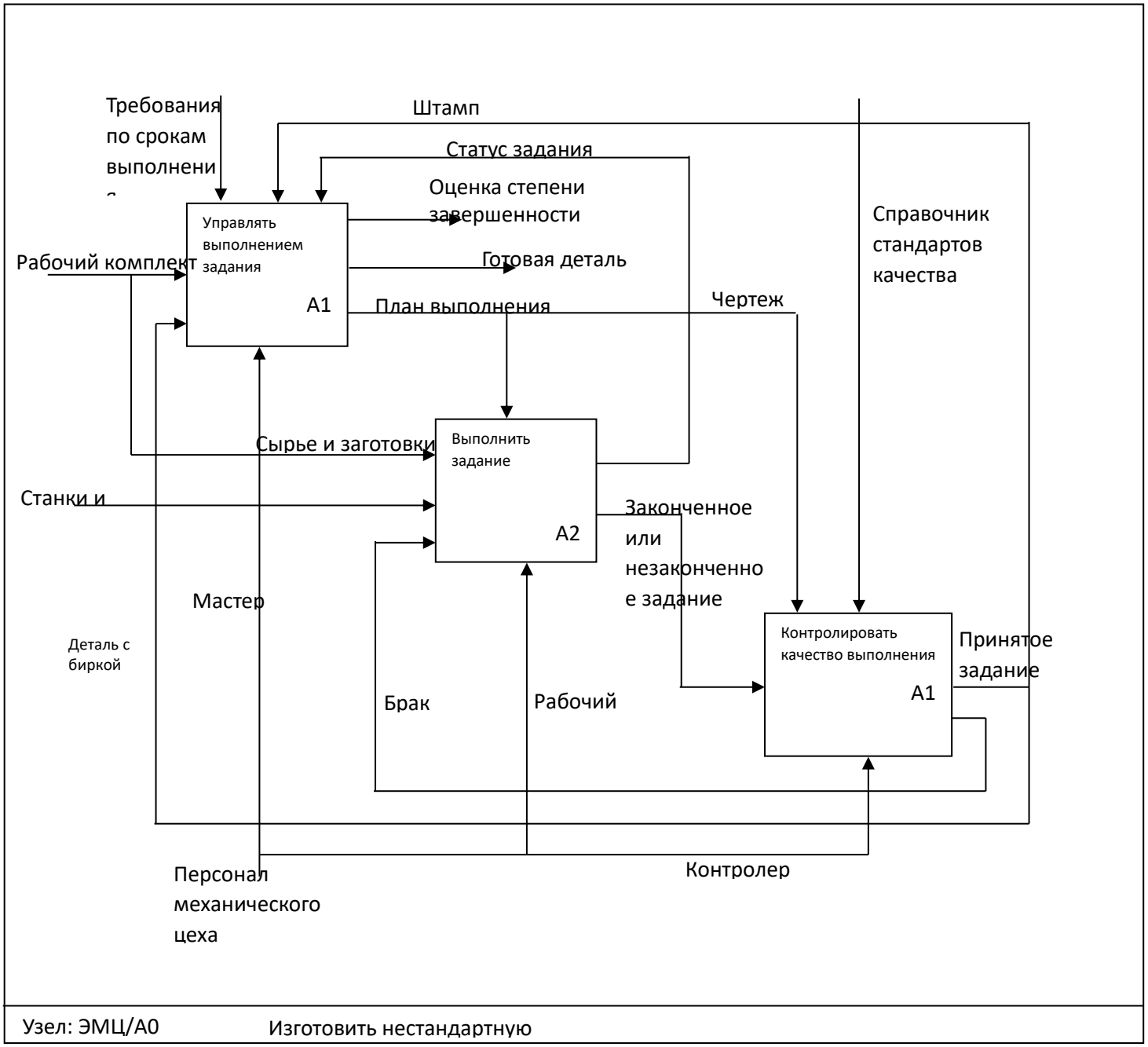


Рисунок 6.11 *SADT-диаграмма*

Одна SADT-диаграмма сложна сама по себе, поскольку содержит от трех до шести блоков, связанных множеством дуг. Для адекватного описания системы требуется несколько таких диаграмм. Диаграммы, собранные и связанные вместе, становятся SADT-моделью. В SADT дополнительно к синтаксису диаграмм существуют правила синтаксиса моделей.

SADT-модель является иерархически организованной совокупностью диаграмм, состоящей из 3-6 блоков. При этом каждый блок может пониматься как отдельный тщательно определенный объект. Каждый блок в SADT рассматривается как формальная

граница некоторой части целой системы. Другими словами, блок и касающиеся его дуги определяют точную границу диаграммы, представляющей декомпозицию этого блока.

Принцип ограничения объекта встречается на каждом уровне. Один блок и несколько дуг на самом верхнем уровне используется для определения границы всей системы. Диаграмма, состоящая из одного блока и его дуг, определяет границу системы и называется контекстной диаграммой модели.

Кроме этого на контекстной диаграмме отображается цель системы и точка зрения.

Понятие цели системы.

SADT-модель дает полное, точное и адекватное описание системы, имеющее конкретное назначение. Это назначение называют целью системы. Таким образом, целью модели является получение ответов на совокупность вопросов. Если модель отвечает не на все вопросы или ее ответы недостаточно точны, то говорят о том, что модель не достигла своей цели. Определяя модель таки образом, SADT закладывает основы практического моделирования.

Точка зрения модели.

С определением модели тесно связана позиция, с которой наблюдается система и создается ее модель. Эта позиция и называется «точкой зрения» данной модели. «Точку зрения» лучше всего представлять себе как место (позицию) человека или объекта, в которое надо встать, чтобы увидеть систему в действии.

Построим контекстную диаграмму модели изготовления нестандартной детали.

Определим для начала цель и точку зрения модели.

Цель: *определить, какие функции должны быть включены в процесс изготовления нестандартной детали и как эти функции взаимосвязаны между собой*.

Точка зрения: *лучше всего описать все функции может начальник цеха, в котором изготавливаются нестандартные детали*.

Итак.

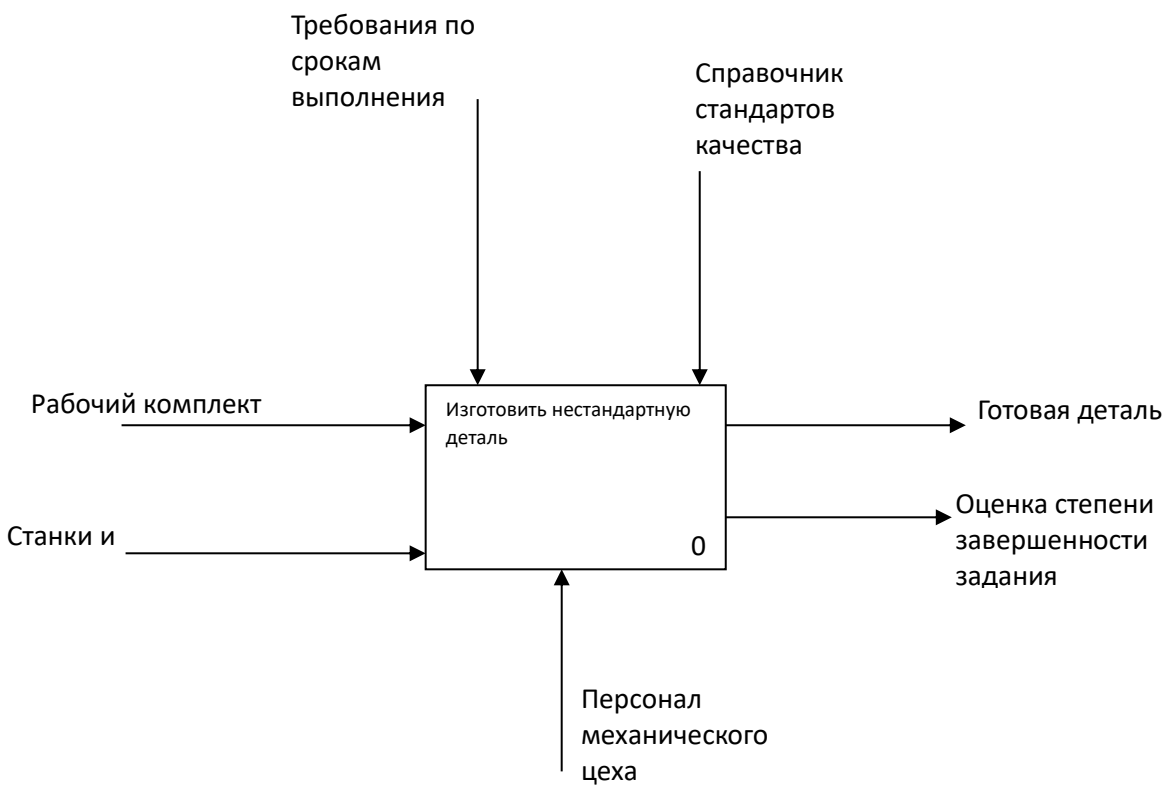


Рисунок 6.12 *Контекстная диаграмма*

SADT-модели развиваются в процессе структурной декомпозиции сверху вниз. Сначала декомпозируется один блок, являющийся границей модели. Название диаграммы совпадает с названием декомпозируемого блока. В методологии SADT идентифицируется каждая диаграмма данной модели посредством того, что называется «номер узла». Номер узла для контекстной диаграммы имеет следующий вид: название модели или аббревиатура, косая черта, заглавная буква A (activity в функциональных диаграммах), дефис и ноль.

Номером узла диаграммы, декомпозирующей контекстную диаграмму, является тот же номер узла, но без дефиса.

Среди современных методов визуального моделирования, пожалуй, самым широко распространенным является RUP/USDP – промышленный метод создания ПО, использующий UML практически на всех стадиях и во всех видах деятельности разработки. RUP/USDP является тяжеловесным методом применения UML: он содержит множество предписаний, непростую последовательность шагов, определяет разные роли участников, охватывает все стадии разработки ПО. Его внедрение в процесс компании требует значительных затрат и существенной перестройки принципов ее работы. Существуют и легковесные методы применения UML, которые не имеют жестких предписаний и допускают вариативность при использовании. Примером может служить метод случаев использования, применяемый для выявления и первичной формализации требований к программной системе. Это метод будет описан в следующих лекциях, посвященных UML. Наконец, специализированные программные инструменты позволяют удобно работать с визуальными языками и пользоваться тем или иным методом их применения. Это прежде всего графические редакторы, а также средства валидации моделей, генераторы конечного кода по диаграммам и т. д. О программных инструментах. Средства, реализующие языки и методы визуального моделирования, бывают двух видов – универсальные и предметно-ориентированные. Универсальные инструменты являются коробочными и многофункциональными пакетами, предназначенными для анализа и проектирования ПО «вообще», то есть без какой-либо специализированной ориентации. Как правило, сегодня такие пакеты строятся на базе языка UML и называются CASE-пакетами. Самыми известными CASE-пакетами являются IBM Rational Rose, Borland Together, Telelogic Tau, Microsoft Visio/UML Add-on. Эти средства поддерживают различные виды диаграмм, удобную среду их разработки с такими функциями, как печать и копирование диаграмм, различные способы редактирования графических символов, средства просмотра и поиска в визуальной модели, различные режимы отображения диаграмм и многое другое. Они также обеспечивают генерацию программного кода в разные целевые платформы программирования, версионный контроль визуальных моделей, часто являются кросс-платформенными (например, работают под управлением операционных систем Windows и Linux), обеспечивают интеграционные «мосты» с другими средствами разработки ПО, например, со средствами управления требованиями. Как правило, все современные CASE-пакеты имеют открытые программные интерфейсы и позволяют расширять свою базовую функциональность.

Термин CASE (Computer Aided Software Engineering) появился в индустрии разработки ПО в начале 1980-х годов. Довольно быстро он стал обозначать графические средства анализа и проектирования ПО, отражая надежды и упования создать на основе визуального моделирования универсальный процесс разработки ПО. Пик развития этих средств приходится на начало 1990-х годов, когда они стали использоваться на базе платформы IBM Mainframe для автоматизации бизнеса крупных компаний. CASE-системы предоставляли мощные средства генерации кода, являясь не только инструментами анализа и проектирования, но и средами разработки ПО. CASE-средства интегрировали многообразные и разрозненные средства разработки под

Mainframe-платформой – инструменты разработки пользовательского интерфейса и баз данных, средства взаимодействия основного приложения с операционной системой и пр. Типичное крупное Mainframe-приложение состояло из тестов примерно на 3-5 разных языках программирования, для которых существовало (и активно использовалось в других приложениях) множество альтернативных вариаций. Одна из самых известных систем такого рода – ADW (Application Developing Workbench). В итоге было разработано много промышленных информационных систем с использованием этих CASE-средств, успешно работающих и по сей день. В результате данные CASE-системы, многие из которых сменили не по одной компании-хозяину, до сих пор поддерживаются и развиваются, чтобы созданные на их основе информационные системы могли успешно функционировать и модернизироваться под современные бизнес-потребности. Почти все подобные CASE-системы и, в частности, ADW, в настоящее время куплены компанией Computer Associates International. С середины 1990-х годов, в связи с прекращением распространения Mainframe-платформ, развитие этих CASE-систем прекратилось, и стали появляться CASE-системы для персональных компьютеров. Их эволюция происходила и продолжает происходить уже по иному сценарию. Современные CASE-пакеты не являются комплексными средами разработки, а заняли нишу средств анализа и проектирования, и в основном используются без средств кодогенерации, а лишь как инструменты для построения проектных спецификаций.

Предметно-ориентированные (domain-specific) программные инструменты поддержки визуального моделирования предназначены для определенных областей разработки ПО и тоже могут быть коробочными, как, например, пакет WebRatio для моделирования web-приложений. Однако предметно-ориентированные инструменты могут создаваться и отдельными компаниями для своих собственных проектов, особенно в рамках линеек программных продуктов (product lines). Это особенно удобно, поскольку во- первых, такие средства могут хорошо решать задачи именно того процесса, той компании, для которых они создаются. А во-вторых, сейчас на рынке имеются развитые среды для разработки средств визуального моделирования, самые известные из которых – Microsoft Visio, Microsoft DSL Tools и Eclipse/GMF. Эти и другие пакеты делают задачу создания собственного графического редактора посильной для обычных, рядовых компаний-разработчиков. Предметно-ориентированное визуальное моделирование будет подробно рассмотрено в следующих лекциях. Визуальное моделирование на фоне эволюции средств программирования. Идея автоматической генерации программного кода по визуальным моделям понятна и притягательна. Диаграммы являются более близкими к предметной области, чем программный код, понятны инженерам, менеджерам, заказчикам и т.д. Долгое время считалось, что визуальное моделирование является следующим шагом эволюции средств программирования, вслед за алгоритмическими языками высокого уровня (см. рис. 6.13). Кратко рассмотрим эволюцию средств программирования. Сначала программировали в кодах целевых ЭВМ. Машине подавалось на вход в точности то, что она исполняла в качестве целевой задачи. Не было никакой промежуточной обработки вводимой информации.

Рисунок 6.13 *Эволюция средств программирования*

При визуальном моделировании программного обеспечения используются следующие уровни абстракции:

- предметная область;
- модель;
- метамодель;
- метаметамодель.

Для визуального моделирования в качестве предметной области (domain) обычно выступают:

- тот фрагмент действительности, куда создаваемое ПО будет встроено: бизнес-процессы компании, для которой создается информационная система, электромеханическая среда для встроенного ПО и т. д.; программистам необходимо тщательно изучить тот контекст, в котором их ПО будет работать, чтобы оно было там адекватно;
- архитектурные решения ПО, которые должны быть тщательно проработаны, обсуждены с разными специалистами и ими понятны; с этой целью они и подвергаются визуализации.

Модель (model) – это упрощенное описание предметной области, созданное для удобства выполнения там действий, работы. Более простая модель дает возможность не рассматривать все бесконечное многообразие предметной области, а сосредоточиться лишь на некоторых ее свойствах. Например, для создания информационной системы автоматизации предприятия строится модель предприятия, которая фокусируется на бизнес-процессах, потоках данных, бизнес-ролях. В эту модель не входит следующая информация о предприятии: межличностные отношения сотрудников, детали

планировки помещений офисов, расписание работы компании (начало работы, обеденный перерыв, выходные) и т. д.

При визуальном моделировании ПО обычно строятся следующие модели.

- *Модели анализа* (*analysis models*), формализующие результаты изучения программистами того контекста, где будет работать их будущее ПО; эти модели позволяют хорошо формализовать требования к ПО, согласовать их с будущими пользователями системы, заказчиком и др. заинтересованными лицами, тем самым, создав хорошую основу для дальнейшей разработки программной системы.
- *Модели проектирования* (*design models*), в которых фиксируются архитектурные решения будущего ПО – его структура, внешние и внутренние интерфейсы, принципиальные вопросы реализации с учетом средств разработки, платформ исполнения и т.д.

Модели анализа должны «плавно» переходить в модели проектирования, и это является одним из главных принципов модельно-ориентированного подхода к разработке ПО. В индустриальном производстве создание той или иной модели – это не единичный прецедент. Например, люди, специализирующиеся на разработке информационных систем, создают много моделей разных компаний. Соответственно, у них возникает потребность в специальном языке, который существенно упростил бы разработку таких моделей. Этот язык должен содержать описание всех тех абстракций, которые обычно нужны при моделировании деятельности предприятий. Само множество этих моделей оказывается предметной областью для новой модели, которую поэтому естественно называть метамоделью (*metamodel*). В рамках одной области деятельности может быть востребовано много разных модельных языков, и тогда необходим общий способ по их разработке и спецификации. В этом случае оказывается востребованным язык описания языков (метамоделей) – метаметамодель (*meta-metamodel*). Предметной областью для этой новой модели являются соответствующие метамодели. Теоретически, приведенную выше цепочку метауровней можно продолжать бесконечно. Каждый следующий уровень будет служить моделью для предыдущего, а предыдущий уровень оказывается для него предметной областью, как показано на рис. 6.14. Переход на следующий метауровень целесообразен лишь тогда, когда на некотором уровне появляется много сходных объектов, нуждающихся в структурировании, а, значит, в метаописании. В какой-то момент будет достигнут предел по количеству объектов, требующих унификации и упорядочивания. На рис. 2.2 приведен пример четырех метауровней с кратким обоснованием, почему пятый уровень и далее не нужны.

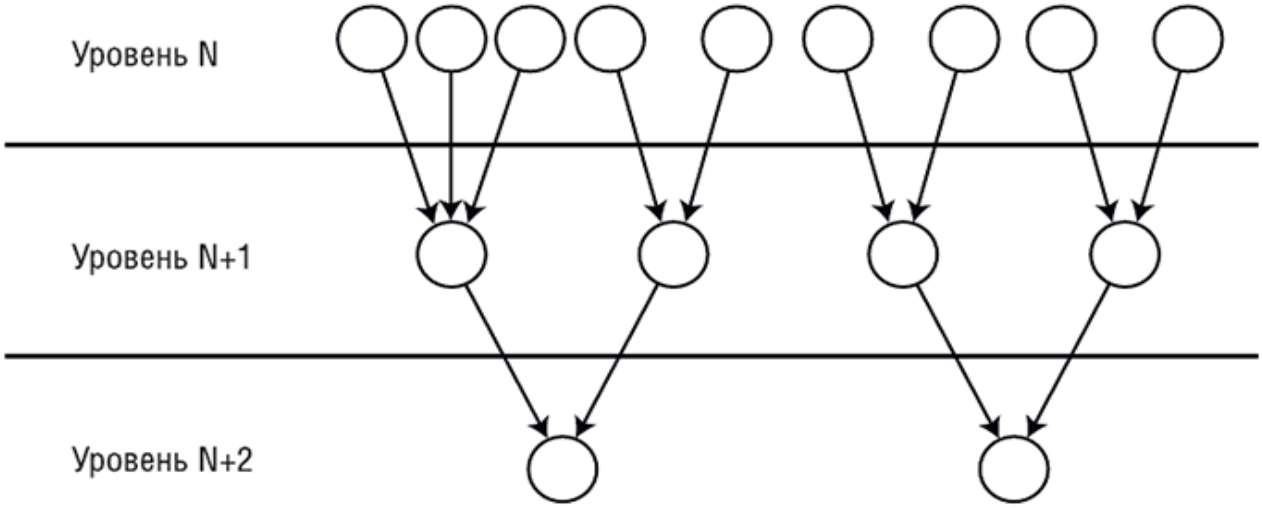


Рисунок 6.14 Уровни метамоделирования

Рассмотрим подробнее.

1. *Предметная область* – некоторая программная система, ее функции и пользователи. Пользователей у системы могут быть десятки, сотни и даже тысячи, функциональность может быть очень сложной. Очевидно, что необходима специальная модель, структурирующая все это разнообразие.

2. *Модель в данном случае* – это одна или несколько диаграмм случаев использования, классифицирующих и описывающих функции системы и ее пользователей. Пользователи сгруппированы по типам, функциональность – по случаям использования (см. следующую лекцию, где этот тип диаграмм будет описываться подробно). Очевидно, что разработчикам ПО приходится часто строить такие модели для разных систем.

3. Поэтому необходима *метамодель*, описывающая язык случаев использования. В данном случае, в упрощенном варианте она состоит из актера и случая использования, соединенных между собой связью многие ко многим – один актер может быть связан с несколькими случаями использования, несколько актеров могут быть связаны с одним случаем использования. Очевидно, что подобных метамоделей можно составить множество – для других визуальных языков.

4. *Метаметамодель* – это язык для создания метамоделей всех визуальных языков. В данном, упрощенном случае она состоит из класса и ассоциации.

5. Попытка построить *метаметаметамодель* приводит к забавному противоречию – получается ровно такая же диаграмма, как на предыдущем уровне (попробуйте – увидите сами!). Это происходит потому, что метаметамодель (п. 4) также описана с помощью некоторого визуального языка. А раз так, то этот новый язык тоже описываться средствами метаметамодели (см. определение метамодели в п. 4 – ведь она подходит для описания всех визуальных языков).

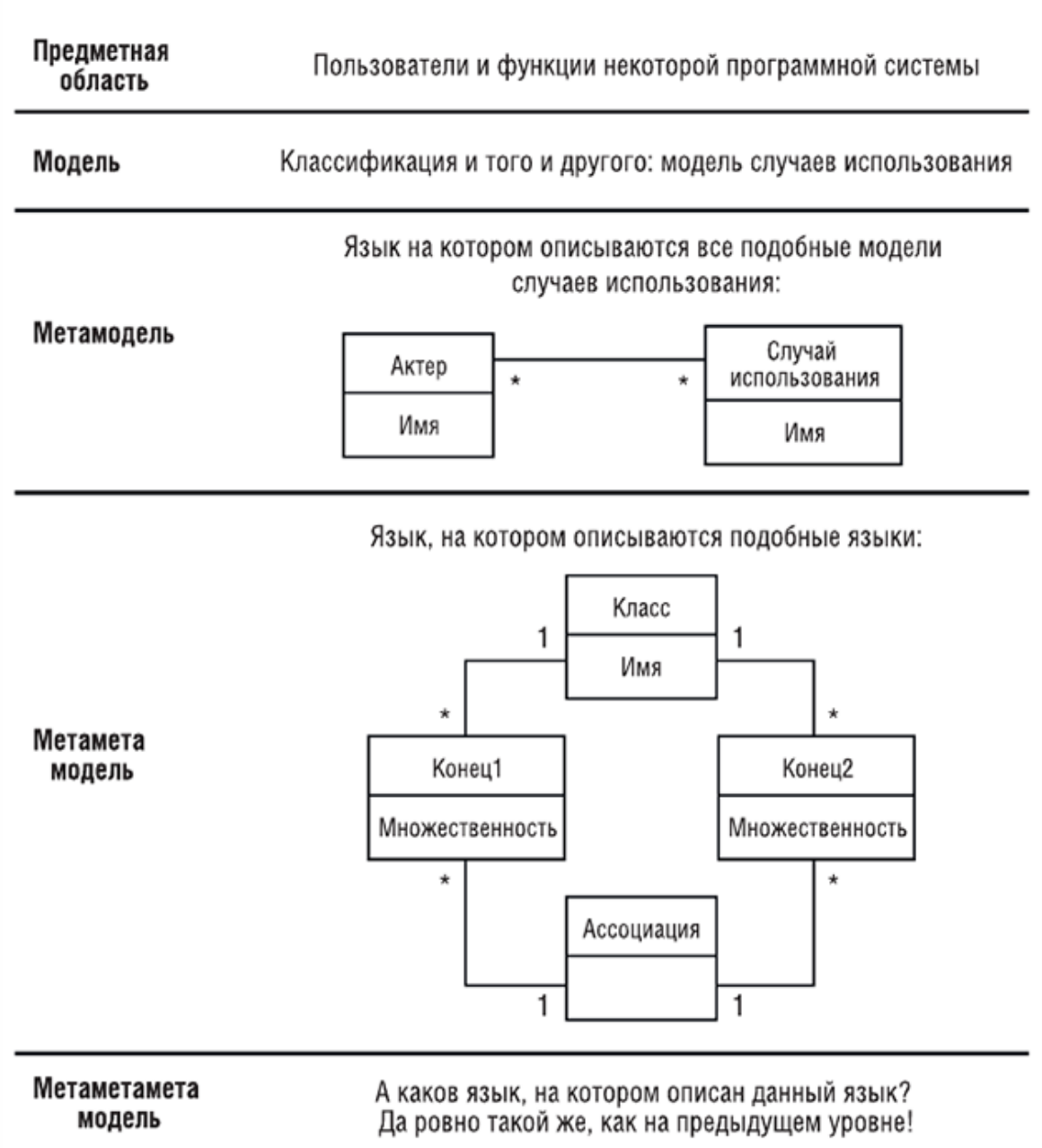


Рисунок 6.15 *Пример четырех метауровней уровней в визуальном моделировании*

Есть и объективная предпосылка к тому, что пятый уровень оказывается вырожденным, безотносительно тому, какие средства используются для создания метаметамодели. Дело в том, что не существует большого множества средств для задания метамоделей визуальных языков. И поэтому не возникает задачи по структурированию и упорядочиванию таких способов путем разработки для них общей модели. То есть метаметаметамодель не требуется…

6.4 Состав функциональной модели

Результатом применения метода SADT является модель, которая состоит из диаграмм, фрагментов текстов и глоссария, имеющих ссылки друг на друга. Диаграммы – главные компоненты модели, все функции организации и интерфейсы на них

представлены как блоки и дуги соответственно. Место соединения дуги с блоком определяет тип интерфейса. Управляющая информация входит в блок сверху, в то время как входная информация, которая подвергается обработке, показана с левой стороны блока, а результаты (выход) показаны с правой стороны. Механизм (человек или автоматизированная система), который осуществляет операцию, представляется дугой, входящей в блок снизу

Функциональные методики, наиболее известной из которых является методика IDEF, рассматривают организацию как набор функций, преобразующий поступающий поток информации в выходной поток. Процесс преобразования информации потребляет определенные ресурсы. Основное отличие от объектной методики заключается в четком отделении функций (методов обработки данных) от самих данных.

С точки зрения бизнес-моделирования каждый из представленных подходов обладает своими преимуществами. Объектный подход позволяет построить более устойчивую к изменениям систему, лучше соответствует существующим структурам организации. Функциональное моделирование хорошо показывает себя в тех случаях, когда организационная структура находится в процессе изменения или вообще слабо оформлена. Подход от выполняемых функций интуитивно лучше понимается исполнителями при получении от них информации об их текущей работе.

6.4.1 Функциональные модели IDEF0

Методологию IDEF0 можно считать следующим этапом развития хорошо известного графического языка описания функциональных систем SADT (Structured Analysis and Design Technique). Исторически IDEF0 как стандарт был разработан в 1981 году в рамках обширной программы автоматизации промышленных предприятий, которая носила обозначение *ICAM (Integrated Computer Aided Manufacturing)*. Семейство стандартов IDEF унаследовало свое обозначение от названия этой программы (*IDEF=Icam DEFinition*), и последняя его редакция была выпущена в декабре 1993 года Национальным Институтом по Стандартам и Технологиям США (NIST).

Целью методики является построение функциональной схемы исследуемой системы, описывающей все необходимые процессы с точностью, достаточной для однозначного моделирования деятельности системы.

В основе методологии лежат четыре основных понятия: *функциональный блок, интерфейсная дуга, декомпозиция, глоссарий*.

Функциональный блок (Activity Box) представляет собой некоторую конкретную функцию в рамках рассматриваемой системы. По требованиям стандарта название каждого функционального блока должно быть сформулировано в глагольном наклонении (например, «производить услуги»). На диаграмме функциональный блок изображается прямоугольником (рис. 6.16). Каждая из четырех сторон функционального блока имеет свое определенное значение (роль), при этом:

- верхняя сторона имеет значение "Управление" (*Control*);
- левая сторона имеет значение "Вход" (*Input*);
- правая сторона имеет значение "Выход" (*Output*);

- нижняя сторона имеет значение "Механизм" (*Mechanism*).

Рисунок 6.16 *Функциональный блок*

Интерфейсная дуга (Arrow) отображает элемент системы, который обрабатывается функциональным блоком или оказывает иное влияние на функцию, представленную данным функциональным блоком. Интерфейсные дуги часто называют потоками или стрелками.

С помощью интерфейсных дуг отображают различные объекты, в той или иной степени определяющие процессы, происходящие в системе. Такими объектами могут быть элементы реального мира (детали, вагоны, сотрудники и т.д.) или потоки данных и информации (документы, данные, инструкции и т.д.).

В зависимости от того, к какой из сторон функционального блока подходит данная интерфейсная дуга, она носит название «входящей», «исходящей» или «управляющей».

Необходимо отметить, что любой функциональный блок по требованиям стандарта должен иметь, по крайней мере, одну управляющую интерфейсную дугу и одну исходящую. Это и понятно – каждый процесс должен происходить по каким-то правилам (отображаемым управляющей дугой) и должен выдавать некоторый результат (выходящая дуга), иначе его рассмотрение не имеет никакого смысла.

Обязательное наличие управляющих интерфейсных дуг является одним из главных отличий стандарта `IDEF0` от других методологий классов *`DFD` (Data Flow Diagram)* и *`WFD` (Work Flow Diagram)*.

Декомпозиция (Decomposition) является основным понятием стандарта `IDEF0`. Принцип декомпозиции применяется при разбиении сложного процесса на составляющие его функции. При этом уровень детализации процесса определяется непосредственно разработчиком модели.

Декомпозиция позволяет постепенно и структурировано представлять модель системы в виде иерархической структуры отдельных диаграмм, что делает ее менее перегруженной и легко усваиваемой.

Последним из понятий `IDEF0` является глоссарий (*Glossary*). Для каждого из элементов `IDEF0` – диаграмм, функциональных блоков, интерфейсных дуг – существующий стандарт подразумевает создание и поддержание набора соответствующих определений, ключевых слов, повествовательных изложений и т.д., которые характеризуют объект, отображенный данным элементом. Этот набор называется глоссарием и является описанием сущности данного элемента. Глоссарий

гармонично дополняет наглядный графический язык, снабжая диаграммы необходимой дополнительной информацией.

Модель IDEF0 всегда начинается с представления системы как единого целого – одного функционального блока с интерфейсными дугами, простирающимися за пределы рассматриваемой области. Такая диаграмма с одним функциональным блоком называется контекстной диаграммой.

В пояснительном тексте к контекстной диаграмме должна быть указана цель (*Purpose*) построения диаграммы в виде краткого описания и зафиксирована точка зрения (*Viewpoint*).

Определение и формализация цели разработки IDEF0-модели является крайне важным моментом. Фактически цель определяет соответствующие области в исследуемой системе, на которых необходимо фокусироваться в первую очередь.

Точка зрения определяет основное направление развития модели и уровень необходимой детализации. Четкое фиксирование точки зрения позволяет разгрузить модель, отказавшись от детализации и исследования отдельных элементов, не являющихся необходимыми, исходя из выбранной точки зрения на систему. Правильный выбор точки зрения существенно сокращает временные затраты на построение конечной модели.

Выделение подпроцессов. В процессе декомпозиции функциональный блок, который в контекстной диаграмме отображает систему как единое целое, подвергается детализации на другой диаграмме. Получившаяся диаграмма второго уровня содержит функциональные блоки, отображающие главные подфункции функционального блока контекстной диаграммы, и называется дочерней (*Child Diagram*) по отношению к нему (каждый из функциональных блоков, принадлежащих дочерней диаграмме, соответственно называется дочерним блоком – *Child Box*). В свою очередь, функциональный блок — предок называется родительским блоком по отношению к дочерней диаграмме (*Parent Box*), а диаграмма, к которой он принадлежит – родительской диаграммой (*Parent Diagram*). Каждая из подфункций дочерней диаграммы может быть далее детализирована путем аналогичной декомпозиции соответствующего ей функционального блока. В каждом случае декомпозиции функционального блока все интерфейсные дуги, входящие в данный блок или исходящие из него, фиксируются на дочерней диаграмме. Этим достигается структурная целостность IDEF0-модели.

Иногда отдельные интерфейсные дуги высшего уровня не имеет смысла продолжать рассматривать на диаграммах нижнего уровня, или наоборот – отдельные дуги нижнего отражать на диаграммах более высоких уровней – это будет только перегружать диаграммы и делать их сложными для восприятия. Для решения подобных задач в стандарте IDEF0 предусмотрено понятие *туннелирования*. Обозначение «туннеля» (*Arrow Tunnel*) в виде двух круглых скобок вокруг начала интерфейсной дуги обозначает, что эта дуга не была унаследована от функционального родительского блока и появилась (из «туннеля») только на этой диаграмме. В свою очередь, такое же обозначение вокруг конца (стрелки) интерфейсной дуги в непосредственной близи от

блока–приемника означает тот факт, что в дочерней по отношению к этому блоку диаграмме эта дуга отображаться и рассматриваться не будет. Чаще всего бывает, что отдельные объекты и соответствующие им интерфейсные дуги не рассматриваются на некоторых промежуточных уровнях иерархии, – в таком случае они сначала «погружаются в туннель», а затем при необходимости «возвращаются из туннеля».

Обычно IDEF0-модели несут в себе сложную и концентрированную информацию, и для того, чтобы ограничить их перегруженность и сделать удобочитаемыми, в стандарте приняты соответствующие ограничения сложности.

Рекомендуется представлять на диаграмме от трех до шести функциональных блоков, при этом количество подходящих к одному функциональному блоку (выходящих из одного функционального блока) интерфейсных дуг предполагается не более четырех.

Стандарт IDEF0 содержит набор процедур, позволяющих разрабатывать и согласовывать модель большой группой людей, принадлежащих к разным областям деятельности моделируемой системы. Обычно процесс разработки является итеративным и состоит из следующих условных этапов:

- Создание модели группой специалистов, относящихся к различным сферам деятельности предприятия. Эта группа в терминах IDEF0 называется авторами (*Authors*). Построение первоначальной модели является динамическим процессом, в течение которого авторы опрашивают компетентных лиц о структуре различных процессов, создавая модели деятельности подразделений. При этом их интересуют ответы на следующие вопросы:
  Что поступает в подразделение «на входе»?
  - Какие функции и в какой последовательности выполняются в рамках подразделения?
  - Кто является ответственным за выполнение каждой из функций?
  - Чем руководствуется исполнитель при выполнении каждой из функций?
  - Что является результатом работы подразделения (на выходе)?

  На основе имеющихся положений, документов и результатов опросов создается черновик (*Model Draft*) модели.
- Распространение черновика для рассмотрения, согласований и комментариев. На этой стадии происходит обсуждение черновика модели с широким кругом компетентных лиц (в терминах IDEF0 – читателей) на предприятии. При этом каждая из диаграмм черновой модели письменно критикуется и комментируется, а затем передается автору. Автор, в свою очередь, также письменно соглашается с критикой или отвергает ее с изложением логики принятия решения и вновь возвращает откорректированный черновик для дальнейшего рассмотрения. Этот цикл продолжается до тех пор, пока авторы и читатели не придут к единому мнению.
- Официальное утверждение модели. Утверждение согласованной модели происходит руководителем рабочей группы в том случае, если у авторов модели и читателей отсутствуют разногласия по поводу ее адекватности. Окончательная

модель представляет собой согласованное представление о предприятии (системе) с заданной точки зрения и для заданной цели.

Наглядность графического языка IDEF0 делает модель вполне читаемой и для лиц, которые не принимали участия в проекте ее создания, а также эффективной для проведения показов и презентаций. В дальнейшем на базе построенной модели могут быть организованы новые проекты, нацеленные на производство изменений в модели.

Помимо модели IDEF0 существуют и другие:

- *IDEF1* – Information Modeling - методология моделирования информационных потоков внутри системы, позволяющая отображать и анализировать их структуру и взаимосвязи;
- *IDEF1X* (*IDEF1* Extended) – *Data Modeling* - методология проектирования реляционных *баз данных*. Заключается в построении моделей данных типа "сущность-связь" (ERD – *Entity-Relationship Diagram*) в нотации этого стандарта;
- *IDEF2* – *Simulation Model* Design - методология динамического моделирования систем. В настоящее время существуют алгоритмы и их компьютерные реализации, которые позволяют превращать набор статических диаграмм *IDEF0* в динамические модели, построенные на базе "раскрашенных сетей Петри" (*CPN* - Color *Petri Nets*);
- *IDEF3* – *Process Description* Capture – методология документирования процессов, происходящих в системе, которая используется, например, при исследовании технологических процессов на предприятиях. С помощью *IDEF3* описываются сценарий и последовательность операций для каждого процесса. *IDEF3* имеет прямую взаимосвязь с методологией *IDEF0* - каждая функция (функциональный блок) может быть представлена в виде отдельного процесса средствами *IDEF3* ;
- *IDEF4* – *Object-Oriented Design* – методология построения объектно-ориентированных систем. Средства *IDEF4* позволяют наглядно отображать структуру объектов и заложенные принципы их взаимодействия, тем самым позволяя анализировать и оптимизировать сложные объектно-ориентированные системы;
- *IDEF5* – *Ontology Description Capture* – методология онтологического исследования сложных систем. В методологии *IDEF5* онтология системы может быть описана при помощи определенного словаря терминов и правил, на основании которых могут быть сформированы достоверные утверждения о состоянии рассматриваемой системы в некоторый момент времени. На основе этих утверждений формируются выводы о дальнейшем развитии системы, и производится ее оптимизация;
- *IDEF6* – *Design Rationale Capture* – методология использования рационального опыта проектирования, назначение: сохранение рационального опыта проектирования информационных систем для предотвращения структурных ошибок при новом проектировании;
- *IDEF7* – *Information System Auditing* – методология аудита информационной системы;

- *IDEF8 – User Interface Modeling* – методология проектирования интерфейса пользователя;
- *IDEF9 – Scenario-Driven IS Design* – методология анализа имеющихся условий и ограничений, в том числе физических, юридических, политических, и их влияния на принимаемые решения в процессе реинжиниринга;
- *IDEF10 - Implementation Architecture Modeling* – моделирование архитектуры выполнения;
- *IDEF11 – Information Artifact Modeling* – информационное моделирование артефактов;
- *IDEF12 – Organization Modeling* – организационное моделирование;
- *IDEF13 – Three Schema Mapping Design* – трехсхемный дизайн карт;
- *IDEF14 – Network Design* – методология моделирования компьютерных сетей. Позволяет выполнять представление и анализ данных при проектировании вычислительных сетей на графическом языке с описанием конфигураций, очередей, сетевых компонентов, требований к надежности и т.п.

  Подробнее остановимся на *IDEF3*.

IDEF3 – способ описания процессов с использованием структурированного метода, позволяющего эксперту в предметной области представить положение вещей как упорядоченную последовательность событий с одновременным описанием объектов, имеющих непосредственное отношение к процессу.

IDEF3 является технологией, хорошо приспособленной для сбора данных, требующихся для проведения структурного анализа системы.

В отличие от большинства технологий моделирования бизнес-процессов, **IDEF3** не имеет жестких синтаксических или семантических ограничений, делающих неудобным описание неполных или нецелостных систем. Кроме того, автор модели (системный аналитик) избавлен от необходимости смешивать свои собственные предположения о функционировании системы с экспертными утверждениями в целях заполнения пробелов в описании предметной области. На рис. 6.17 изображен пример описания процесса с использованием **методологии IDEF3**.

IDEF3 также может быть использован как метод проектирования бизнес-процессов. **IDEF3-моделирование** органично дополняет традиционное моделирование с использованием стандарта методологии IDEF0. В настоящее время оно получает все большее распространение как вполне жизнеспособный путь построения моделей проектируемых систем для дальнейшего анализа имитационными методами. Имитационное тестирование часто используют для оценки эксплуатационных качеств разрабатываемой системы.

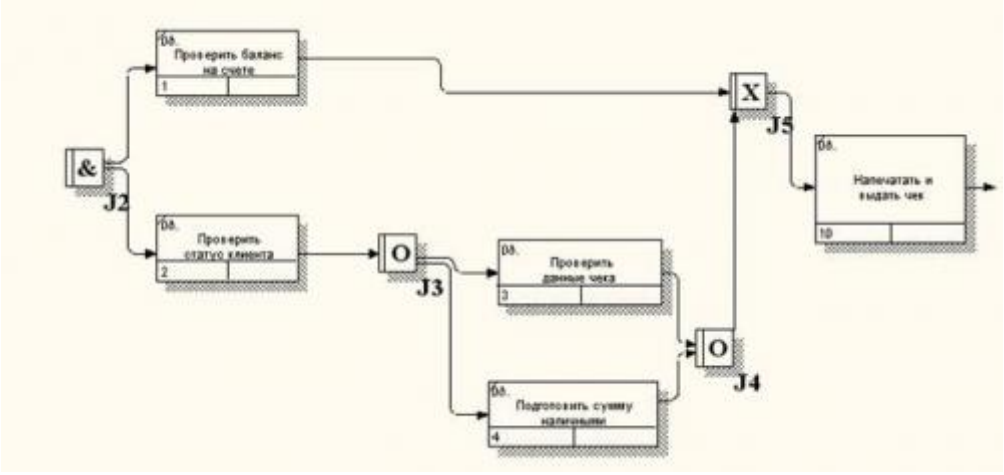


Рисунок 6.17 *Описание процесса в методологии* `IDEF3`

6.4.2 Синтаксис и семантика моделей IDEF3

Основой модели **`IDEF3`** служит так называемый сценарий бизнес-процесса, который выделяет последовательность действий или подпроцессов анализируемой системы. Поскольку сценарий назначение и границы модели, довольно важным является подбор подходящего наименования для обозначения действий. Для подбора необходимого имени применяются стандартные рекомендации по предпочтительному использованию глаголов и отглагольных существительных, например, «обработать заказ клиента» или «применить новый дизайн».

Сценарий для большинства моделей должен быть документирован. Обычно это название набора должностных обязанностей человека, являющегося источником информации о моделируемом процессе.

Также важным для системного аналитика является понимание цели моделирования – набора вопросов, ответами на которые будет служить модель, границ **моделирования** – какие части системы войдут, а какие не будут отображены в модели, и целевой аудитории – для кого разрабатывается модель.

Диаграммы

Главной организационной единицей модели **`IDEF3`** является диаграмма. Взаимная организация диаграмм внутри модели **`IDEF3`** особенно важна в случае, когда модель заведомо создается для последующего опубликования или рецензирования, что является вполне обычной практикой при проектировании новых систем. В этом случае системный аналитик должен позаботиться о таком информационном наполнении диаграмм, чтобы каждая из них была самодостаточной и в то же время понятной пользователю.

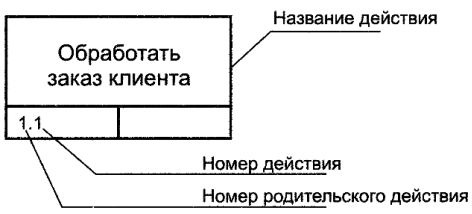


Рисунок 6.18

Единица работы. Действие

Аналогично другим технологиям моделирования действие, или в терминах IDEF3 *«единица работы» (Unit of Work — UOW)*, – другой важный компонент модели. Диаграммы IDEF3 отображают действие в виде прямоугольника. Как уже отмечалось, действия именуются с использованием глаголов или отглагольных существительных, каждому из действий присваивается уникальный идентификационный номер. Этот номер не используется вновь даже в том случае, если в процессе построения модели действие удаляется. В диаграммах IDEF3 номер действия обычно предваряется номером его родителя (рис. 6.18)

Связи

Связи выделяют существенные взаимоотношения между действиями. Все связи в IDEF3 являются однонаправленными, и хотя, стрелка может начинаться или заканчиваться на любой стороне блока, обозначающего действие, диаграммы IDEF3 обычно организуются слева направо таким образом, что стрелки начинаются на правой и заканчиваются на левой стороне блоков. В табл. 6.1 приведены три возможных типа связей.

Связь типа «временное предшествование». Как видно из названия, связи этого типа показывают, что исходное действие должно полностью завершиться, прежде чем начнется выполнение конечного действия. Связь должна быть поименована таким образом, чтобы человеку, просматривающему модель, была понятна причина ее появления.

Изображение	Название	Назначение
→	Временно́е предшествование (Temporal precedence)	Исходное действие должно завершиться, прежде чем конечное действие сможет начаться
→►	Объектный поток (Object flow)	Выход исходного действия является входом конечного действия. Из этого, в частности, следует, что исходное действие должно завершиться, прежде чем конечное действие сможет начаться
- - →	Нечеткое отношение (Relationship)	Вид взаимодействия между исходным и конечным действиями задается аналитиком отдельно для каждого случая использования такого отношения

Таблица 6.1

Во многих случаях завершение одного действия инициирует начало выполнения другого, как показано на рис. 6.19. В этом примере автор должен принять рекомендации рецензентов, прежде чем начать вносить соответствующие изменения в работу.

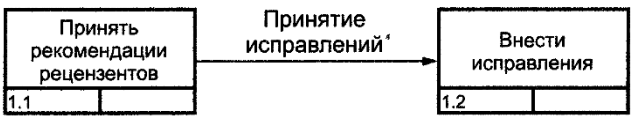


Рисунок 6.19 *Связь типа «временное предшествование» между действиями 1.1 и 1.1*

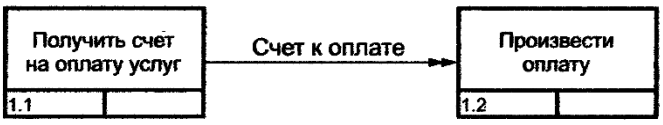


Рисунок 6.20 *Объектная связь между действиями 1.1 и 1.2*

Временная семантика объектных связей аналогичная связям предшествования, это означает, что порождающее объектную связь исходное действие должно завершиться, прежде чем конечное действие может начать выполняться, как показано на рис. 6.20. В приведенном примере счет на оплату услуг является результатом выполнения действия 1.1

Рисунок 6.21 *Связь типа «нечеткое отношение»*

Связь типа «нечеткое отношение». Связи этого типа используются для выделения отношений между действиями, которые невозможно описать с использованием предшественных или объектных связей. Значение каждой такой связи должно быть определено, поскольку связи типа «нечеткое отношение» сами по себе не предполагают никаких ограничений. Одно из применений нечетких отношений – отображение взаимоотношений между параллельно выполняющимися действиями. На рис. 6.21 приводится фрагмент процесса запуска бензопилы с водяным охлаждением и нечеткое отношение между действиями «запустить двигатель» и «запустить водяной насос». Название стрелки может быть использовано для описания типа отношения, более подробное объяснение может быть приведено в виде отдельной ссылки.

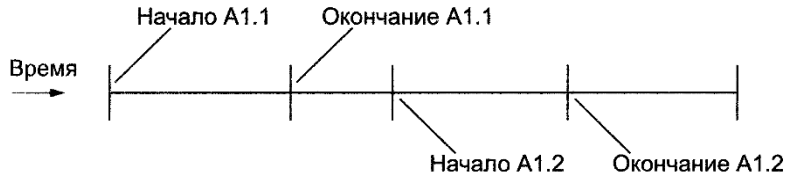


Рисунок 6.22 *Временная шкала выполнения действия для рис. 6.19*

Графическое обозначение	Название	Вид	Правила инициации
&	Соединение «и»	Разворачивающее	Каждое конечное действие обязательно инициируется
		Сворачивающее	Каждое исходное действие обязательно должно завершиться
X	Соединение «эксклюзивное “или”»	Разворачивающее	Одно и только одно конечное действие инициируется
		Сворачивающее	Одно и только одно исходное действие должно завершиться

Таблица 6.2

Продолжение

Графическое обозначение	Название	Вид	Правила инициации
O	Соединение «или»	Разворачивающее	Одно или несколько конечных действий инициируются
		Сворачивающее	Одно или несколько исходных действий должны завершиться

Продолжение таблицы 6.2

Примеры разворачивающих и сворачивающих соединений приведены на рис. 6.23.

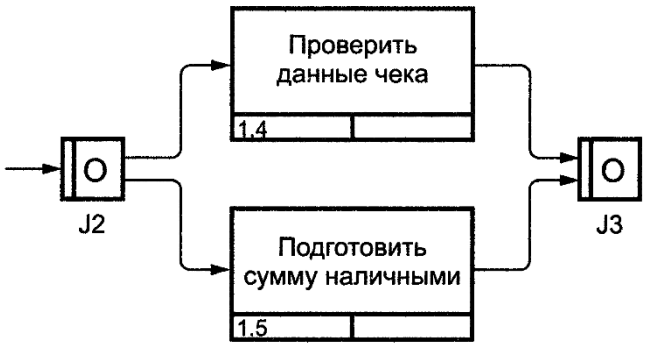


Рисунок 6.23 *Два вида соединений*

«И» соединения. Соединения этого типа инициируют выполнение конечных действий. Все действия, присоединенные к сворачивающему «и» соединению, должны завершиться, прежде чем начнется выполнение следующего действия. На рис. 6.24 после обнаружения

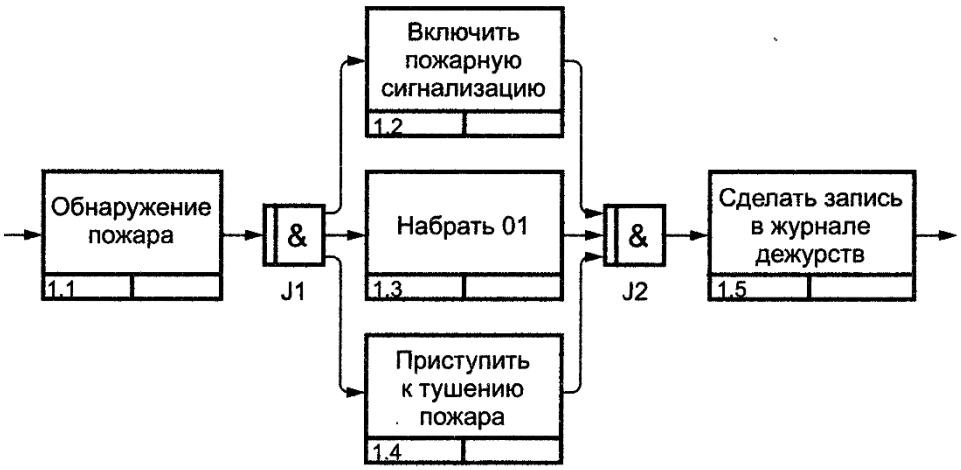


Рисунок 6.24 *«И»-соединения*

Соединение «эксклюзивное «или». Вне зависимости от количества действий, связанных со сворачивающим или разворачивающим соединением «эксклюзивное «или», инициировано будет только одно из них, и поэтому только оно будет завершено перед тем, как любое действие, следующее за сворачивающим соединением «эксклюзивное «или», сможет начаться. Если правила активации соединения известны, они обязательно должны быть документированы либо в его описании, либо пометкой стрелок, исходящих из разворачивающего соединения, как показано на рис. 6.25.

На рис. 6.25 соединение «эксклюзивное «или» используется для отображения того факта, что студент не может одновременно быть направлен на лекции по двум разным курсам.

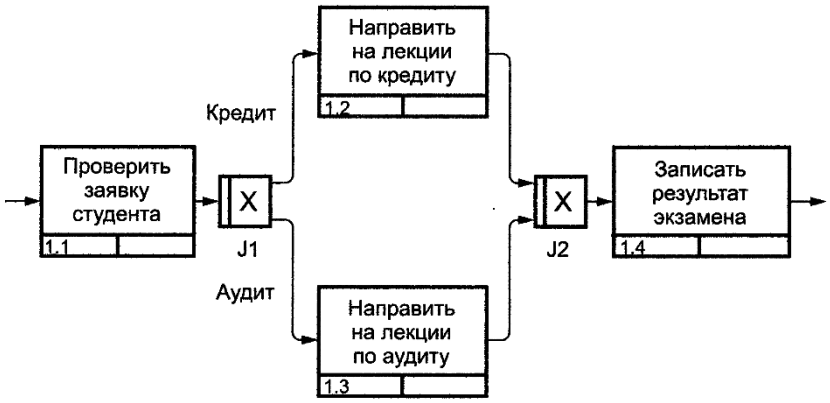


Рисунок 6.25 *Соединение «эксклюзивное «или»»*

Графическое обозначение	Тип	Вид	Правила инициации
	Соединение «и»	Разворачивающее	Все действия начнутся одновременно
		Сворачивающее	Все действия закончатся одновременно
	Соединение «или»	Разворачивающее	Может быть, несколько действий начнутся одновременно
		Сворачивающее	Может быть, несколько действий закончатся одновременно
	Соединение «эксклюзивное «или»	Разворачивающее	Одновременное начало действий невозможно
		Сворачивающее	Одновременное окончание действий невозможно

Таблица 6.3

Указатели

Указатели – это специальные символы, которые ссылаются на другие разделы описания процесса. Они используются при построении диаграммы для привлечения внимания пользователя к каким-либо важным аспектам модели.

Указатель изображается на диаграмме в виде прямоугольника, похожего на изображение действия. Имя указателя обычно включает его гип (например, ОБЪЕКТ, UOB и т.п.) и идентификатор (табл. 6.4).

Тип указателя	Назначение
ОБЪЕКТ (OBJECT)	Для описания того, что в действии принимает участие какой-либо заслуживающий отдельного внимания объект
ССЫЛКА (GOTO)	Для реализации цикличности выполнения действий. Указатель ССЫЛКА может относиться и к соединению
ЕДИНИЦА ДЕЙСТВИЯ (Unit of Behavior — UOB)	Для многократного отображения на диаграмме одного и того же действия. Например, если действие «Подсчет наличных» выполняется несколько раз, в первый раз оно создается как действие, а последующие его появления на диаграмме оформляются указателями UOB
ЗАМЕТКА (NOTE)	Для документирования любой важной информации общего характера, относящейся к изображенному на диаграммах. В этом смысле ССЫЛКА служит альтернативой методу помещения текстовых заметок непосредственно на диаграммах
УТОЧНЕНИЕ (Elaboration — ELAB)	Для уточнения или более подробного описания изображенного на диаграмме. Указатель УТОЧНЕНИЕ обычно используется для описания логики ветвления у соединений

Таблица 6.4

Декомпозиция действий

Действия в **IDEF3** могут быть декомпозированы или разложены на составляющие для более детального анализа. Метод **IDEF3** позволяет декомпозировать действие несколько раз, что обеспечивает документирование альтернативных потоков процесса в одной модели.

Для корректной идентификации действий в модели с множественными декомпозициями схема нумерации действий расширяется и наряду с номерами действия и его родителя включает в себя порядковый номер декомпозиции. Например, в номере действия 1.2.5: 1 — номер родительского действия, 2 — номер декомпозиции, 5 — номер действия.

7 UML ДИАГРАММЫ

7.1 Основы UML

Унифицированный язык моделирования (Unified Modeling Language, UML) – это универсальный язык визуального моделирования систем он предназначен для описания, визуализации и документирования объектно-ориентированных систем в процессе их анализа и проектирования.

Важно понимать, что UML не предлагает какой либо методологии моделирования.

Язык UML предоставляет стандартный способ написания проектной документации на системы, включая концептуальные аспекты, такие как бизнес процессы и функции системы, а также конкретные аспекты, такие как выражения языков программирования, схемы баз данных и повторно используемые компоненты ПО

До 1994 года в мире Обьектно-ориентированных методов царил хаос. Существовало несколько конкурирующих языков и методологий визуального моделирования, каждая с собственными преимуществами и недостатками, сторонниками и противниками. Среди языков визуального моделирования очевидными лидерами были *метод Буча (Booch Method), Гради Бучем (Grady Booch), и техника объектного моделирования (Object Modeling Technique, OMT) Джеймса Рамбо (James Rumbaugh)*, которые занимали более половины рынка. Что касается методологий, самую строгую систему создал *Айвар Джекобсон (Ivar Jacobson)*. В 2005 году была завершена спецификация UML 2.0. Теперь UML – вполне сформировавшийся язык моделирования.

- Язык UML не является методологией
- Язык UML не является процессом
- Язык UML не является языком программирования
- Язык UML не является формальным языком
- UML = нотация + семантика !

Основные понятия визуального моделирования

- *Нотация* – система условных обозначений для графического представления визуальных моделей

- *Семантика* – система правил и соглашений, определяющая смысл и интерпретацию конструкций некоторого языка
- *Методология* – совокупность принципов моделирования и подходов к логической организации методов и средств разработки моделей

В UML 2 появилось много новых визуальных синтаксических структур. Некоторые из них замещают (и уточняют) существующий синтаксис версии 1.x, другие – абсолютно новые и представляют вновь введённую в язык семантику. UML как всегда предлагает множество вариантов представления конкретного элемента модели, но не все они будут поддерживаться каждым из инструментов моделирования.

Основная идея UML – возможность моделировать программное обеспечение и другие системы как наборы взаимодействующих объектов.

В UML есть два типа моделей:

- *Структурные модели (structured models)* – модели, предназначенные для описания статической структуры сущностей или элементов некоторой системы, включая их классы, интерфейсы, атрибуты и отношения.
- *Модели поведения (behavioral models)* – модели, предназначенные для описания процесса функционирования элементов системы, включая их методы и взаимодействие между ними, а также процесс изменения состояний отдельных элементов и системы в целом.
- Особенности изображения диаграмм в нотации UML
- Графические узлы на плоскости, которые изображаются с помощью геометрических фигур и могут иметь различную высоту и ширину с целью размещения внутри этих фигур других конструкций языка UML
- Пути, которые представляют собой последовательности из отрезков линий, соединяющих отдельные графические узлы
- Значки или пиктограммы. Значок представляет собой графическую фигуру фиксированного размера и формы, которая не может увеличивать свои размеры, чтобы разместить внутри себя дополнительные символы.
- Строки текста. Служат для представления различных видов информации в некоторой грамматической форме.
- Общие рекомендации по изображению диаграмм в нотации языка UML
- Каждая диаграмма должна служить законченным представлением соответствующего фрагмента моделируемой предметной области
- Все сущности на диаграмме модели должны быть одного концептуального уровня
- Вся информация о сущностях должна быть явно представлена на диаграммах
- Диаграммы не должны содержать противоречивой информации
- Диаграммы не следует перегружать текстовой информацией
- Каждая диаграмма должна быть само достаточной для правильной интерпретации всех ее элементов и понимания семантики всех используемых графических символов

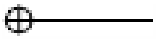
Тип отношения	UML-синтаксис		Краткая семантика
	источник	цель	
Зависимость			Исходный элемент зависит от целевого элемента и изменение последнего может повлиять на первый.
Ассоциация			Описание набора связей между объектами.
Агрегация			Целевой элемент является частью исходного элемента.
Композиция			Строгая (более ограниченная) форма агрегирования.
Включение			Исходный элемент содержит целевой элемент.
Обобщение			Исходный элемент является специализацией более обобщенного целевого элемента и может замещать его.
Реализация			Исходный элемент гарантированно выполняет контракт, определенный целевым элементом.

Таблица 7.1

7.2 Диаграммы UML.

Во всех инструментальных средствах UML моделирования новые сущности или новые отношения при создании добавляются в модель. *Модель* – это хранилище всех сущностей и отношений, созданных для описания требуемого поведения проектируемой программной системы.

Диаграммы – это своего рода картины, или представления модели. Диаграмма – это не модель! На самом деле, различие между диаграммой и моделью является очень важным для понимания, поскольку сущность или отношение могут быть удалены с диаграммы, или даже со всех диаграмм, но по-прежнему они продолжают существовать в модели. Они будут оставаться в модели до тех пор, пока не будут явно удалены из нее. Общая ошибка новичков в UML моделировании состоит в том, что они удаляют сущности с диаграмм, не удаляя их из модели.

Существует тринадцать различных типов UML диаграмм. Каждый прямоугольник представляет определённый тип диаграммы, при этом курсивом выделяются абстрактные категории типов диаграмм. Например, существует шесть разных типов Диаграмм структуры. Обычный текст указывает на конкретную диаграмму, которую можно реально создать. Заштрихованные блоки обозначают типы диаграмм, впервые появившиеся в UML 2.

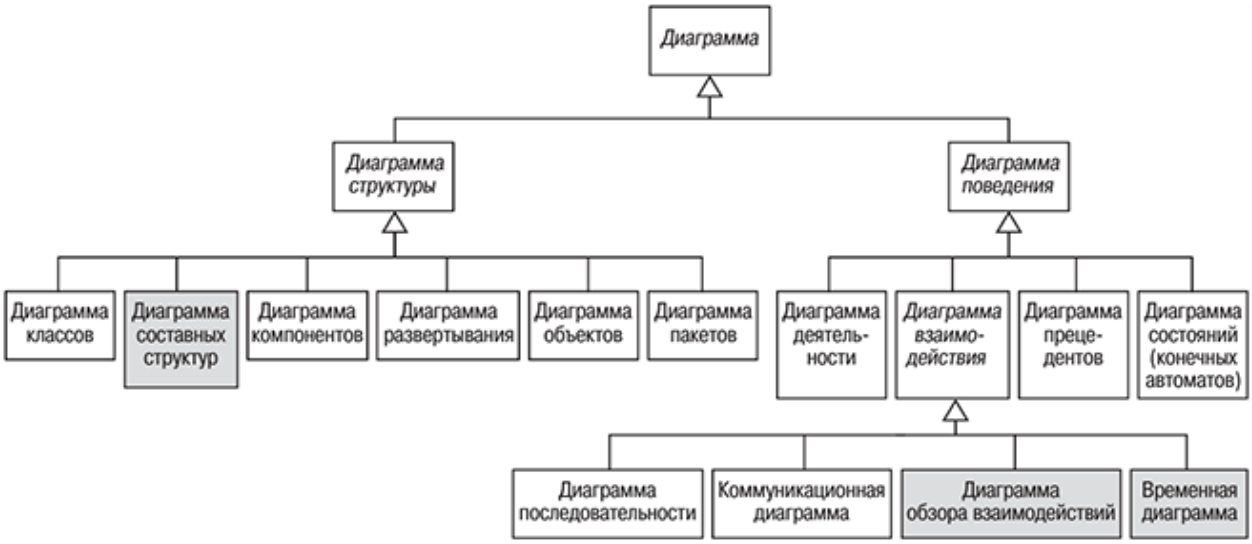


Рисунок 7.1

В UML 2 представлен новый синтаксис диаграмм, изображенный на рисунке. У каждой диаграммы может быть рамка, область заголовка и область содержимого. Область заголовка – это неправильный пятиугольник, содержащий тип (не обязательно), имя и параметры (не обязательно) диаграммы.

Рисунок. 7.2

<Тип> указывает тип данной диаграммы и должен быть одним из типов, перечисленных на рис. 7.1. Спецификация UML определяет, что <тип> может быть сокращен, но не предоставляет списка стандартных сокращений. Явное задание <типа> требуется редко, потому что его обычно легко определить из визуального синтаксиса.

<Имя> должно описывать семантику диаграммы (например, CourseRegistration (регистрация курса)). <Параметры> предоставляют информацию, необходимую элементам модели, представленным на диаграмме. Примеры использования <параметров> будут приведены позже. У диаграммы может быть (не обязательно) условная рамка, ограничивающая область в инструменте моделирования, внутри которой находится диаграмма.

Рекомендации по изображению диаграмм в нотации языка UML

- Количество диаграмм различных типов для модели конкретного приложения не является строго фиксированным
- Любая из моделей системы должна содержать только те элементы, которые определены в соответствующей версии языка UML
- Каждая диаграмма в нотации языка UML 2.x имеет область содержания для изображения графических узлов и путей между ними, которые представляют собой собственно элементы модели в нотации UML 2.x
- Фрейм в нотации UML 2.x используется в тех случаях, когда отдельные элементы диаграммы имеют графическую границу с другими элементами диаграммы
- Общие механизмы UML

В UML существует четыре общих механизма, последовательно применяемых ко всему языку моделирования. Они описывают четыре стратегии подхода к моделированию объектов, которые в разных контекстах многократно применяются в UML.

Спецификации

Модели UML имеют, по крайней мере, два измерения: графическое, позволяющее визуализировать модель с помощью диаграмм и пиктограмм, и текстовое, состоящее из спецификаций различных элементов модели. Спецификации – это текстовые описания семантики элемента.

UML обеспечивает большую гибкость при создании моделей. В частности, модели могут быть:

•*сокращенными* – некоторые элементы присутствуют в заднем плане, но скрыты в той или иной диаграмме для упрощения представления;

•*неполными* – некоторые элементы модели могут быть полностью пропущены;

•*несогласованными* – модель может содержать противоречия.

Дополнения

В UML каждый элемент модели обозначается простым символом, к которому можно добавлять ряд дополнений, визуализирующих аспекты спецификации элемента. С помощью этого механизма видимая на диаграмме информация может быть представлена в соответствии с конкретными требованиями. Важно помнить, что любая UML диаграмма – это только представление модели, и поэтому необходимо показывать лишь те дополнения, которые подчеркивают важные характеристики модели и делают диаграмму более понятной и удобочитаемой.

Принятые деления

Принятые деления описывают конкретные способы представления мира. В UML существует два принятых деления: классификатор/экземпляр и интерфейс/реализация.

Механизмы расширения:

	Механизмы расширения UML
Ограничения	Расширяют семантику элемента, обеспечивая возможность добавлять новые правила.
Стереотипы	Обеспечивают возможность определять новые элементы модели UML на основании существующих: мы определяем семантику стереотипа самостоятельно. Стереотипы добавляют новый элемент в метамодель UML.
Помеченные значения	Предоставляют способ расширения спецификации элемента, обеспечивая возможность добавлять в него новую специальную информацию.

Таблица 7.2

- *Стереотип (stereotype)* — новый тип элемента модели, который расширяет семантику базового типа метамодели языка UML
- *Ограничение (constraint)* — некоторое логическое условие, ограничивающее семантику выбранного элемента модели
- *Помеченное значение (tagged value)* — явное определение некоторого свойства объекта как пары «имя – значение»

Use case диаграмма

Диаграмма, на которой изображаются варианты использования проектируемой системы, заключенные в границу системы, и внешние актеры, а также определенные отношения между актерами и вариантами использования

Актер – это любая внешняя по отношению к проектируемой системе сущность, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей или решения частных задач

Примеры актеров: кассир, клиент банка, банковский служащий, президент, продавец магазина, менеджер отдела продаж, пассажир авиарейса, водитель автомобиля, администратор гостиницы, сотовый телефон. Важно осознавать, что актеры всегда являются внешними по отношению к системе.

Назначение диаграммы use case

- Определить общие границы функциональности проектируемой системы в контексте моделируемой предметной области.
- Специфицировать требования к функциональному поведению проектируемой системы в форме вариантов использования.
- Разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей.
- Подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями

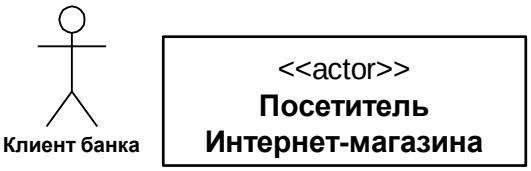


Рисунок 7.3

Отношения в диаграмме

Отношение ассоциации

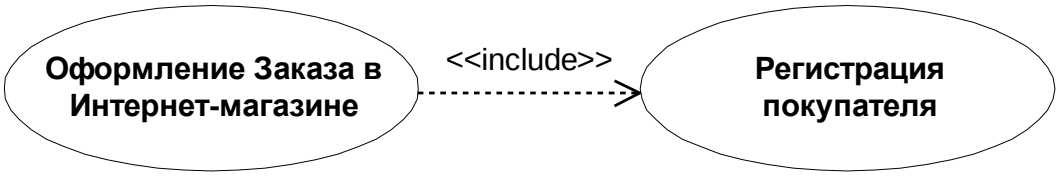
Ассоциация (*association*) является одним из фундаментальных понятий в языке UML 2.x и может использоваться на различных канонических диаграммах при построении визуальных моделей

Применительно к диаграммам вариантов использования отношение ассоциации может служить только для обозначения взаимодействия актера с вариантом использования.

Рисунок 7.3

Отношение включения

- Отношение *зависимости (dependency)* определяется как форма взаимосвязи между двумя элементами модели, предназначенная для спецификации того обстоятельства, что изменение одного элемента модели приводит к изменению некоторого другого элемента
- Отношение *включения (include)* специфицирует тот факт, что некоторый вариант использования содержит поведение, определенное в другом варианте использования



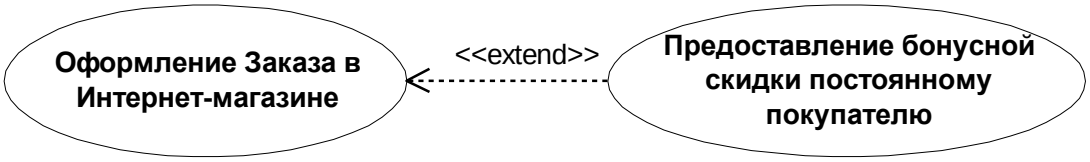
вариант использования *А* вариант использования *Б*

Клиент зависимости источник зависимости

Рисунок 7.4

Отношение расширения

Отношение *расширения (extend)* определяет взаимосвязь одного варианта использования с некоторым другим вариантом использования, функциональность или поведение которого задействуется первым не всегда, а только при выполнении некоторых дополнительных условий.



вариант использования *А* вариант использования *Б*

Рисунок 7.5

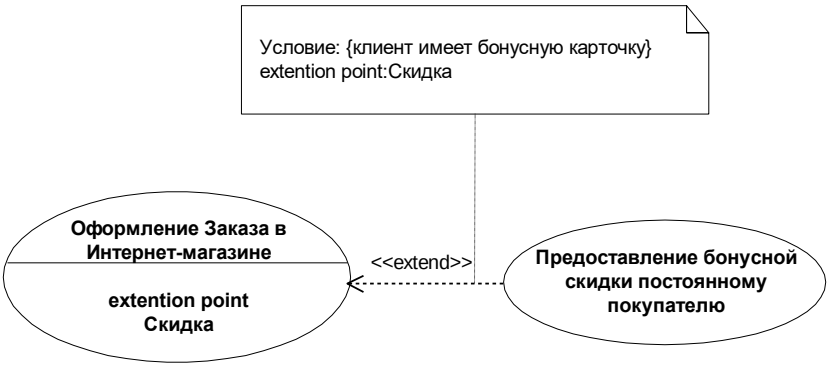


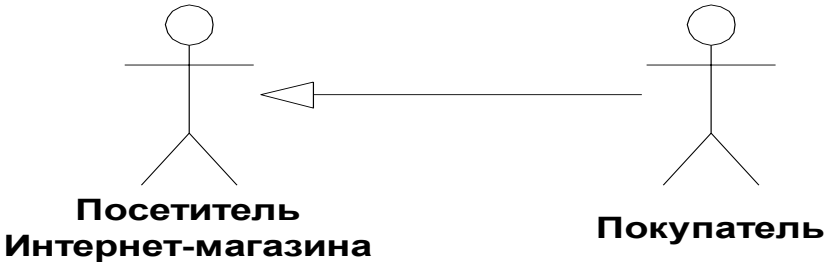
Рисунок 7.6 *Изображение отношения расширения с условием выполнения*

Отношение обобщения

Отношение обобщения (generalization relationship) предназначено для спецификации того факта, что один элемент модели является специальным или частным случаем другого элемента модели

вариант использования *А* вариант использования *Б*

Рисунок 7.7



(актер *А*) (актер *Б*)

Рисунок 7.8

Типичные ошибки при разработке диаграмм вариантов использования

- Превращение диаграммы вариантов использования в диаграмму деятельности за счет желания отразить все функциональные действия
- Инициатором действий является разрабатываемая система
- Спецификация атрибутов и операций классов до того, как сформулированы все варианты использования
- Задание слишком кратких имен вариантам использования
- Описание вариантов использования в терминологии, непонятной пользователям системы или заказчику
- Отсутствие описаний альтернативных последовательностей действий
- Тратится слишком много времени на решение вопросов о том, какие стереотипы и ассоциации использовать на диаграмме

7.3 Диаграммы классов

Диаграммы классов – один из основных типов диаграмм в UML. Они предназначены для отражения статических связей в программах – иначе говоря, они показывают, как сущности работают в сочетании друг с другом.

В ходе написания программы разработчику приходится постоянно принимать архитектурные решения: какие классы содержат ссылки на другие классы, какой класс является «владельцем» другого класса и т. д. Диаграммы классов дают возможность отразить эту «физическую» структуру системы.

7.3.1 Классы

Класс представляет группу сущностей, обладающих общим состоянием и поведением. Это своего рода «пресс-форма» для изготовления объектов в объектно-ориентированной системе. В терминологии UML класс является разновидностью *классификатора*. Например, Volkswagen, Toyota и Ford – марки автомашин, и для их представления можно использовать класс с именем Car. Каждый конкретный тип машин является *экземпляром* класса, или *объектом*. Класс может представлять некоторую материальную, конкретную концепцию (скажем, счет); обобщенное понятие (например, документ или транспортное средство – в отличие от счета или мотоцикла с объемом свыше 1000 см^3) или же абстрактную концепцию вроде стратегии рискованных вложений в ценные бумаги.

Класс представляется прямоугольным блоком, состоящим из *разделов*. Раздел – часть прямоугольника для записи информации. В первом разделе содержится имя класса, во втором – атрибуты, а третий используется для операций. Любой раздел класса можно скрыть, если это сделает программу более удобочитаемой. При чтении диаграммы не следует делать допущений относительно отсутствующих разделов; даже если они не показаны на диаграмме, это не означает, что они пусты. К классу можно добавить новые разделы для дополнительной информации (скажем, исключений или событий), хотя это и выходит за границы типичной записи.

UML рекомендует, чтобы имя класса:

- Начиналось с прописной буквы;

- Было выровнено по центру верхнего раздела;
- Записывалось жирным шрифтом;
- Записывалось курсивом, если класс является *абстрактным*.

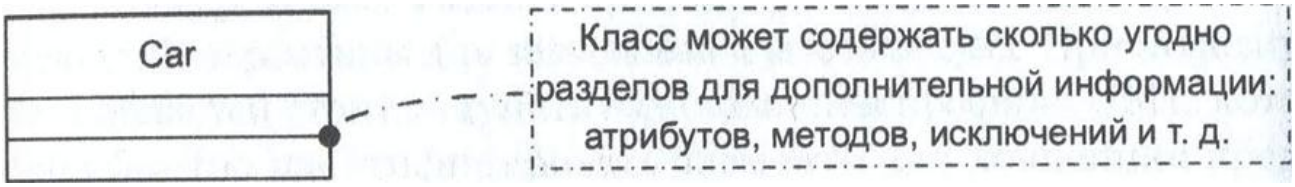


Рисунок 7.9 *Простой класс*

Объекты

Объект является экземпляром класса. Например, класс `Car` может быть воплощен в нескольких разновидностях-экземплярах: 2-дверная машина с двумя дверями, 4-дверная синяя машина, зеленый «хэтчбэк». Каждый экземпляр `Car` является объектом и может обладать собственным именем, хотя на диаграммах объектов нередко встречаются неименованные (анонимные) объекты. Обычно после имени объекта идет двоеточие, за которым следует тип (то есть класс). Имя и тип экземпляра класса подчеркивается – например, на рис 7.10 показан экземпляр класса `Car` с именем `Toyota`. Обратите внимание: на этом рисунке пустые разделы скрыты.

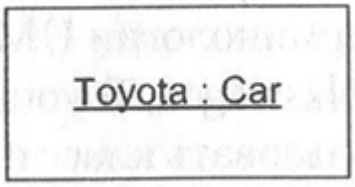


Рисунок 7.10 *Экземпляр Car*

Это как снимок части исполняющейся системы в определенный момент, показывающий объекты и связи между ними.

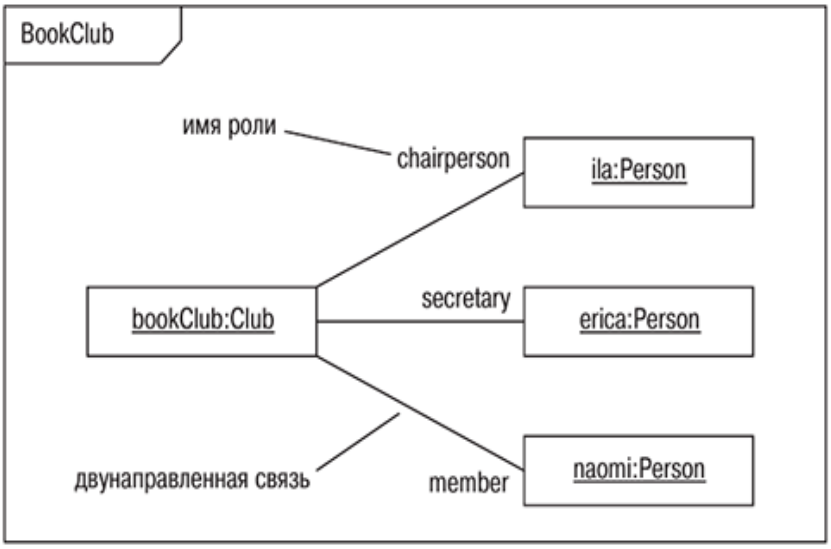


Рисунок 7.11

Обычно одна связь соединяет только два объекта, как показано на рис. 7.11. Однако UML допускает соединение нескольких объектов одной связью. Такую связь называют n-арной. Она изображается в виде ромба, от которого отходят линии к каждому из объектов участников.

Иерархии и сети

В процессе моделирования вы обнаружите, что часто объекты организуются в иерархии или сети. Иерархия имеет один корневой объект.

У каждого последующего узла иерархии только один прямой предшественник. Деревья каталогов обычно формируют иерархии

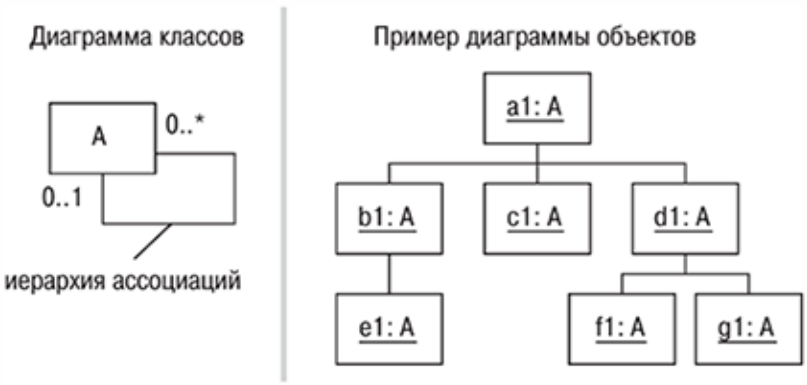


Рисунок 7.12

В сети нет строгого представления «над» или «под». Это намного более гибкая структура, в которой возможно равенство между узлами.

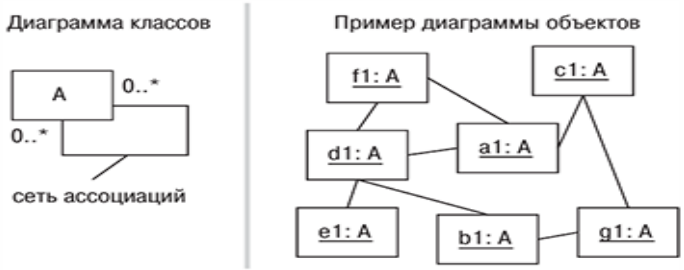


Рисунок 7.13

Возможность навигации (`navigability`) указывает на возможность прохода от объекта исходного класса к одному или более объектам в зависимости от кратности целевого класса. Смысл навигации в том, что «сообщения могут посылаться только в направлении, в котором указывает стрелка».

Одна из целей хорошего ОО анализа и проектирования – минимизировать количество взаимосвязей между классами. И применение возможности навигации – верное средство достижения этой цели.

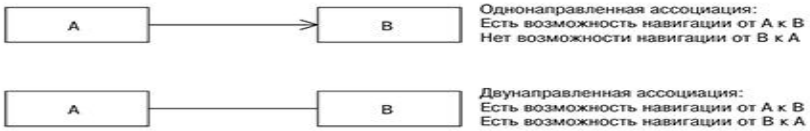


Рисунок 7.14

Разновидности классов

- *Абстрактный* (`abstract`) класс не имеет экземпляров или объектов, для обозначения его имени используется наклонный шрифт (*курсив*)
- *Активный класс (`active class`)* – класс, каждый экземпляр которого имеет свою собственную нить управления
- *Пассивный класс (`passive class`)* – класс, каждый экземпляр которого выполняется в контексте некоторого другого объекта
- *Квалифицированное имя* (`qualified name`) используется для того, чтобы явно указать, к какому пакету относится тот или иной класс. Для этого применяется специальный символ в качестве разделителя имени – двойное двоеточие «`::`» Имя класса без символа разделителя называется *простым именем* класса.

Атрибуты

Уточняющие характеристики класса (цвет машины, количество сторон в геометрической фигуре и т. д.) представляются *атрибутами*. Атрибуты могут относиться к простым примитивным типам (целые числа, вещественные числа и т. д.) или быть другими, сложными объектами.

Существуют два разных варианта записи атрибутов: встроенные и отношения между классами. Кроме того, существует запись для отображения таких показателей, как множественность, уникальность и упорядоченность.

Встроенные атрибуты

Атрибуты класса могут перечисляться прямо на прямоугольном блоке; обычно такие атрибуты называются *встроенными* (*inline*). Между ними и атрибутами, оформленными посредством отношений, не существует семантических различий; обычно дело лишь в количестве подробностей, которые вы хотите представить на диаграмме (или в случае с примитивными типами вроде целочисленного – которые вы можете представить).

Чтобы представить атрибут в теле класса, поместите его во второй раздел блока. При записи встроенных атрибутов применяется следующая схема:

```
видимость / имя : тип множественность = по_умолчанию
  { строки свойств и ограничений }

видимость  ::= { + | - | # | ~}

множественность ::= [ нижняя_граница .. верхняя_граница ]
```

На рис 7.15 приведены примеры атрибутов, демонстрирующие различные варианты их записи.

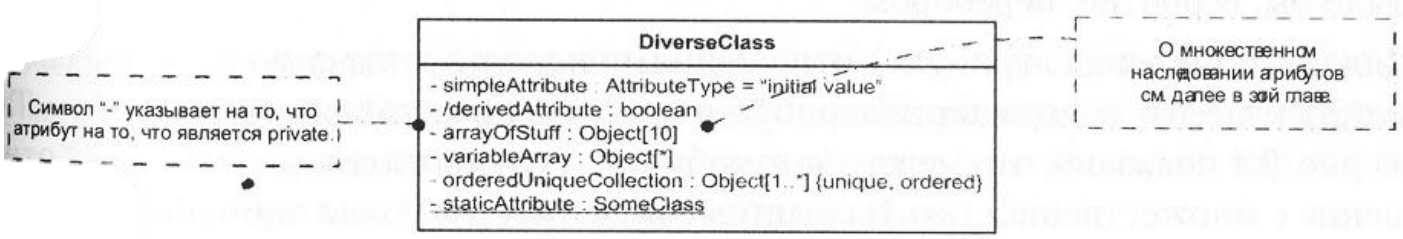


Рисунок 7.15 *Примеры атрибутов*

Основные элементы синтаксиса атрибутов:

- *видимость* – знаки `+`, `-`, `#` или `~` соответствуют уровням `public`, `private`, `protected` и `package` соответственно;
- `/` - признак *производного* атрибута. Производным является атрибут, вычисляемый на основании других атрибутов класса.
- *имя* – существительное или именное словосочетание с описанием атрибута. Обычно имя начинается со строчной буквы, но первая буква каждого последующего слова является прописной;
- *тип* – тип атрибута как другой классификатор (обычно класс, интерфейс или встроенный тип вроде `int`);
- *множественность* – количество экземпляров типа атрибута, соответствующих данному атрибуту. Может не указываться (подразумевается множественность 1), задаваться одним целым числом или в виде диапазона значений в квадратных скобках, с разделением границ «`..`». Символ «`*`» в качестве верхней границы диапазона соответствует максимальному значению, а сам по себе обозначает ноль и более.
- *по_умолчанию* – значение атрибута по умолчанию;
- *свойства* – набор свойств (меток), присоединяемых к атрибутам. Обычно свойства зависят от контекста и обозначают такие понятия, как упорядоченность или уникальность. Они заключаются в фигурные скобки `{}` и разделяются запятыми;
- *ограничения* – одно или несколько ограничений, установленных для атрибута. Ограничения записываются на естественном языке или в формальной грамматике типа `OCL`.

Запись атрибутов с использованием отношений

Для предоставления атрибутов также используется запись с отношениями. В этом случае диаграмма классов занимает больше места, но содержит больше информации для сложных типов атрибутов. Кроме того, запись с отношениями точнее передает место атрибута в классе. Например, при моделировании типа `Car` отношения позволяют более четко указать, что машина *содержит* двигатель `(Engine)`, нежели при простом включении атрибута в прямоугольник `Car`. С другой стороны, использовать запись с отношениями для простой строки с именем `Car` было бы, вероятно, перебором.

Для представления атрибута с отношениями используется один из видов связей между классом, содержащим атрибут, и классом, предоставляющим атрибут. Так на

рис. 7.16 показано, что между автомобилем и его двигателем существует отношение с множественностью 1; на автомобиле имеется ровно один двигатель.

Рисунок 7.16 *Атрибуты с записью отношений*

Запись с отношениями поддерживает тот же синтаксис, что встроенная запись, хотя и выглядит несколько иначе. Видимость и имя атрибута размещаются рядом с линией отношения. Не используйте квадратные скобки для обозначения множественности – последняя указывается рядом с классификатором атрибута.

Для атрибутов можно указывать ограничения. В записи с отношениями ограничения ставятся рядом с классификатором атрибута у линии отношения. UML также поддерживает возможность выражения в записи с отношениями ограничения между атрибутами (рис 7.17).

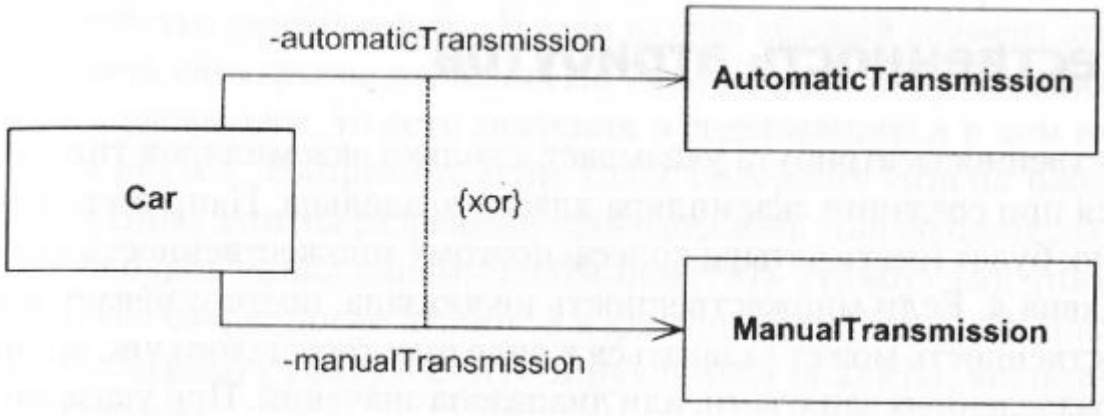


Рисунок 7.17 *Ограничения в записи атрибутов с отношениями*

На рис 7.17 стандартное ограничение UML `xor` («исключающее ИЛИ») показывает, что в любой момент может быть задано либо значение `automaticTransmission` либо `manualTransmission`. При использовании встроенной записи атрибута это ограничение должно быть выражено в примечании.

Производные атрибуты

Производные атрибуты, обозначаемые префиксом / (косая черта), сообщает реализующей стороне, что атрибут не является строго необходимым. Допустим, простой класс с именем `Account` моделирует банковский счет. Класс хранит текущий баланс `balance` в вещественном формате. Для отслеживания превышения кредита в класс добавляется логическая (`boolean`) переменная `overdraw`. Превышения кредита в действительности определяется положительностью или отрицательностью баланса, а не добавленной логической переменной. Чтобы указать на это обстоятельство разработчику, обозначьте overdraw как произвольный атрибут, состояние которого определяется атрибутом balance. На рис 7.18 показано, как это отношение между balance и overdraw передается на диаграмме.

Спецификация `UML` указывает, что произвольный атрибут обычно доступен только для чтения (`readOnly`), то есть его сообщение не может изменяться пользователем. Но если пользователю все же разрешено его изменять, предполагается, что класс соответствующим образом обновит источник производной информации.

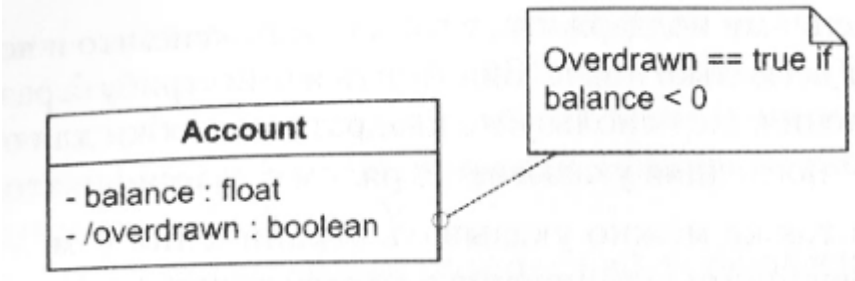


Рисунок 7.18 *Производный атрибут*

Множественность атрибутов

Множественность атрибута указывает, сколько экземпляров типа атрибута появляется при создании экземпляра класса-владельца. Например, наш класс `Car`, вероятно, будет иметь четыре колеса, поэтому множественность атрибута `Wheel` будет равна 4. Если множественность не указана, подразумевается значение 1. Множественность может задаваться в виде простого целого числа, списка целых чисел, разделенных запятыми, или диапазона значений. При указании диапазона бесконечная верхняя граница обозначается `*`; для нижней границы `*` означает ноль и более. Множественность указывается в квадратных скобках – либо как одно целое число, либо как два числа, разделенных двумя точками `(..)`. На рис. 7.19 показаны различные способы представления множественности атрибута.

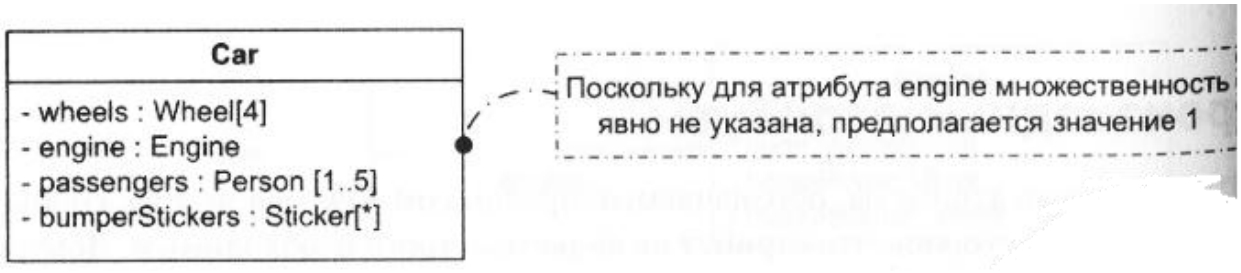


Рисунок 7.19 *Примеры множественности*

Упорядоченность

Для атрибута с множественность больше 1 может быть *упорядоченность*. Если атрибут упорядочен, его элементы должны храниться последовательно. Например, чтобы список имен хранился в алфавитном порядке, пометьте его как упорядоченный. Что именно подразумевается под последовательным хранением атрибутов, зависит от типа атрибута. По умолчанию атрибуты не упорядочиваются. Чтобы пометить атрибут как упорядоченный, укажите свойство `ordered` после атрибута в фигурных скобках, как показано на рис. 7.20.

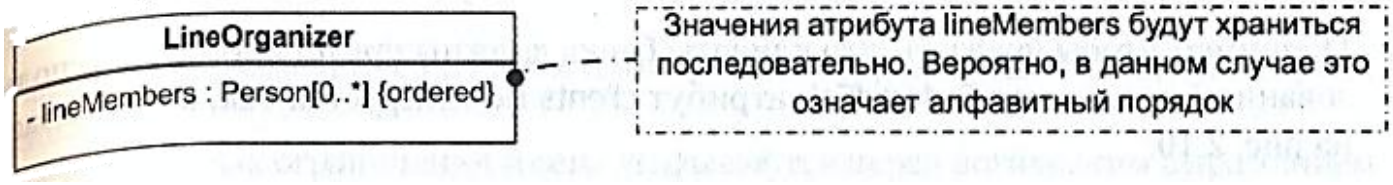


Рисунок 7.20 *Упорядоченный множественный атрибут*

Уникальность

Помимо упорядоченности, для атрибута с множественностью больше 1 может быть задано свойство *уникальности*. В этом случае каждый элемент этого атрибута должен иметь уникальное значение. По умолчанию атрибуты с множественностью более 1 *уникальны*, то есть значения содержащихся в нем элементах не могут повторяться. Например, если класс содержит список избирателей и каждому участнику списка разрешено проголосовать только один раз, каждый элемент списка будет уникальным. Чтобы пометить атрибут как уникальный, укажите свойство `unique` после атрибута в фигурных скобках, как показано на рис. Чтобы атрибут содержал дублирующиеся объекты, воспользуйтесь свойством `not unique`.

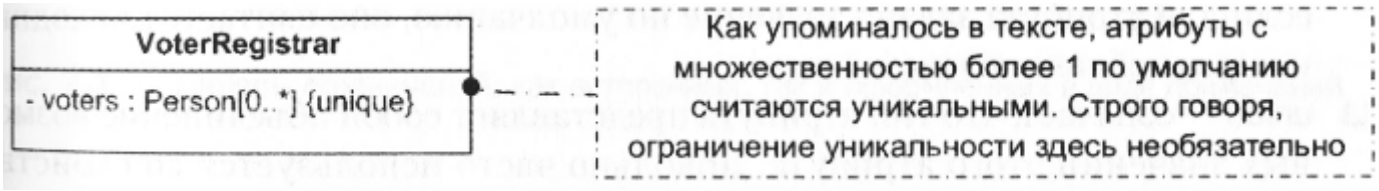


Рисунок 7.21 *Уникальный множественный атрибут*

Коллекции

В спецификации UML определены отображения различных сочетаний свойств упорядоченности и уникальности на типы-коллекции UML; они показаны в таблице 7.3. Обратите внимание: коллекции из табл. 7.3 являются отображениями UML и могут не иметь прямого соответствия среди классов используемого языка.

Таблица 7.3 *Типы коллекций для атрибутов*

Порядок	Уникальность	Ассоциированный тип коллекции
False	False	Bag
True	True	OrderedSet

False	True	Set
True	False	Sequence

Например, чтобы показать, что клиенты банка должны представляться с использованием коллекции `OrderedSet`, атрибут clients моделируется так, как показано на рис. 7.22.

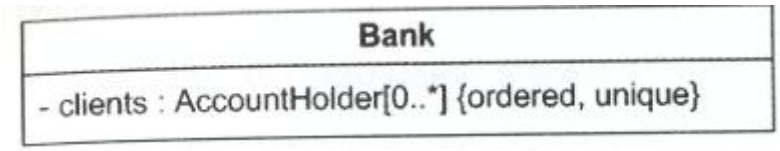


Рисунок 7.22 *Пример атрибута, хранимого в виде коллекции* `OrderedSet`

Свойства атрибутов

Кроме свойств, связанных с множественностью, атрибут может иметь свойства, передающие дополнительную информацию читателю диаграмм. В `UML` определяются следующие основные свойства:

- `readOnly` – означает, что атрибут не может модифицироваться после задания исходного значения. Обычно это соответствует *константе* в языке разработки. UML не указывает, когда именно должно задаваться исходное значение, но если для атрибута задано значение по умолчанию, оно считается исходным, и в дальнейшем не изменяется;
- `union` – означает, что тип атрибута представляет собой объединение возможных значений этого атрибута. Довольно часто используется со свойством derived, означающим что атрибут является производным объединением другого множетсва атрибутов;
- `subset`<*имя-атрибута*> – означает, что тип атрибута является подмножеством всех допустимых значений некоторого атрибута. Свойство не является основным, но в случае использования оно обычно ассоциируется с субклассами типа атрибута;
- `redefines`<*имя-атрибута*> – означает, что атрибут действует как псевдоним для заданного атрибута. Атрибут не является основным, но он, например, может показывать, что атрибут субкласса является псевдонимом для атрибута суперкласса;
- `composite` – указывает, что атрибут является частью отношения «целое-часть» с классификатором.

Ограничения

Ограничения, устанавливаемые для элементов, записываются на естественном языке или с использованием формальной грамматики вроде OCL; результатом вычисления должно быть логическое выражение. Ограничения обычно указываются в фигурных скобках (`{}`) после ограничиваемого элемента, хотя они также могут размещаться в примечаниях, соединенных с элементом пунктирной линией.

У именованных ограничений имена указываются перед логическим выражением и отделяются от него двоеточием (`:`). Данная возможность часто используется для идентификации ограничений, связанных с операциями.

На рис. 7.23 показаны примеры ограничений атрибутов и операций.

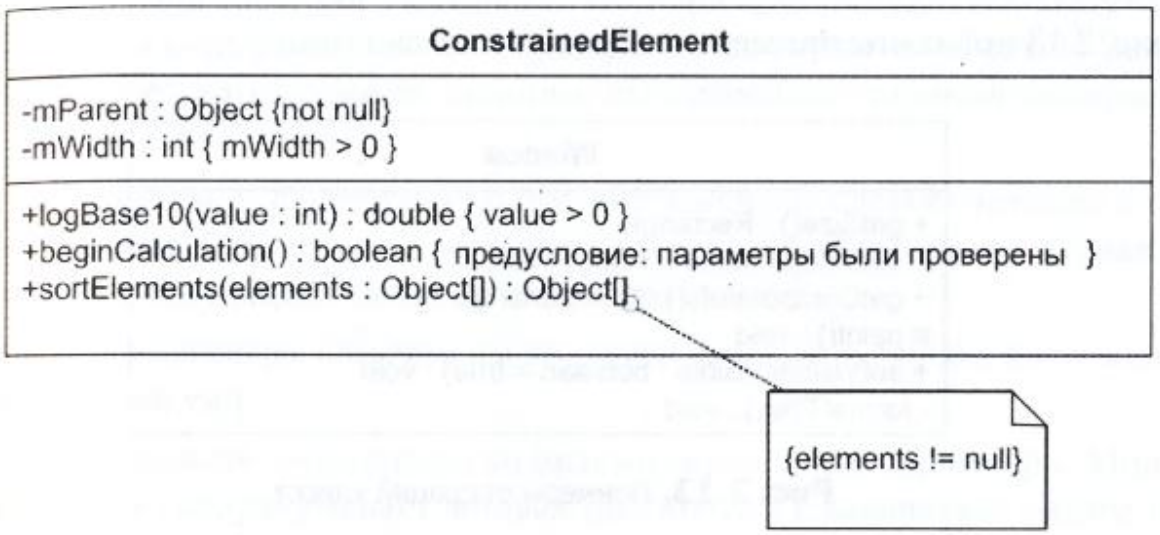


Рисунок 7.23 *Примеры ограничений: как встроенных, так и оформленных в виде примечаний*

Статические атрибуты

Статические атрибуты связываются с классом, а не с его конкретным экземпляром. Например, вы можете инициализировать для класса набор констант, а затем использовать их во всех экземплярах класса. Статические атрибуты обозначаются подчеркиванием как во встроенной записи, так и в записи с отношениями, как показано на рис. 7.24.

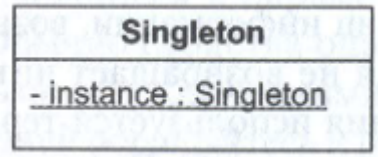


Рисунок 7.24 *Статический атрибут*

Операции

Операциями называются аспекты классов, определяющие активизацию некоторого поведения. Например, класс может поддерживать операцию для рисования прямоугольника на экране или подсчета количества элементов в списке. В UML существует четкое различие между описанием способа активизации поведения (операция) и фактической реализацией этого поведения (метод).

```
видимость имя ( параметры ) : возвращаемый_тип { свойства }
```

где *параметры* записываются в следующем виде:

```
направление имя_параметра : тип [ множественность ]
  = по_умолчанию { свойства }
```

На рис. 7.25 показаны примеры операций класса.

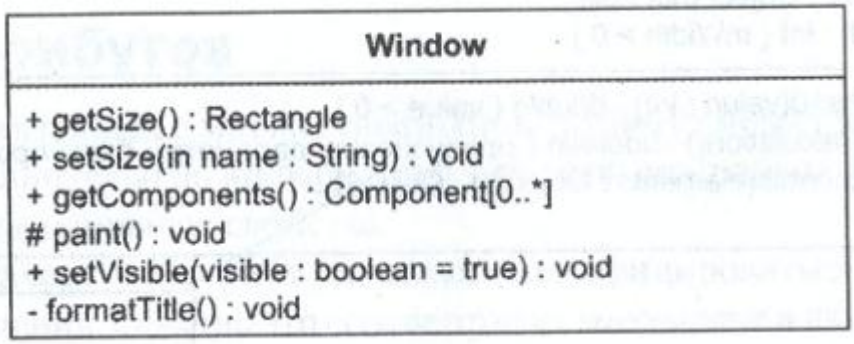


Рисунок 7.25 *Примеры операций класса*

Элементы синтаксиса операций:

- *видимость* – признак видимости операций. Знаки `+`, `-`, `#` или `~` соответствуют уровням `public`, `private`, `protected` и `package` соответсвенно.
- *имя* – короткая фраза с описанием операции. Обычно операции соответствует глагольное словосочетание, представляющее действия, выполняемые классификатором по поручению вызывающей стороны. Спецификация UML рекомендует, чтобы имя операции начиналось со строчной буквы, а все последующие начинались с прописных букв и записывались слитно. Пример показан на рис. 7.25.
- *возвращаемый_тип* – тип информации, возвращаемой операцией, если она имеется. Если операция не возвращает никакого значения (в некоторых языках программирования используется термин *процедура*), в качестве возвращаемого типа указывается тип `void`. Если операция возвращает значение (в некоторых языках программирования используется термин *функция*), указывается его тип – другой классификатор, примитивный тип или коллекция. В спецификации UML указано, что спецификация возвращаемого типа не является обязательной. Если он не указан, не следует делать никаких предположений относительно возвращаемого значения операции (даже если оно существует);
- *свойства* – ограничения и свойства, связанные с операцией. Не являются обязательными; при отсутствии свойств фигурные скобки не отображаются на диаграмме.

Элементы синтаксиса параметров:

- *направление* – необязательная часть синтаксиса, которая указывает, как параметр используется операцией. Допустимые значения – `in`, `inout`, `out` или `return`. Значение `in` указывает, что параметр передается операцией вызывающей стороной. Значение `inout` указывает, что параметр передается вызывающей стороной, а затем (после возможной модификацией операцией) возвращается обратно. Значение `out` указывает, что параметр задается вызывающей стороной, но модифицируется операцией и возвращается обратно. Значение `return` означает, что значение, заданное вызывающей стороной, возвращается обратно;

- *имя_параметра* – существительное или именное словосочетание с описанием атрибута. Обычно имя начинается со строчной буквы, но первая буква каждого последующего слова является прописной;
- *тип* – тип параметра. Обычно им является класс, интерфейс, коллекция или примитивный тип;
- *множественность* – количество экземпляров типа параметра. Может не указываться (подразумевает множественность 1), задаваться одним целым числом, списком целых чисел, разделенных запятыми, или в виде диапазона значений в квадратных скобках, с разделением границ «`..`». символ «`*`» а качестве верхней границы диапазона соответствует максимальному значению, а сам по себе обозначает 0 и более.
- *по_умолчанию* – значение параметра по умолчанию. Наличие значения по умолчанию не обязательно. Если оно не задано, знак = не отображается на диаграмме. UML не указывает, допускается ли отсутствие параметра со значением при вызове операций (как, например, в реализации значений параметров по умолчанию C++). Фактически синтаксис вызова операции зависит от конкретного языка;
- *свойства* – свойства, относящиеся к параметру. Свойства заключаются в фигурные скобки `{ }`. Обычно свойства определяются в контексте конкретной модели за несколькими исключениями: `ordered`, `readOnly` и `unique`. Наличие свойства для параметра необязательно. Если они не используются, фигурные скобки на диаграмме не отображаются.

Ограничения операций

С операцией могут связываться ограничения, определяющие правила взаимодействия операции с остальными компонентами системы. Ограничения операции в совокупности устанавливают контракт, который должен соблюдаться реализацией операции. Ограничения операций подчиняются стандартной записи ограничений и следует либо непосредственно за сигнатурой операции, либо в примечании, соединенном пунктирной линией.

Предусловия

Предусловия определяют, в каком состоянии должна находиться система перед началом операции. Кончено, в практическом контексте речь идет не о выражении состояния *всей* системы. Всего этого предусловия обычно выражают действительные значения параметров, состояние класса, владеющего операцией, или ключевые атрибуты системы.

В спецификации однозначно указано, что операция обязана проверять предусловия в теле операции перед выполнением; теоретически операция вообще не должна активизироваться при соблюдении предусловий. На практике лишь немногие языки предоставляют подобную защиту. Если вы потратили усилия на выражение предусловий, обычно в ваших же интересах проверить из правильность в реализации операции.

Предусловия дают вам как разработчику одну из нескольких возможностей «подстраховаться» и точно указать, как именно должно активизироваться поведение вашей реализации; пользуйтесь ими.

На рис. 7.26 показаны примеры предусловий.

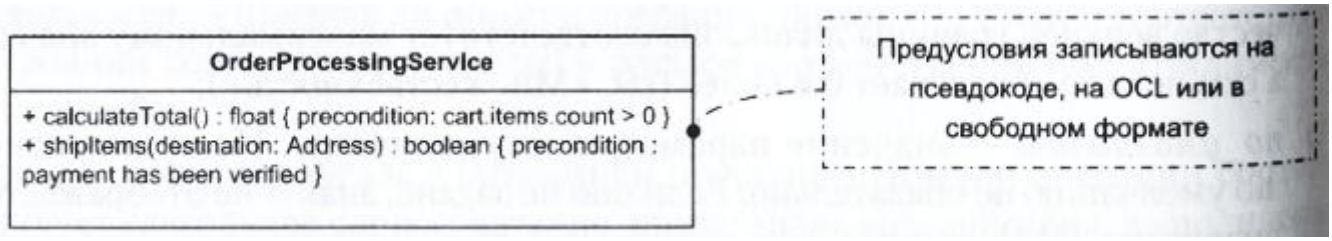


Рисунок 7.26 *Предусловие для вызова операций*

Постусловия

В постусловиях отражаются гарантии, относящиеся к состоянию системы после выполнения операции. Постусловия, как и предусловия, обычно выражают состояние одного или нескольких ключевых атрибутов или представляют некоторые гарантии относительно состояния класса-владельца операции.

На рис. 7.27 показаны примеры постусловий, ассоциированных с операцией.

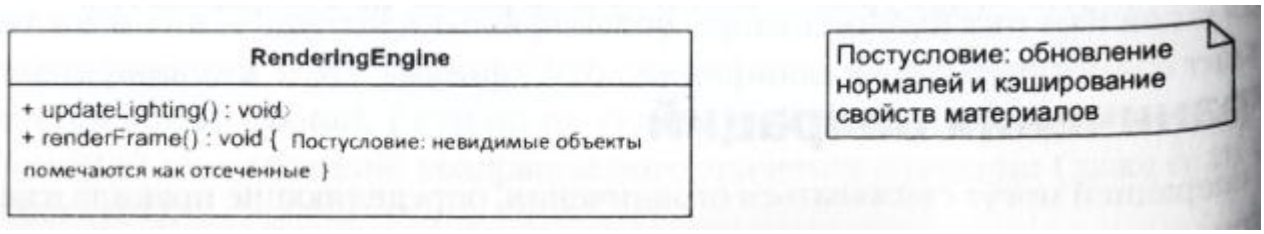


Рисунок 7.27 *Постусловия операций*

Условия `bodyCondition`

Операция может иметь условие `bodyCondition`, ограничивающее возвращаемое значение. Не путайте условие `bodyCondition` с постусловием – `bodyCondition` может заменяться методами субклассов класса-владельца. Например, класс с именем `Window` может установить для метода `getSize()` условие `bodyCondition`, которое требует, чтобы ширина и высота окна были отличны от нуля. Субкласс `SquareWindow` может предоставить собственное условие `bodyCondition`, согласно которому ширина окна должна совпадать с его высотой. По аналогии с пред- и пост- условиями, условия `bodyCondition` могут выражаться на естественном языке или на `OCL`. На рис. 7.28 показан пример условия `bodyCondition` в операции.

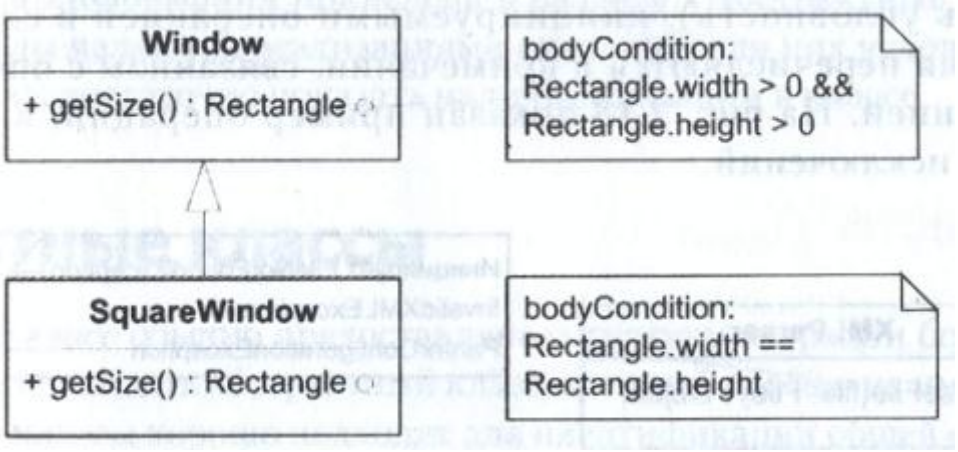


Рисунок 7.28 *Условия* `bodyCondition` *для операций*

Запросы

Операция может быть объявлена как *операция запроса*; это означает, что ее реализация никак не изменяет класса-владельца. На практике разработчики моделей часто используют запросы для обозначения методов, не изменяющих «осмысленные» атрибуты объекта. Например, какие-нибудь внутренние кэшированные атрибуты могут обновляться в результате запроса. Важно то, что состояние системы с внешней точки зрения не изменяется в результате запроса; вызов метода не должен иметь побочных эффектов.

Запрос обозначается ограничением `query` после сигнатуры операции. Например, операция `getAge()`, которая просто возвращает целое число без изменения внутренних данных класса-владельца, будет считаться методом запроса. В C++ это обычно соответствует `const`-методу. На рис. 7.29 показаны примеры методов запроса в классе.

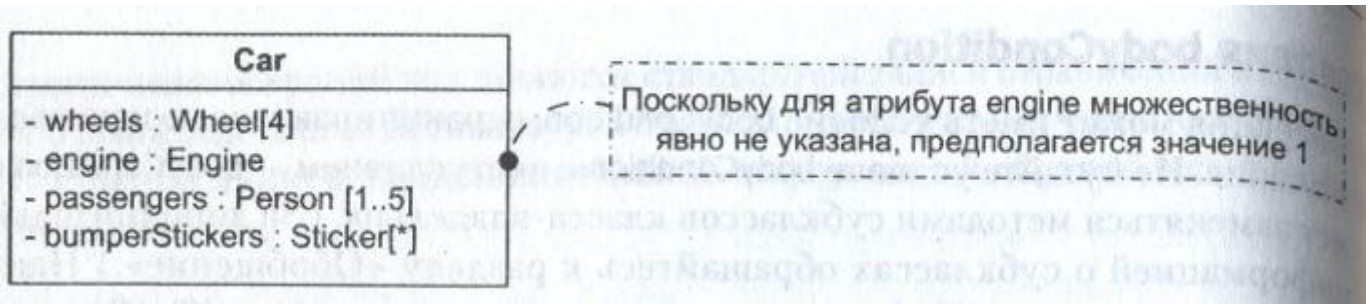


Рисунок 7.29 *Примеры операций запроса*

Исключения

Для выражения исключений, инициируемых операцией, используется исходная запись (хотя с технической точки зрения исключения не являются ограничениями). Обычно исключения представляются специальными классами (которые часто обозначаются ключевым словом `exception`, но это всего лишь условность), инициируемых операцией в случае ошибки. Исключения перечисляются в примечании,

связанном с операцией пунктирной линией. На рис. 7.30 показан пример операции, инициирующей несколько исключений.

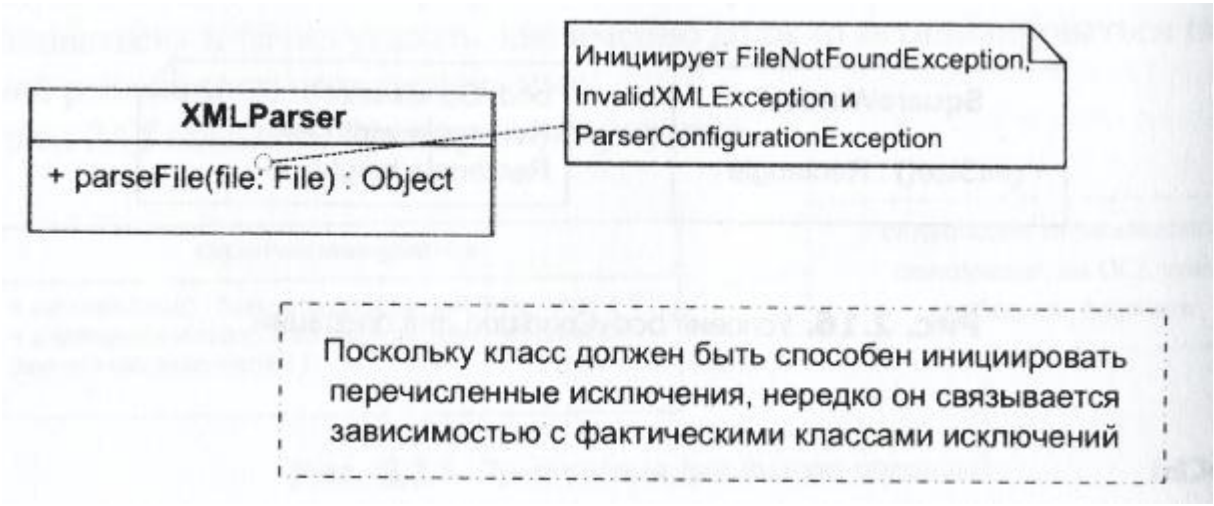


Рисунок 7.30 *Метод, инициирующий несколько исключений*

Статические операции

Обычно операции задают поведение уровня *экземпляра* класса. Тем не менее UML позволяет задавать поведение на уровне *самого класса*. Такие операции называются *статическими операциями* и вызываются непосредственно для класса, а не для экземпляра. Статические операции часто используются для выполнения вспомогательных действующих, не требующих использования атрибутов класса-владельца. В UML отсутствует формальная спецификация записи таких операций, но на практике они часто представляются по той же схеме, что и статические атрибуты. Чтобы указать, что операция является статической, подчеркните сигнатуру операции. На рис. 7.31 показан пример статической операции.

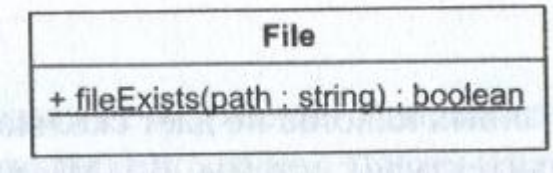


Рисунок 7.31 *Класс со статической операцией*

Методы

Метод представляет собой реализацию операции. Каждый класс обычно представляет реализации своих операций или наследуют их от суперкласса. Если класс не предоставляет реализацию операции и последняя наследуется от суперкласса, операция считается *абстрактной*. Поскольку методы являются реализациями операции, для них условная запись не предусмотрена; достаточно показать наличие операции в классе.

Абстрактные классы

Абстрактный класс обычно предоставляет сигнатуру операции без реализации; впрочем, можно создать абстрактный класс, который вообще не имеет операций.

Абстрактные классы хорошо подходят для идентификации общей функциональности нескольких типов объектов. Предположим, имеется абстрактный класс с именем `Movable`. Объект `Movable` имеет текущую позицию и может быть перемещен куда-либо операцией `move()`. Возможны разные специализации этого абстрактного класса – `Car`, `Grasshoper`, `Person` и т.д., каждая из которых представляет свою реализацию `move()`. Так как базовый класс Movable не содержит реализации `move()`, этот класс называется абстрактным.

Имена абстрактных классов записываются курсивом. Абстрактные операции также помечаются курсивным начертанием. На рис. 7.32 показан пример абстрактного класса `Movable`.

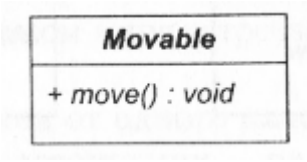


Рисунок 7.32 *Абстрактный класс*

Создать экземпляр абстрактного класса невозможно. Сначала на основе абстрактного класса необходимо создать субкласс, предоставляющий реализацию операции, а уже затем создать экземпляр этого класса.

Отношения

Перечисление изолированных классов не дает сколько-нибудь содержательного представления об архитектуре системы. В `UML` предусмотрено несколько способов представления отношений между классами. Каждое отношение `UML` соответствует некоторому способу логической связи между классами и обладает своими нюансами, которые не полностью отражаются в спецификации `UML`. При моделировании связей из реального мира позабыть о том, чтобы предполагаемая аудитория понимала, какую информацию вы хотите передать тем или иным отношениям. Пусть эта фраза станет предупреждением для разработчика и одновременно небольшой оговоркой относительно следующих объяснений: они всего лишь представляют наше видение спецификаций `UML`. Например, споры по поводу того, когда следует применять агрегирование или композицию, не утихают по сей день. Чтобы вам было проще определить, для какой ситуации лучше подойдет то или иное отношение, мы приводим для каждого типа короткую формулировку; она поможет вам в принятии решения. Стоит ещё раз подчеркнуть, что главное – логическая согласованность с моделью.

Зависимость

Самая слабая разновидность отношений между классами – отношение *зависимости*. Наличие зависимости между классами означает, что один класс использует другой класс, или располагает информацией о его существовании. Обычно эта связь является краткосрочной: зависимый класс быстро взаимодействует с целевым классом, но не нуждается в поддержании связи на сколько-нибудь продолжительное время.

Зависимости обычно выражаются формулировкой «…использует…». Например, если класс Window при закрытии инициирует класс-событие WindowClosingEvent, можно сказать «Window использует WindowClosingEvent».

Зависимость между классами выражается пунктирной линией со стрелкой, ведущей от зависимого класса к используемому. На рис. 7.33 показана зависимость между классом Window и классом WindowCursorEvent.

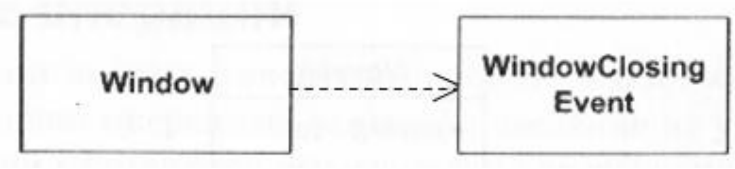


Рисунок 7.33 *Зависимость Window от WindowCursorEvent*

На рис. 7.33 можно с большой вероятностью предположить, что Window не поддерживает отношения с WindowCursorEvent в течение сколько-нибудь продолжительного времени. Класс просто использует объект события, а потом «забывает» о нем.

Зависимость обозначает отношение между двумя или более элементами модели, при котором изменение одного элемента (поставщика) может повлиять или предоставить информацию, необходимую другому элементу (клиенту). Иначе говоря, клиент некоторым образом зависит от поставщика. Зависимости используются для моделирования отношений между классификаторами, когда один классификатор зависит от другого, но отношение не является ни ассоциацией, ни обобщением.

Например, объект одного класса передается как параметр в операцию объекта другого класса. Очевидно, что между классами этих объектов существует отношение некоторого рода, но это не совсем обычная ассоциация. Отношение зависимости (специализированное некоторыми предопределенными стереотипами) можно использовать как универсальное средство для моделирования данного вида отношений. UML 2 определяет три основных типа зависимостей (табл. 7.4). Мы включили их в обсуждение для полноты информации, но в повседневном моделировании редко используется что либо кроме простой пунктирной стрелки зависимости. Обычно разработчики моделей не утруждают себя определением типа зависимости.

Таблица 7.4

Тип	Семантика
Usage (Использование)	Клиент использует некоторые из доступных сервисов поставщика для реализации собственного поведения; это самый распространенный тип зависимости.
Abstraction (Абстракция)	Обозначает отношение между клиентом и поставщиком, где поставщик более абстрактный, чем клиент. Что подразумевается под «более абстрактный»? Это может означать, что поставщик находится на другой стадии разработки, чем клиент (например, в аналитической модели, а не в проектной модели).
Permission (Доступ)	Поставщик предоставляет клиенту разрешение на доступ к своему содержимому – это дает возможность поставщику контролировать и ограничивать доступ к своему содержимому.

Рисунок 7.34 *Типы зависимостей*

Зависимость «use»

Самым распространенным стереотипом зависимости является «use», который просто обозначает, что клиент каким-то образом использует поставщика. Если на диаграмме указана просто пунктирная линия со стрелкой зависимости без стереотипа, можно быть совершенно уверенным, что подразумевается зависимость «use».

На рис. показаны два класса, A и B, между которыми установлено отношение зависимости «use». Эта зависимость генерируется любой из следующих ситуаций.

1. Операции класса A необходим параметр класса B.
2. Операция класса A возвращает значение класса B.

Рисунок 7.35

3. Операция класса A где-то в своей реализации использует объект класса B, но не в качестве атрибута.

Варианты 1 и 2 довольно просты, а вот вариант 3 представляет больший интерес. Такая ситуация возможна, если одна из операций класса A создала временный объект класса B.

Хотя одна зависимость «use» может использоваться как универсальная для всех трех перечисленных случаев, есть и другие, более специализированные стереотипы зависимостей, которые можно было бы применить.

Зависимость «call» устанавливается между операциями – операция клиент вызывает операцию поставщик

Зависимость «parameter» Поставщик является параметром операции клиента.

Зависимость «send» Клиент – это операция, посылающая поставщика (который должен быть сигналом) в некоторую неопределенную цель

Зависимость «instantiate» Клиент – это экземпляр поставщика.

Зависимости абстракции Зависимости абстракции моделируют зависимости между сущностями, находящимися на разных уровнях абстракции. В качестве примера можно привести класс аналитической модели и тот же класс в проектной модели. Существует четыре зависимости абстракции: «trace», «substitute», «refine» и «derive».

Зависимость «trace» Зависимость «trace» часто используется, чтобы проиллюстрировать отношение, в котором поставщик и клиент представляют одно понятие, но находятся в разных моделях. Например, поставщик и клиент могут находиться на разных стадиях разработки. Поставщик мог бы быть аналитическим представлением класса, а клиент – более детальны проектным представлением. Также «trace» можно использовать, чтобы показать отношение между функциональным требованием, таким как «банкомат должен обеспечивать возможность снятия наличных денег вплоть до достижения кредитного лимита карты», и прецедентом, поддерживающим это требование.

Зависимость «substitute» Зависимость «substitute» (заместить) показывает, что клиент во время выполнения может заменять поставщика. Замещаемость основывается на общности контрактов и интерфейсов клиента и поставщика, т. е. они должны предоставлять один и тот же набор сервисов.

Зависимость «refine» Тогда как зависимость «trace» устанавливается между элементами разных моделей, «refine» (уточнить) может использоваться между элементами одной и той же модели. Например, в модели может быть две версии класса, одна из которых оптимизирована по производительности. Поскольку оптимизация производительности является разновидностью уточнения, это отношение между двумя классами можно смоделировать как зависимость «refine» с примечанием, описывающим суть уточнения.

Зависимость «derive» Стереотип «derive» (получить) используется, когда необходимо явно показать возможность получения одной сущности как производной от другой.

Например, в имеющемся классе BankAccount есть список Transaction (транзакция), в котором каждая Transaction содержит Quantity (количество) денег. По требованию всегда можно вычислить текущий баланс, суммируя Quantity

по всем Transaction. Существует три способа показать, что balance (баланс) счета (его Quantity) может быть производной сущностью.

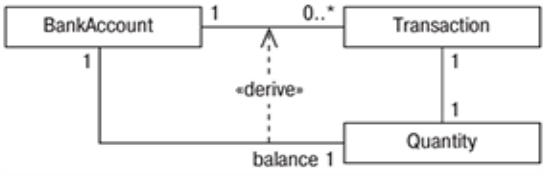
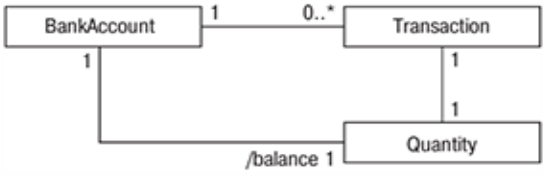
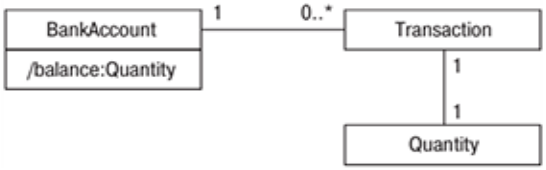
Модель	Описание
	Класс BankAccount имеет «derive» ассоциацию с Quantity, где Quantity играет роль баланса банковского счета. Эта модель подчеркивает, что balance является производным от коллекции Transaction класса BankAccount.
	В данном случае в имени роли используется слэш, чтобы показать, что между BankAccount и Quantity установлено отношение «derive». Такое обозначение менее явное, поскольку не показывает, производным чего является balance.
	Здесь balance показан как производный атрибут, что обозначено слэшем, предваряющим имя атрибута. Это самое краткое выражение зависимости «derive».

Рисунок 7.36

Зависимости доступа

Зависимости доступа выражают способность доступа одной сущности к другой. Существует три зависимости доступа: «access», «import» и «permit».

Зависимость «access» Зависимость «access» (доступ) устанавливается между пакетами. В UML пакеты используются для группировки сущностей. Самое главное здесь то, что «access» разрешает одному пакету доступ ко всему открытому содержимому другого пакета. Однако каждый пакет определяет пространство имен, и с установлением отношения «access» пространства имен остаются изолированными.

Зависимость «import» Зависимость «import» концептуально аналогична «access», за исключением того, что пространство имен поставщика объединяется с пространством имен клиента. Это обеспечивает возможность элементам клиента организовывать доступ к элементам поставщика без необходимости указывать в именах элементов имя пакета. Однако иногда это может приводить к конфликтам имен, если имена элемента клиента и элемента поставщика совпадают.

Зависимость «permit» Зависимость «permit» (разрешить) обеспечивает возможность управляемого нарушения инкапсуляции, но в целом этого отношения следует избегать. Клиентский элемент имеет доступ к элементу поставщику не зависимо от объявленной видимости последнего. Часто зависимость «permit» устанавливается между двумя родственными классами, когда клиентскому классу выгодно (вероятно, по причинам производительности) иметь доступ к закрытым членам поставщика. Не все языки программирования поддерживают зависимости «permit». C++ позволяет классу

объявлять друзей, которые имеют разрешение на доступ к его закрытым членам. Но эта возможность была (и, наверное, благоразумно) изъята из `Java` и `C#`.

Ассоциация

Отношение *ассоциации* сильнее отношения зависимости. Обычно оно показывает, что один класс поддерживает отношение с другим классом в течение длительного периода. Жизненные циклы двух объектов, скорее всего, не зависят друг от друга (то есть один объект может быть уничтожен без обязательного уничтожения второго).

Ассоциации обычно выражаются формулировкой «…содержит…» например, если класс `Window` содержит ссылку на текущий указатель мыши, можно сказать «`Window` содержит `Cursor`». Обратите внимание на тонкое различие между формулировками «…содержит…» и «…является владельцем…». В данном случае `Window` не является владельцем `Cursor`; `Cursor` совместно используется всеми приложениями в системе. Однако `Window` содержит ссылку на объект указателя, чтобы класс `Window` мог скрыть его, изменить его внешний вид и т.д. Ассоциации обозначаются сплошной линией между классами, участвующими в отношениях. На рис. 3.37 изображена ассоциация между `Window` и `Cursor`.

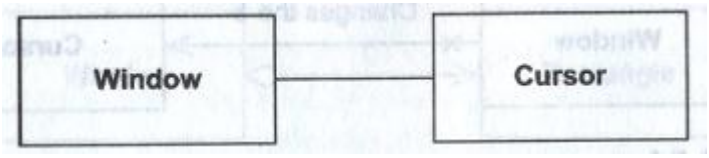


Рисунок 7.37 *Ассоциация «`Window` содержит `Cursor`»*

Очень распространено явление, когда класс имеет ассоциацию с самим собой. Это называется рефлексивной ассоциацией и означает, что объекты данного класса имеют связи с другими объектами этого же класса.

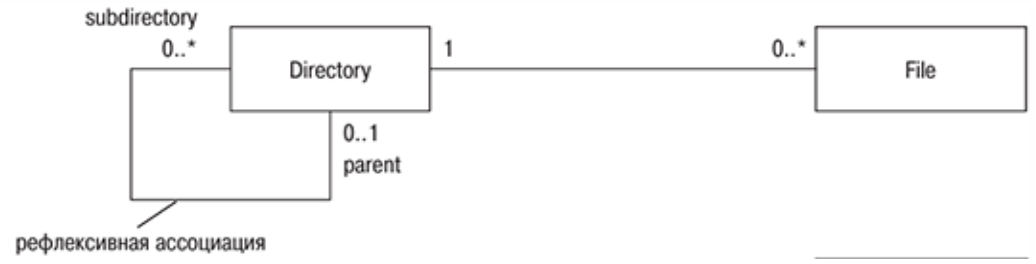


Рисунок 7.38 *Диаграмма классов*

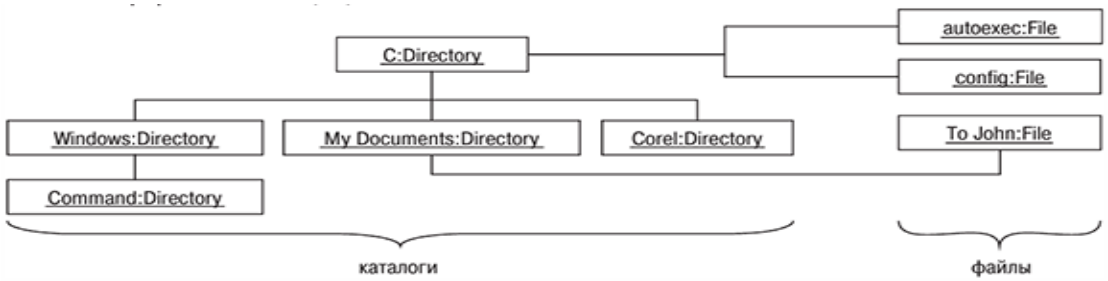


Рисунок 7.39 *Диаграмма объектов*

Направленные переходы

У ассоциации предусмотрена запись для выражения возможности перехода. В случае возможности одностороннего перехода от одного класса к другому на диаграмме рисуется стрелка в направлении того класса, к которому осуществляется переход. Если переход может производиться в обоих направлениях, обычно стрелки не показываются на диаграмме, но как указано в спецификации `UML`, в случае удаления всех стрелок вы не сможете отличить ассоциации без возможности перехода от двусторонней ассоциации. Впрочем, ассоциации без возможности перехода в реальном мире встречаются крайне редко, так что это вряд ли создаст проблемы.

Чтобы явно запретить переход от одного класса у другому, разместите маленький значок `X` на линии ассоциации – рядом с тем классом, переход к которому невозможен. На рис. 7.40 показана ассоциация между классом `Window` и классом `Cursor`. Поскольку от экземпляра `Cursor` невозможно перейти к экземпляру `Window`, этот факт явно обозначается стрелкой и `X` на соответствующем конце линии.

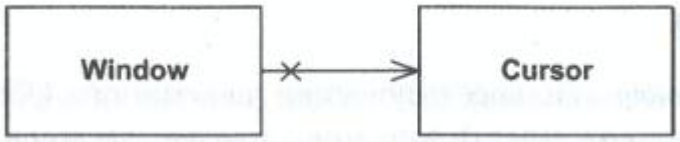


Рисунок 7.40 *Ассоциация между `Window` и `Cursor` показывает, что переход от `Cursor` к `Window` невозможен*

Именованные ассоциации

Ассоциации могут быть дополнены несколькими символами, добавляющими информацию к вашей модели. Простейший вариант – треугольник, обозначающий направление, в котором пользователь должен читать ассоциацию. Часто вместе с треугольником приводится короткая фраза с описанием контекста ассоциации. Она не преобразуется в какую-либо форму представления кода и предназначается исключительно для целей моделирования. На рис. 7.41 показан треугольник над ассоциацией `Window` и `Cursor`.

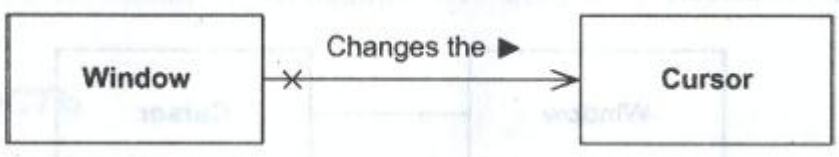


Рисунок 7.41 *Направление чтения ассоциации между `Window` и `Cursor`*

Свойства, относящиеся к множественности, могут применяться и к ассоциациям.

Множественность

Ассоциации обычно представляют долгосрочные отношения, поэтому они часто используются для обозначения атрибутов класса. Если множественность не указана, она предполагается равной 1. Чтобы задать другое значение, достаточно указать множественность рядом с «подчиненным» классом. Обратите внимание: при использовании множественности в ассоциациях значения не заключаются в квадратные скобки. На рис. показана ассоциация с явно заданной множественностью.

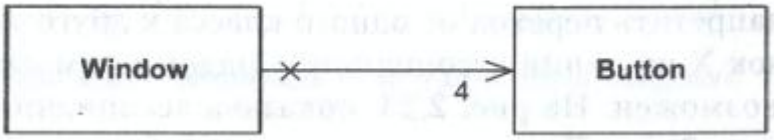


Рисунок 7.42 *Простая ассоциация показывает, что* `Window` *содержит 4 экземпляра* `Button`

Агрегирование

Агрегирование является усиленной разновидностью ассоциации. В отличие от ассоциации, агрегирование обычно подразумевает отношения владения, а также может подразумевать связи между жизненными циклами. Агрегирование обычно формулируется в виде «…является владельцем…» например, если класс `Window` хранит позицию и размер окна в классе `Rectangle`, можно сказать «`Window` является владельцем `Rectangle`». Экземпляр `Rectangle` может использоваться совместно с другими классами, но `Window` связан с `Rectangle` неразрывной связью. Между ней и базовой ассоциацией существует тонкое различие; в данном случае связь является более сильной. И все же агрегирование – не самый сильный вид связи, которая может существовать между классами. Если описание связи приближается к формулировке «класс `A` является частью класса `B`», стоит воспользоваться композицией.

Агрегирование обозначается ромбовидной стрелкой рядом с классом-владельцем и сплошной линией, ведущей к подчиненному классу. На рис. 7.43 показан пример агрегирования между классом `Window` и классом `Rectangle`.

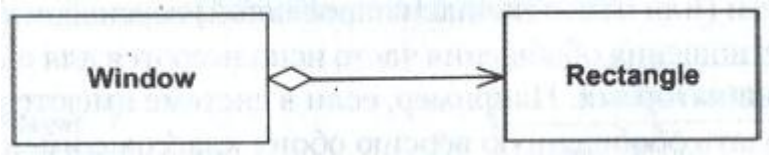


Рисунок 7.43 `Window` *«является владельцем»* `Rectangle`

Как для ассоциации, при агрегировании на диаграмме могут обозначаться возможности перехода и множественность.

Композиция

Композиция представляет очень сильные отношения между классами, граничащие с внедрением. Композиция отражает отношения типа «целое-часть». «Часть» в любой момент времени может участвовать только в одном отношении композиции. Жизненный цикл экземпляров, участвующих в отношениях композиции, почти всегда связан; уничтожение большего экземпляра-владельца почти всегда приводит к уничтожению

вложенного экземпляра. UML позволяет связать часть с другим владельцем перед ее уничтожением, в результате чего она продолжает существование, но подобная ситуация скорее является исключением, нежели правилом.

Отношение композиции обычно формулируется в виде «…является частью…», а это означает, что композиция читается от части к целому. Например, если окно в вашей системе должно иметь строку заголовка, это отношение может быть представлено классом Titlebar, который *является частью* класса Window.

Отношение композиции обозначается заполненным ромбом рядом с классом-владельцем и сплошной линией, ведущей к конечному классу. На рис. 7.44 показан пример композиции между классом Window и классом Titlebar.

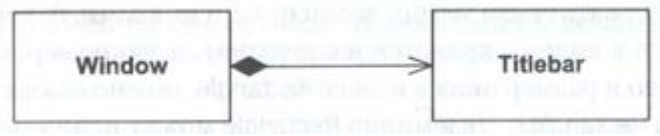


Рисунок 7.44 *Titlebar «является частью» Window*

Как и для ассоциаций, при композиции на диаграмме могут обозначаться возможности перехода и множественность.

Обобщение

Отношение *обобщения* указывает на то, что подчиненный класс отношения является более общей (или менее специализированной) версией исходного класса или интерфейса. Отношения обобщения часто используются для выявления сходства между классификаторами. Например, если в системе имеются классы Cat и Dog, вы можете создать обобщенную версию обоих классов с именем Animal.

Обобщения обычно читаются в виде «…является частным случаем…» от специализированного класса к общему. Возвращаясь к примеру Cat и Dog, можно сказать, что «Cat (кошка) является частным случаем Animal (животного)».

Отношения обобщения обозначается сплошной линией с замкнутой треугольной стрелкой, ведущей от специализированного класса к общему. На рис. 7.45 показан пример отношения между классами Cat и Animal.

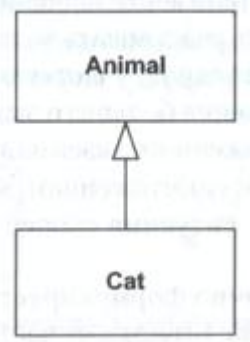


Рисунок 7.45 *Производный класс Cat является специализацией базового класса Animal*

В отличие от ассоциаций, связи обобщения обычно не бывают именованными, и не обладают множественностью. UML допускает множественное наследование, то есть наличие у класса нескольких обобщенных версий, каждая из которых представляет некоторый аспект произвольного класса. Однако некоторые современные языки (например, Java и C++) не поддерживают множественное наследование; вместо него применяется реализация интерфейсов.

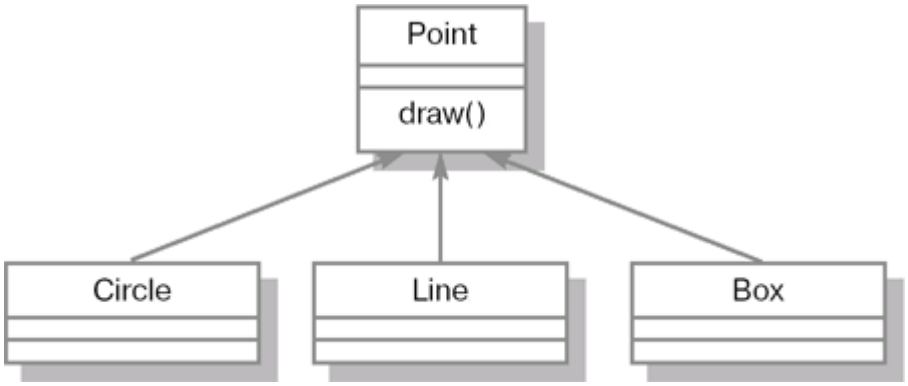


Рисунок 7.46

В иерархии обобщения (рис. 7.46) кроется наследование между классами, посредством которого подклассы наследуют все возможности своих надклассов. Чтобы быть более специальными, подклассы наследуют:

- *атрибуты;*
- *операции;*
- *отношения;*
- *ограничения.*

Подклассы также могут вводить новые возможности и переопределять операции суперкласса.

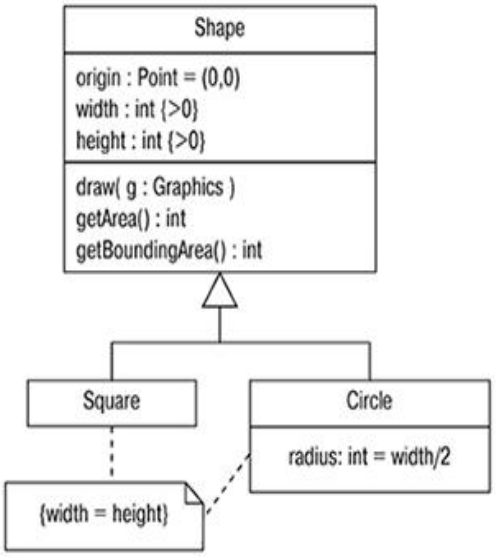


Рисунок 7.47

В примере на рис. 7.47 подклассы Square и Circle класса Shape наследуют все его атрибуты, операции и ограничения. Это означает, что хотя мы и не видим этих

элементов в подклассах, они присутствуют в них неявно. Говорят, что `Square` и `Circle` типа `Shape`.

Обратите внимание, что операции `draw()` (отрисовать) и `getArea()` (найти площадь), определенные в `Shape`, не подходят для подклассов. Посылая сообщение `draw()`, мы ожидаем, что объект `Square` отрисует квадрат, а объект `Circle` – круг. Очевидно, что стандартная операция `draw()`, унаследованная обоими подклассами от их родителя, не годится. Фактически данная операция может вообще ничего не отрисовывать.

Эти проблемы явно указывают на необходимость возможности изменения поведения суперкласса в подклассах. Классам `Square` и `Circle` надо реализовать собственные операции `draw()` и `getArea()`, которые переопределяют стандартные операции, предоставляемые родителем, и обеспечивают более подходящее поведение.

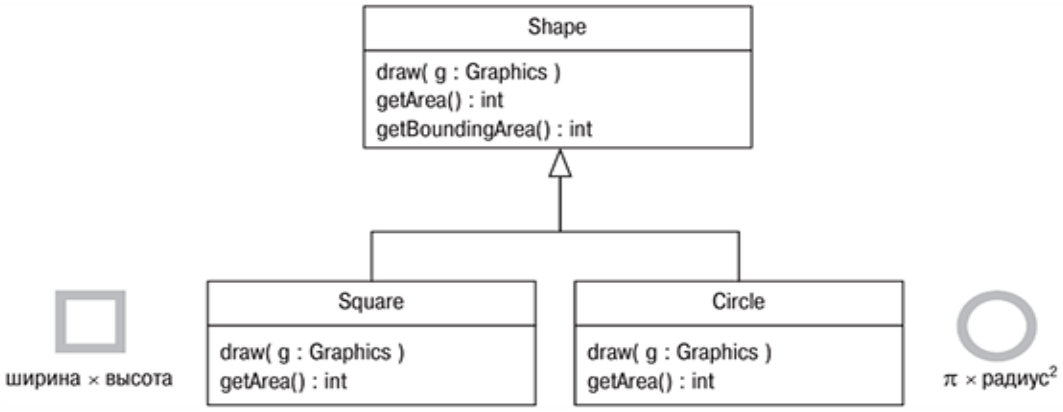


Рисунок 7.48

Классы-ассоциации

Часто отношение между двумя элементами не является простым структурным соединением. Например, футболист может быть связан с лигой посредством принадлежности к команде. Если ассоциация между двумя элементами имеет сложную природу, для ее представления можно воспользоваться *классом-ассоциацией* – ассоциацией, обладающей именем и атрибутами, как и обычный класс. Класс-ассоциация изображается как обычный класс, соединенный пунктирной линией с ассоциацией, которую от представляет. На рис. 7.49 показано отношение между классами `FootballPlayer` и `FootballLeague`.

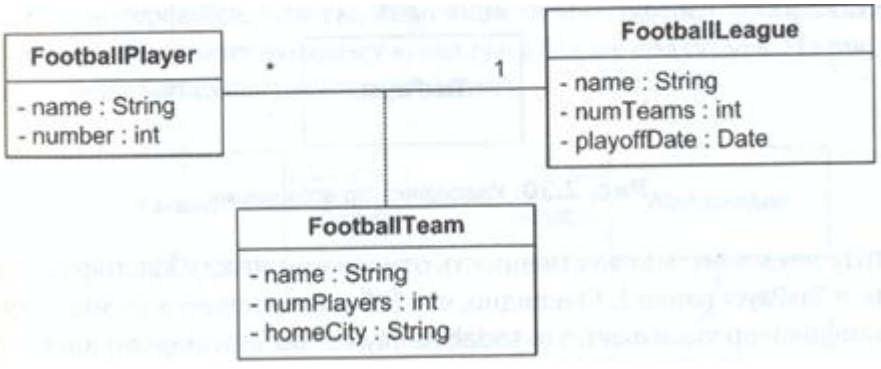


Рисунок 7.49 *Пример класса-ассоциации*

При переводе программный код отношения класса-ассоциациями часто воплощаются в виде трех классов: по одному для каждого участника ассоциации, и еще один для самого класса-ассоциации. Между участниками ассоциации может существовать прямая связь, а может и не существовать; реализация может потребовать, чтобы переход к другому концу связи происходил обязательно через класс-ассоциацию. Иначе говоря, `FootballPlayer` может и не содержать прямую ссылку на FootballLeague, а вместо этого на `FootballTeam`. В этом случае `FootballTeam` будет содержать ссылку на `FootballLeague`. Способ конструирования отношений выбирается на уровне реализации; тем не менее базовая концепция класса-ассоциации остается неизменной.

Экземпляры класса ассоциации – это на самом деле связи, у которых есть атрибуты и операции. Уникальная идентификация этих связей определяется исключительно индивидуальностью объектов, находящихся на каждом конце. Этот фактор ограничивает семантику класса ассоциации: его можно использовать только тогда, когда между двумя объектами в любой момент времени установлена единственная уникальная связь. Это обусловлено тем, что каждая связь, которая является экземпляром класса ассоциации, должна быть уникальной. На рис. применение класса ассоциации означает, что на модель накладывается следующее ограничение: для данного объекта `Person` и данного объекта Company может существовать только один объект `Job` (должность). Иначе говоря, каждый `Person` может занимать только одну `Job` в данной Company. Однако если ситуация такова, что данный объект `Person` может занимать несколько `Job` в данном объекте Company, класс ассоциацию использовать нельзя – семантика не соответствует!

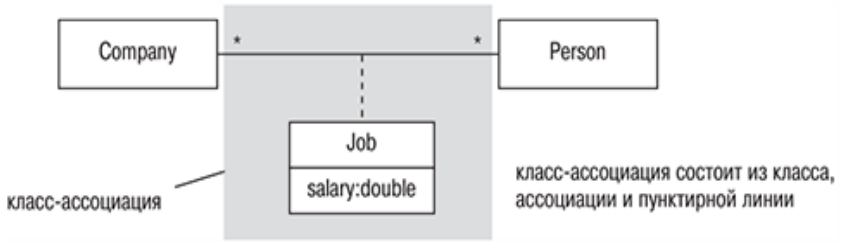


Рисунок 7.50

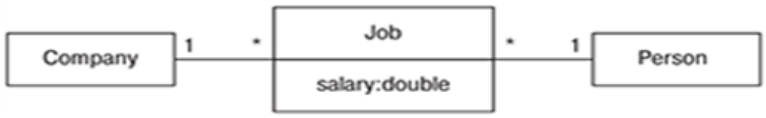


Рисунок 7.51

Откровенно говоря, многие разработчики объектных моделей просто не понимают семантической разницы между классами ассоциациями и материализованными

отношениями. Поэтому одни часто подменяются другими. Однако разница на самом деле очевидна: классы ассоциации могут использоваться только в том случае, когда каждая связь имеет уникальную индивидуальность. Надо просто запомнить, что индивидуальность связи определяется индивидуальностью объектов, располагающихся на ее концах.

Квалификаторы ассоциаций

Отношения между элементами часто индексируются по некоторому показателю. Например, клиент банка может идентифицироваться по номеру счета; а налогоплательщик – по номеру социального страхования. В UML для сохранения подобной информации предусмотрены *квалификаторы* ассоциаций. Квалификатор обычно является атрибутом целевого элемента, хотя это и необязательно. Он обозначается маленьким прямоугольником между ассоциацией и сходным элементом. В прямоугольник вписывается имя классификатора (обычно совпадающее с именем атрибута). На рис. 5.52 показано отношение между Налоговым управлением США (`IRS`) и налогоплательщиком, квалифицированное по номеру социального страхования налогоплательщика.

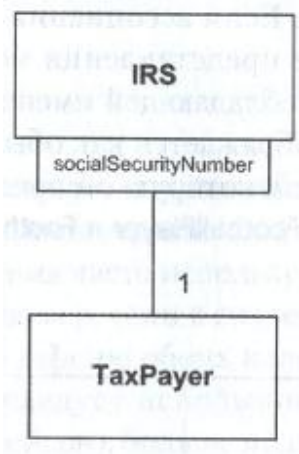


Рисунок 7.52 *Квалификатор ассоциации*

Обратите внимание: множественность отношения квалификатором ассоциации и `TaxPlayer` и равна `1`. Очевидно, что `IRS` ассоциируются со многими `TaxPlayer`, но квалификатор указывает, что `socialSecurityNumber` однозначно идентифицирует один экземпляр `TaxPlayer`.

Интерфейсы

Интерфейс представляет собой классификатор, содержащий объявления свойств и методов, но без реализации. Интерфейсы используются для группировки общих элементов между классификаторами и определения контракта, который должен соблюдаться реализацией интерфейса. Например, можно создать интерфейс `Sortable`, состоящий из единственной операции `comesBefore(…)`. Любой класс, реализующий интерфейс `Sortable`, должен предоставить реализацию `comesBefore(…)`.

Некоторые современные языки (например, C++) не поддерживают концепцию интерфейсов; в них интерфейсы UML обычно представляются в виде чисто абстрактных классов. Другие языки – такие как Java – поддерживают интерфейсы, но не разрешают включить в них свойства. Из этого следует, что при моделировании системы необходимо учитывать реализацию вашей модели.

Интерфейсы имеют два обозначения; выбор зависит от того, что именно вы собираетесь показать на своей диаграмме. Первое представление – стандартная запись, классификатора UML со стереотипом interface. На рис. показан интерфейс Sortable.

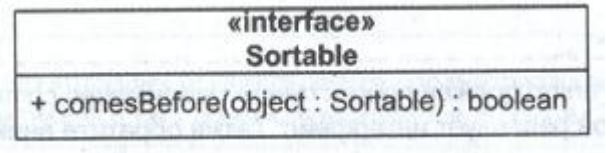


Рисунок 7.53 *Интерфейс Sortable*

Второе обозначение интерфейса известно под названием «шарнирной записи». Оно предоставляет меньше информации об интерфейсе, но лучше показывает отношения между классами. Интерфейс изображается в виде кружка, под которым пишется имя интерфейса. Классы, зависящие от интерфейса, изображаются присоединенными к соответствующему «гнезду» в форме полукруга. На рис. 7.54 показан интерфейс Sortable в шарнирной записи.

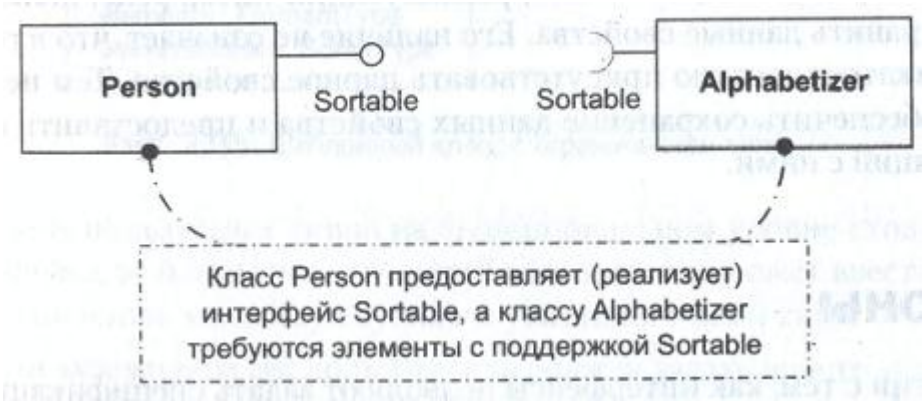


Рисунок 7.54 *Примеры предоставления и необходимости поддержки интерфейса*

Поскольку интерфейс задает контракт только для ограниченной функциональности, создать экземпляр интерфейс напрямую невозможно. Вместо этого класс должен *реализовать* интерфейс, то есть предоставить реализацию его операций и свойств. Реализация обозначается пунктирной линией, которая начинается от реализующего классификатора и ведет к интерфейсу, с заполненной треугольной стрелкой на конце. Классы, зависимые от интерфейса, обозначаются пунктирной линией с незаполненной стрелкой (зависимость). На рис. 7.55 показан класс, реализующий интерфейс Sortable, и класс, зависимый от него.

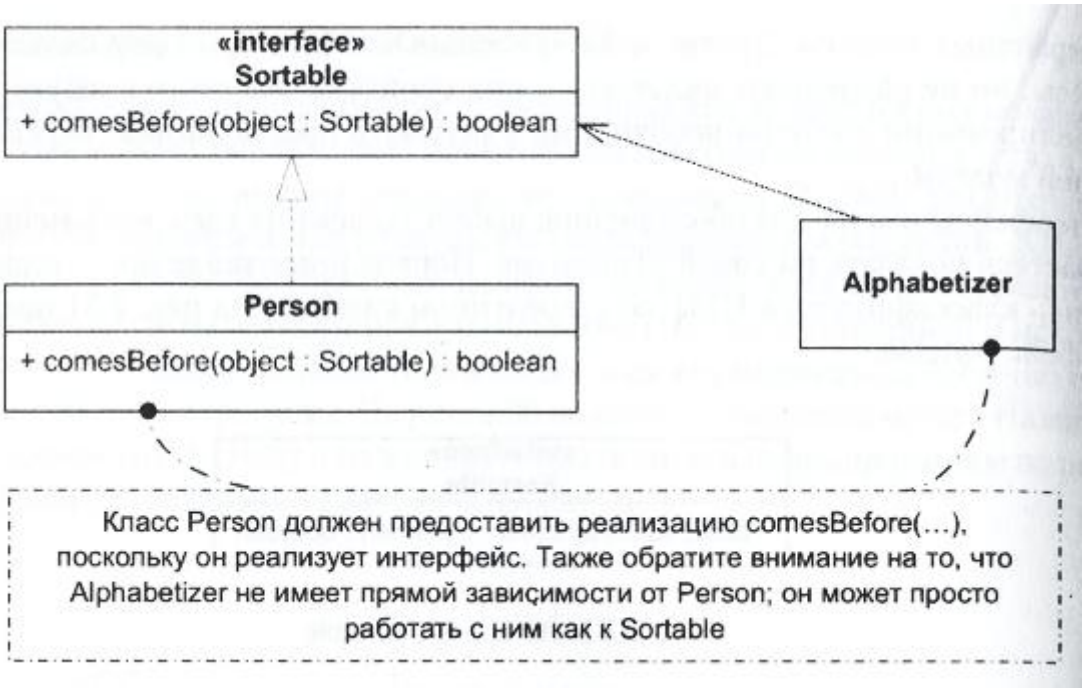


Рисунок 7.55 *Person реализует интерфейс* `Sortable`*, а* `Alphabetizer` *зависит от него*

Предоставить реализацию не сложно. В соответствующий классификатор включается реализация, имеющая ту же сигнатуру, что и операция интерфейса. Обычно с операцией связываются семантические ограничения, которые должны соблюдаться любой реализацией. Со свойствами дело обстоит чуть сложнее. Свойство интерфейса указывает, что класс, реализующий интерфейс, должен *каким-то образом* хранит данные свойства. Его наличие не означает, что в реализующей классификаторе должно присутствовать парное свойство. Тем не менее класс должен обеспечить сохранение данных свойства и предоставить средства для манипуляции с ними.

Шаблоны

По аналогии с тем, как интерфейсы позволяют задать спецификацию *объектов*, с которыми будет взаимодействовать ваш класс, `UML` позволяет предоставить абстракцию для типа *класса*, с которыми может взаимодействовать ваш класс. Например, можно создать класс `List` для хранения объектов произвольного типа (вероятно, в `C++` это будет `void*`, а в `Java` и `C#` – `Object`). Но если вы хотите, чтобы ваш класс `List` мог поддерживать объекты произвольного типа, все объекты в списке должны относиться к одному типу. Для создания абстракций такого рода в `UML` существуют *шаблоны*.

Чтобы указать, что класс является шаблонным (также используется термин «параметризованный»), поместите пунктирный прямоугольник в правом верхнем углу класса. Для каждого элемента, по которому производится параметризация, необходимо задать имя, которое будет заменять фактический тип. Запишите имя элемента-заполнителя в прямоугольнике. На рис. 7.56 показан пример класса `List` с произвольным типом элементов.

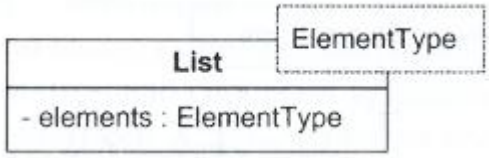


Рисунок 7.56 *Шаблонный класс* `List`

Один класс может быть параметризирован по нескольким типам; в этом случае их имена разделяются запятыми `(,)`. Если понадобится ограничить состав типов, которые могут подставляться пользователем на место заполнителя, поставьте за именем типа двоеточие `(:)`. На рис. 7.57 показана более сложная версия класса `List`, который вместе с типом объектов, хранимых в списке, передается `Sorter`.

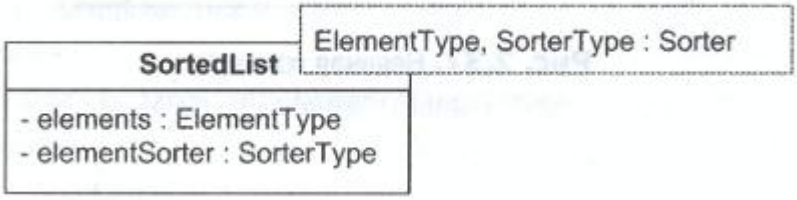


Рисунок 7.57 *Шаблонный класс с ограничением типа*

Ограничение используемых типов на функциональном уровне сходно с указанием интерфейса, за одним исключением: пользователь может ввести дополнительные ограничения, используя субкласс указанного вами типа.

При создании экземпляра `List` пользователь должен задать вместо условного заполнителя `ElementType` реальный тип. Это называется *привязкой* шаблона к типу. Привязка обозначается ключевым словом `bind`, за которым следует спецификация типа со следующим синтаксисом:

```
< ШаблонныйТип -> РеальныйТип >
```

Синтаксис привязки может использоваться при любых упоминаниях шаблонного класса, когда вы хотите указать на необходимость использования «специализированной» версии класса. Такой способ называется *явной привязкой*. Например, на рис. 7.58 показан субкласс `List` с именем `EmployeeList`, который подставляет на место заполнителя `ElementType` в шаблоне `List` класс `Employee`.

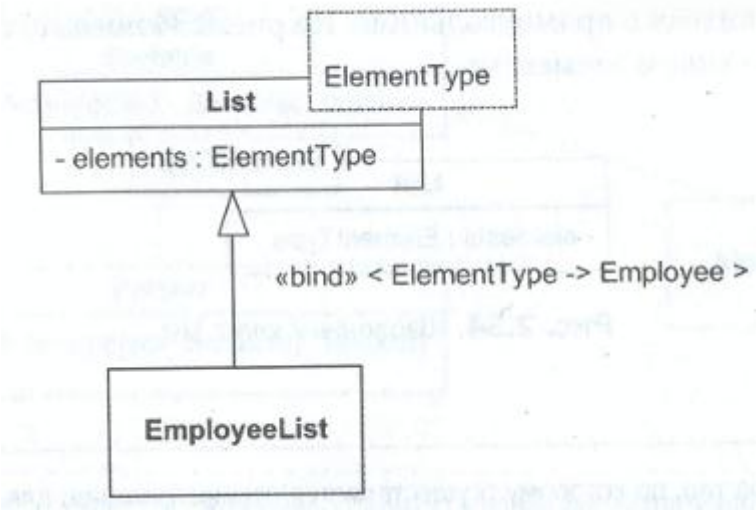


Рисунок 7.58 *Явная привязка*

Пример *неявной привязки*, при которой ключевое слово `bind` не указывается, показан на рис. 7.59.

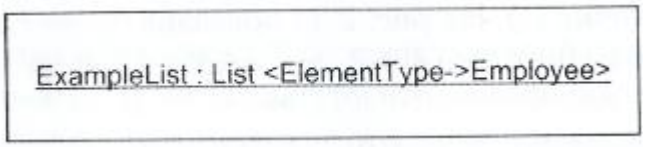


Рисунок 7.59 *Неявная привязка*

Множество всех типов

Множество всех типов – это класс, экземпляры которого являются классами. Эти экземпляры также являются подклассами другого класса. Любой класс, экземпляры которого являются классами, называют *метаклассом (класс класса)*. Таким образом, множество всех типов – это особый тип метакласса, экземпляры которого являются еще и подклассами другого класса.

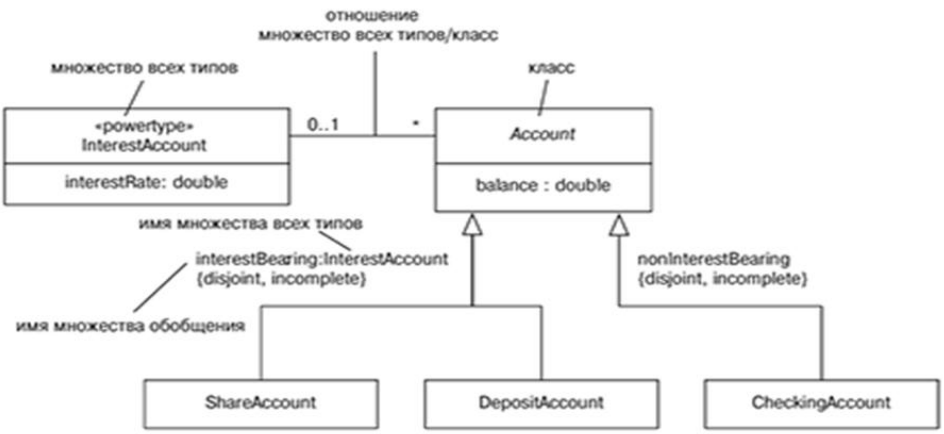


Рисунок 7.60 *Пример применения множества всех типов*

Ни один из основных ОО языков программирования не поддерживает множества всех типов. Как же тогда можно применить это понятие на практике? На рис. 7.60

показано простое решение этой проблемы, где она реализуется посредством делегирования. В этом примере для создания правильной иерархии наследования `AccountType` введены новые классы, `AccountType` и `NonInterestAccount` (беспроцентный счет). Типы каждого `Account` обозначены с помощью ограничений. Это довольно стандартный способ работы с множествами всех типов. Теоретически множества всех типов предоставляют лаконичную и удобную идиому моделирования для аналитических моделей. Однако на практике они не используются или не понятны широкому кругу разработчиков. Таким образом, в случае применения они могут привести к полному замешательству. Множества всех типов не вносят ничего нового в набор моделирования и, возможно, никогда не будут даже поддерживаться средствами моделирования. *Наш совет – избегайте их применения.*

Пакеты

Пакет – это группирующая сущность. Это контейнер и владелец элементов модели. У каждого пакета есть свое пространство имен, в рамках которого все имена должны быть уникальными.

По сути, пакет – это универсальный механизм организации элементов модели (включая другие пакеты) и диаграмм в группы. Он может использоваться для следующих целей:

- предоставления инкапсулированного пространства имен, в рамках которого все имена должны быть уникальными;
- группировки семантически взаимосвязанных элементов;
- определения «семантической границы» модели;
- предоставления элементов для параллельной работы и управления конфигурацией.

Пакеты позволяют создавать допускающую навигацию хорошо структурированную модель, обеспечивая возможность группировать сущности, имеющие близкие семантические связи. В модели можно устанавливать семантические границы, в пределах которых разные пакеты описывают разные аспекты функциональности системы.

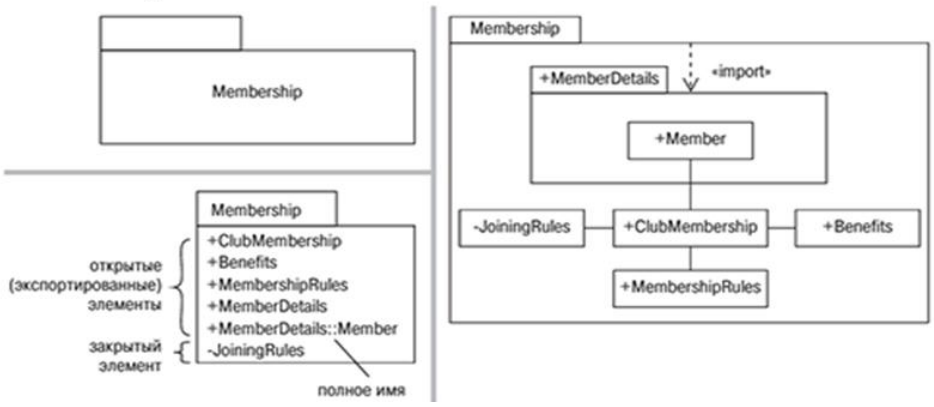


Рисунок 7.61 *Синтаксис пакета: три способа представления пакета на разных уровнях детализации*

Пакет определяет так называемое инкапсулированное пространство имен. Это означает, что пакет создает границу, в рамках которой имена всех элементов должны быть уникальными. Это также означает, что, если элементу из одного пространства имен необходимо обратиться к элементу из другого пространства имен, он должен указать и имя необходимого элемента, и путь к этому элементу, чтобы его можно было найти в пространствах имен. Этот путь навигации называют полным именем (`qualified name`) или составным именем (`pathname`) элемента.

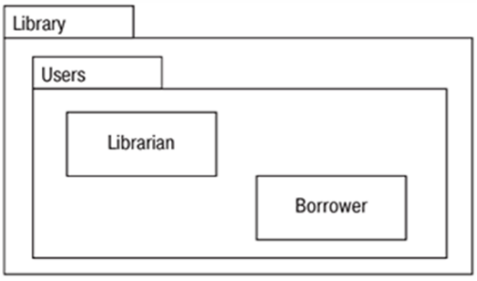


Рисунок 7.62 *Синтаксис вложения путем встраивания*

Пакеты могут быть вложены в другие пакеты с любой глубиной вложенности. Однако обычно достаточно всего двух или трех уровней. В противном случае модель может стать трудной для понимания и в ней будет сложно ориентироваться. UML предлагает два способа представления вложенности. Первый очень нагляден, поскольку в нем элементы модели показаны физически вложенными в пакет. Пример представлен на рис. 7.62.

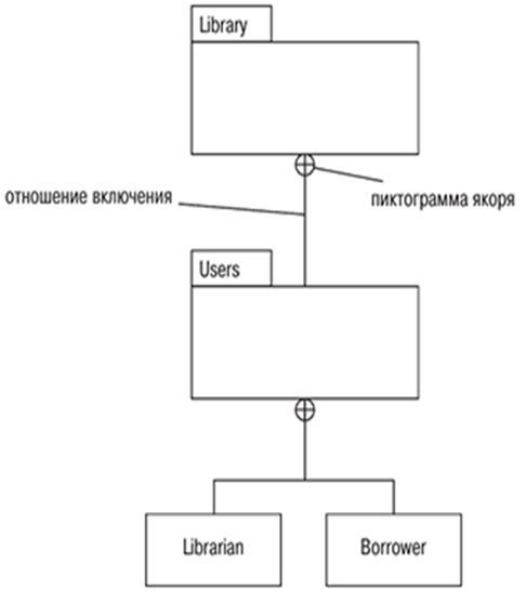
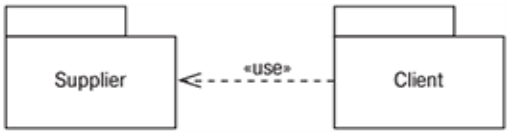
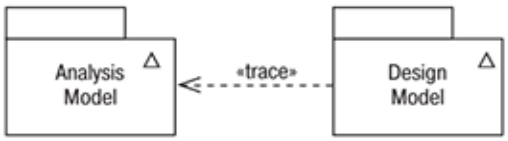


Рисунок 7.63 *Синтаксис сложного вложения*

Таблица 7.5

Отношение зависимости пакетов	Семантика
	Элемент клиентского пакета некоторым образом использует открытый элемент пакета-поставщика – клиент зависит от поставщика. Если зависимость пакета показана без стереотипа, необходимо предполагать зависимость «use».
	Открытые элементы пространства имен поставщика добавляются как открытые элементы в пространство имен клиента. Элементы клиента могут организовывать доступ ко всем открытым элементам поставщика через неполные имена.
	Открытые элементы пространства имен поставщика добавляются как закрытые элементы в пространство имен клиента Элементы клиента могут организовывать доступ ко всем открытым элементам поставщика с помощью неполных имен.
	«trace», как правило, представляет историческое развитие одного элемента в другую более развитую версию; обычно это отношение между моделями, а не элементами (межмодельное отношение).
	Открытые элементы пакета-поставщика объединяются с элементами клиентского пакета. Эта зависимость используется только при создании метамодели; в обычном ОО анализе и проектировании она не должна встречаться.

Транзитивность (`transitivity`) – термин, применяемый к отношениям. Он означает, что если существует отношение между сущностями `A` и `B` и отношение между сущностями `B` и `C`, то существует неявное отношение между сущностями `A` и `C`. Важно отметить, что зависимость «`import`» – транзитивная, а зависимость «`access`» – нет. Как было показано выше, это объясняется тем, что при наличии зависимости «`import`» между клиентом и поставщиком открытые элементы пакета поставщика становятся открытыми элементами клиента. Эти импортированные открытые элементы доступны вне клиентского пакета. С другой стороны, когда между клиентом и поставщиком установлена зависимость «`access`», открытые элементы пакета поставщика становятся закрытыми элементами клиента. Эти закрытые элементы не доступны вне клиентского пакета.

Рисунок 7.64 *«access» -нетранзитивная зависимость*

Отсутствие транзитивности в «access» означает следующее:

- открытые элементы пакета C становятся закрытыми элементами пакета B;
- открытые элементы пакета B становятся закрытыми элементами пакета A;
- следовательно, элементы пакета A не имеют доступа к элементам пакета C.

7.4 UML Проектирование

Проектирование – основная деятельность моделирования последней части фазы Уточнение и первой половины фазы Построение. Основное внимание первых итераций направлено на определение требований и анализ. По мере завершения анализа фокус моделирования перемещается на проектирование. В значительной степени анализ и проектирование могут проводиться параллельно. Однако, как мы увидим позже, важно четко различать артефакты этих двух рабочих потоков – аналитическую модель и проектную модель.

Основной целью анализа было построение логической модели системы, отражающей функциональность, которую должна предоставлять система для удовлетворения требований пользователя. *Цель проектирования* – определить в полном объеме, как будет реализовываться эта функциональность. Одним из путей решения этой задачи является одновременное изучение предметной области и области решения. Требования поступают из предметной области, и анализ можно рассматривать как ее исследование с точки зрения заказчиков системы. Проектирование предполагает объединение технических решений (библиотек классов, механизмов сохранения объектов и т. д.) для создания модели системы (проектной модели), которая может быть реализована в действительности.

При проектировании принимаются решения по стратегическим вопросам, таким как сохранение и распределение объектов, в соответствии с которыми и создается проектная модель. Руководитель и архитектор проекта должны также разработать политики рассмотрения любых тактических вопросов проектирования.

Проектную модель можно рассматривать как уточнение аналитической модели с добавлением деталей и конкретных технических решений. Проектная модель содержит все то же самое, что и аналитическая, но все артефакты в ней проработаны более основательно и должны включать детали реализации. Например, аналитический класс может быть не более чем эскизом с парой атрибутов и только ключевыми операциями. Однако проектный класс должен быть совершенно точно определён – все атрибуты и операции (включая возвращаемые типы и списки параметров) должны быть полностью описаны.

Проектные модели образуются:

- *проектными подсистемами;*
- *проектными классами;*
- *интерфейсами;*
- *реализациями прецедентов – проектными;*

- *диаграммой развертывания.*

При проектировании также создается диаграмма развертывания в первом приближении, которая показывает распределение программной системы на физических вычислительных узлах. Бесспорно, эта диаграмма важна и имеет стратегическое значение.

В идеальном мире для системы создавалась бы единственная модель, и средство моделирования могло бы предоставлять на выбор или аналитическое, или проектное представление. Однако это требование сложнее, чем кажется на первый взгляд. Ни одно из присутствующих в настоящее время на рынке инструментальных средств UML моделирования не может удовлетворительно справиться с задачей создания аналитического и проектного представлений на основании одной базовой модели.

Таблица 7.6

	Стратегия	Результаты
1	Берется аналитическая модель и дополняется до проектной модели.	Имеется единственная проектная модель, но утеряно аналитическое представление.
2	Берется аналитическая модель, дополняется до проектной модели, а с помощью инструмента моделирования восстанавливается «аналитическое представление».	Имеется единственная проектная модель, но восстановленное инструментом моделирования аналитическое представление может быть неприемлемым.
3	Аналитическая модель замораживается в некоторой точке фазы Уточнение. До проектной модели дополняется копия аналитической модели.	Имеются две модели, но они не синхронизированы.
4	Поддерживаются две отдельные модели – аналитическая и проектная.	Имеются две модели – они синхронизованы, но их обслуживание очень трудоемко.

Идеальной стратегии нет, все зависит от проекта. Однако необходимо задать себе фундаментальный вопрос: нужно ли сохранять аналитическое представление системы? Аналитические представления обеспечивают «общую картину» системы. В них может быть лишь от 1 до 10% классов подробного проектного представления, поэтому они более понятны. Их роль неоценима для следующих действий:

- введения в проект новых людей;
- понимания системы спустя месяцы или годы после ее поставки;
- понимания того, как система выполняет требования пользователей;
- обеспечения прослеживаемости требований;
- планирования обслуживания и улучшения;
- понимания логической архитектуры системы;
- привлечения внешних ресурсов к разработке системы.

Если система небольшая (скажем, меньше 200 проектных классов), тогда непосредственно проектная модель достаточно мала и понятна, поэтому в отдельной аналитической модели, возможно, и нет необходимости. Также, если система не имеет стратегического значения или недолговечна, разделение аналитической и проектной моделей – излишнее требование. В этом случае необходимо выбирать между стратегиями 1 и 2, и решающим фактором будут возможности используемого инструментального средства UML моделирования. Некоторые инструменты моделирования сохраняют одну базовую модель и обеспечивают возможность

применения фильтров и сокрытия информации для восстановления «аналитического» представления из проектной модели. Это разумный компромисс для многих систем среднего размера, но для очень больших систем этого, скорее всего, недостаточно. И наконец, стоит помнить о том, что реальный срок службы многих систем существенно превышает предполагаемый!

7.4.1 Детализация рабочего потока проектирования

Рабочий поток UP для проектирования представлен на рис. 7.65. Основные его участники: архитектор, разработчик прецедентов и разработчик компонентов. В большинстве ОО проектов роль архитектора выполняют один или несколько специалистов, но часто они же потом являются разработчиками прецедентов и компонентов.

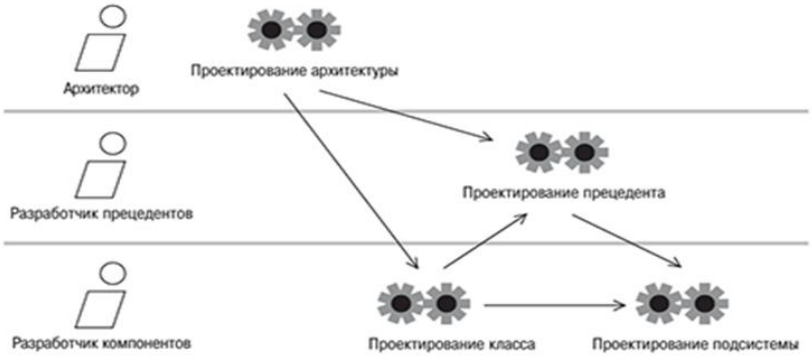


Рисунок 7.65 *Рабочий поток проектирования*

Одна из целей UP состоит в том, чтобы определенные члены команды отвечали за отдельные части системы, начиная с анализа и заканчивая реализацией. Таким образом, специалист или команда, ответственные за создание конкретной части ОО аналитической модели, будут дополнять ее до проектной модели и, возможно с помощью программистов, превращать ее в код. При таком подходе (в этом его основное преимущество) нет «перекладывания ответственности» друг на друга между аналитиками, проектировщиками и программистами, что распространено в ОО проектах**.**

7.4.2 Деятельность UP: проектирование архитектуры

В UP начало всему процессу проектирования дает деятельность под названием *Проектирование архитектуры* (`architectural design`). Ее осуществляют один или более архитекторов. Детали этой деятельности представлены на рис. 7.66. Как видно из рисунка, результатом Проектирования архитектуры является множество артефактов (затушеванные артефакты указывают на изменения, внесенные в оригинальный рисунок). Самое главное, необходимо понять, что проектирование архитектуры заключается в предварительном описании артефактов, важных с архитектурной точки зрения, с целью создания общего плана архитектуры системы. Обозначенные в общих чертах артефакты являются входными данными для более детального проектирования, в процессе которого происходит их конкретизация.

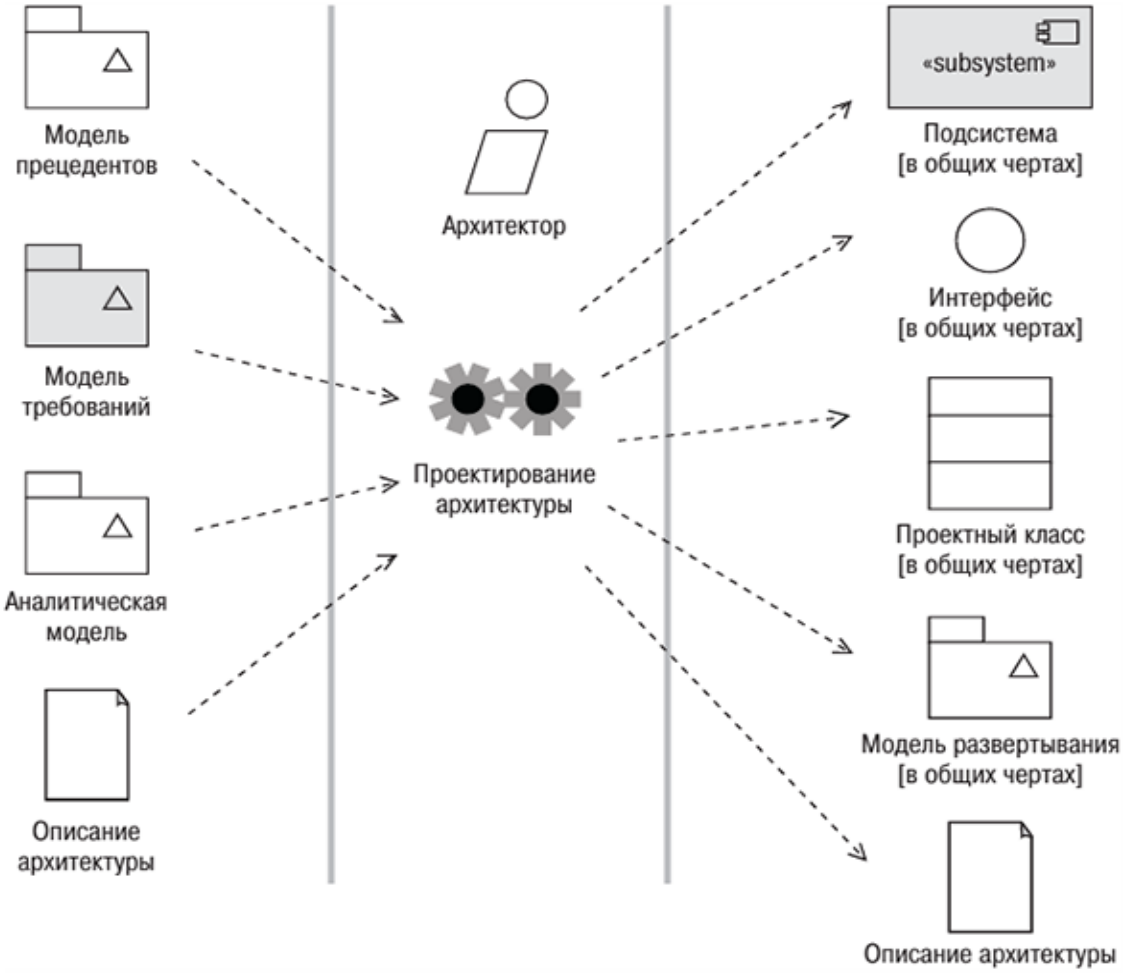


Рисунок 7.66

Обычно Проектирование архитектуры не выделяется в отдельный шаг. Не следует забывать, что UP – итеративный процесс. Таким образом, проработка деталей архитектуры системы ведется на всем протяжении завершающих этапов Уточнения и в начале Построения.

В рабочем потоке проектирования определяется, как будет реализовываться функциональность, описанная в аналитической модели. Мы узнали следующее:

- Проектирование – основная деятельность при моделировании в последней части фазы Уточнение и первой части фазы Построение.
- Анализ и проектирование в некоторой степени могут происходить параллельно.
- Одна команда должна провести артефакт от анализа до проектирования.
- ОО проектировщики основное внимание должны уделять главнейшим вопросам проектирования, таким как архитектуры распределенных компонентов – политики и стандарты должны вводиться для решения тактически важных вопросов проектирования.

Проектная модель включает:

- *проектные подсистемы;*
- *проектные классы;*
- *интерфейсы;*

- *реализации прецедентов – проектные;*
- *диаграмму развертывания (в первом приближении).*

Отношения отображения существуют между:

- *проектной и аналитической моделями;*
- *одной или более проектными подсистемами и пакетом анализа.*

Следует поддерживать две отдельные модели, аналитическую и проектную, если система:

- *большая;*
- *сложная;*
- *стратегически важная;*
- *подвержена частым изменениям;*
- *предположительно с большим сроком службы;*
- *для ее разработки привлекаются внешние ресурсы.*

7.4.3 Проектные классы

Проектные классы – строительные блоки проектной модели. Для ОО проектировщика жизненно важно понимать, как моделировать эти классы эффективно.

Проектирование класса

В детализации рабочего потока проектирования в UP (рис. 7.65) после проектирования архитектуры (`Architectural design`) следуют деятельности проектирование класса (`Design a class`) и Проектирование прецедента (`Design a use case`). Они являются параллельными и итеративными. на рис. 7.67.

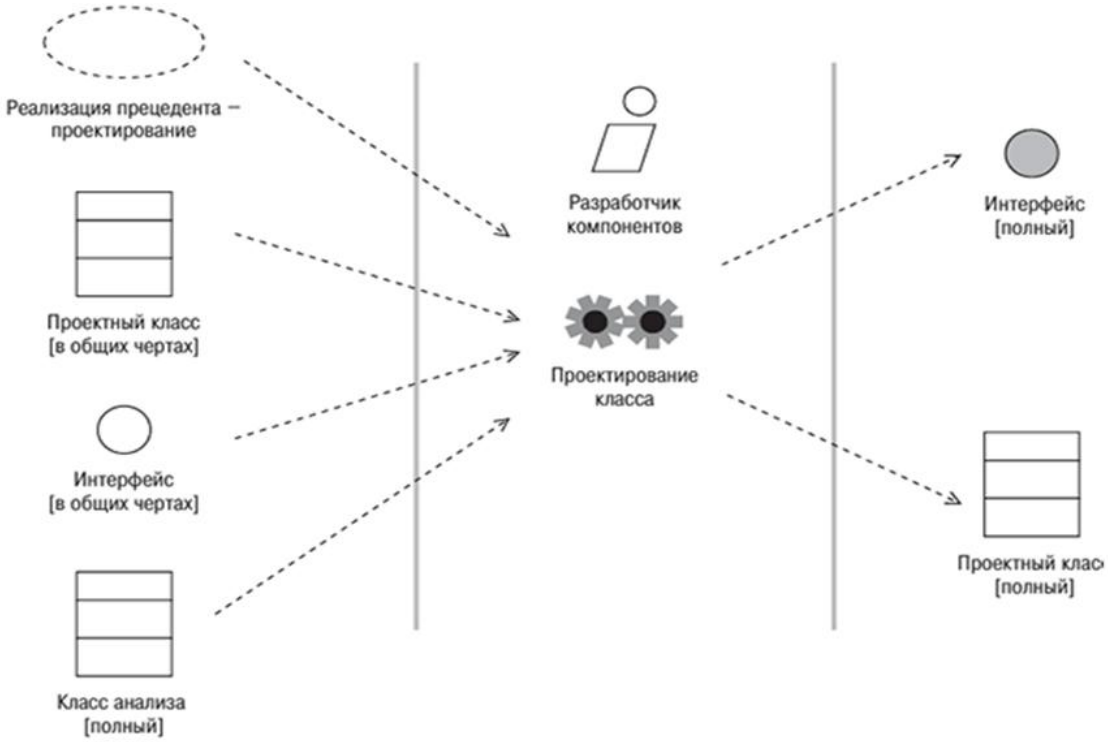


Рисунок 7.67 *Деятельность Проектирование класс*

Мы расширили ее и показали Интерфейс [полный] как явный результат Проектирования класса. Этот артефакт затушёван, чтобы показать, что он был изменен. В оригинальном описании деятельности он был неявным результатом.

7.4.4 Создание входного артефакта.

Стоит более подробно рассмотреть артефакт Проектный класс [в общих чертах]. С точки зрения деятельности кажется, что здесь присутствуют два отдельных самостоятельных артефакта, Проектный класс [в общих чертах] и Проектный класс [полный]. Однако это не так. Они просто представляют один и тот же артефакт (проектный класс) на разных этапах развития. Если взять набор артефактов UP проекта в конце фазы Уточнение или в начале Построения, то не найдется артефактов под названием Проектный класс [в общих чертах] или Проектный класс [полный]. Там будут только проектные классы, находящиеся на разных этапах своего развития. С точки зрения UP «полный» проектный класс – это класс, достаточно детализированный, чтобы служить базой для создания исходного кода. Это основное положение, о котором начинающие разработчики моделей часто забывают. Проектные классы моделируются с той степенью детализации, которая позволяет создать код, полученный на их основе. Поэтому модели проектных классов редко бывают исчерпывающими. Необходимый уровень детализации определяется проектом. Если предполагается генерировать код прямо из модели, то проектные классы необходимо моделировать очень подробно. С другой стороны, если программисты не будут использовать их в качестве рабочего прототипа, модели проектных классов могут быть менее детальными. **Проектные классы, достаточно подробные для любого проекта**.

Проектные классы – это классы, описания которых настолько полные, что они могут быть реализованы. При анализе источником классов является предметная область. Это набор требований, описывающий задачу, которую необходимо решить. Как мы видели, прецеденты, описания требований, глоссарии и любая другая относящаяся к делу информация могут использоваться как источник классов анализа.

У проектных классов два источника:

- Предметная область посредством уточнения классов анализа; это уточнение включает добавление деталей реализации. В ходе этого процесса часто обнаруживается, что высокоабстрактный класс анализа необходимо разбить на два или более детализированных проектных класса. Реализацию класса анализа описывает отношение «`trace`», устанавливаемое между ним и одним или более проектными классами.
- Область решения – это царство библиотек утилитных классов и многократно используемых компонентов, таких как `Time`, `Date`, `String`, коллекции и т. д. Здесь находится промежуточное программное обеспечение (`middleware`), такое как коммуникационное ПО, базы данных (и реляционные, и объектные) и компонентные инфраструктуры, например `.NET`, `CORBA` или `Enterprise JavaBeans`, а также средства для построения GUI. Эта область предоставляет технические инструментальные средства для реализации системы.

При переходе к проектированию все операции и атрибуты класса должны быть полностью описаны, поэтому нередко он становится слишком большими. Если это происходит, необходимо разбить его на два или более меньших классов. Помните, что всегда надо стремиться проектировать небольшие классы, являющиеся самодостаточными, связными элементами, которые хорошо справляются с одной двумя функциями. Любой ценой необходимо избегать больших классов, напоминающих «швейцарский армейский нож», которые делают все. Выбранный метод реализации определяет требуемую степень полноты описаний проектных классов. Если планируется передать модель проектного класса программистам в качестве руководства для написания кода, проектные классы должны быть полными лишь настолько, чтобы обеспечить им возможность эффективного выполнения задания. Всё зависит от квалификации программистов и того, насколько хорошо они понимают предметную область и область решения. Это выясняется для каждого конкретного проекта индивидуально. Однако если предполагается использовать проектные классы для генерации кода с помощью соответствующим образом оснащенного инструментального средства моделирования, их описания должны быть полными во всех отношениях. Генератор кода, в отличие от программиста, не может заполнять пробелы. С помощью классов анализа делается попытка зафиксировать требуемое поведение системы без рассмотрения его возможной реализации. В проектных классах необходимо точно определить, как каждый класс будет осуществлять свои обязанности. Для этого нужно сделать следующее:

- закончить набор атрибутов и полностью описать их, включая имя, тип, видимость и (необязательно) применяемое по умолчанию значение;
- закончить набор операций и полностью описать их, включая имя, список параметров и возвращаемый тип.

Этот процесс уточнения проиллюстрирован на рис. 7.68. Как было показано ранее, операция в классе анализа – это высокоуровневое логическое описание части функциональности, предлагаемой классом. В соответствующих проектных классах каждая операция класса анализа уточняется и превращается в одну или более детализированных и полностью описанных операций, которые могут быть реализованы как исходный код. Следовательно, одна высокоуровневая операция этапа анализа на самом деле может распадаться на одну или более проектных операций, которые можно реализовать. Эти детализированные операции уровня проектирования иногда называют методами. Для иллюстрации рассмотрим следующий пример. При анализе системы регистрации авиапассажиров можно определить высокоуровневую операцию `checkIn()` (зарегистрировать). Однако если вам когда-либо приходилось стоять в очереди на регистрацию на рейс, то вы знаете, что это довольно сложный бизнес процесс, включающий сбор и проверку достоверности определенной информации о пассажире.

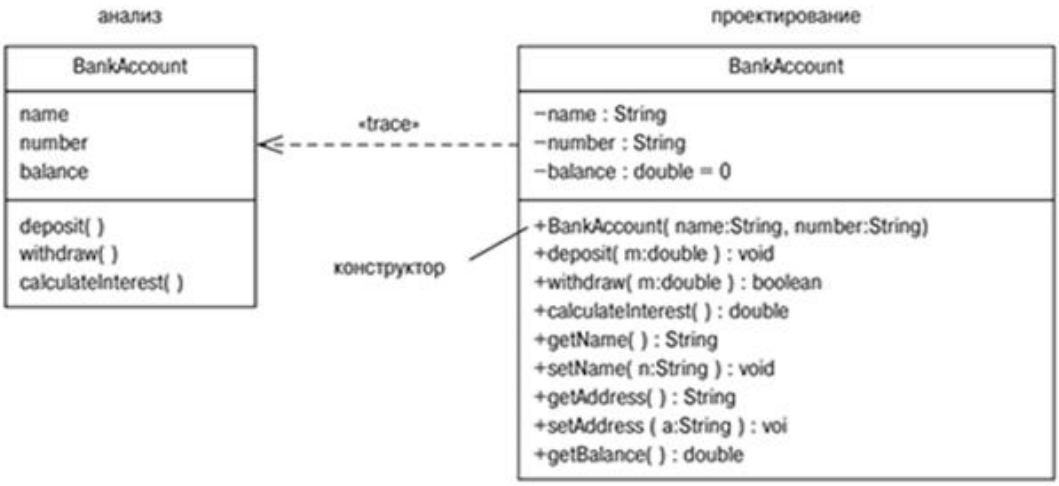


Рисунок 7.68 *Детализация на этапе проектирования*

7.4.5 Правильно сформированные проектные классы

Проектная модель будет передана программистам для фактического создания исходного кода. Код также может генерироваться непосредственно из самой модели, если это поддерживается инструментом моделирования. Следовательно, проектные классы должны быть достаточно подробно описаны. В процессе описания определяется, является ли класс «правильно сформированным» или нет.

При создании важно всегда рассматривать проектный класс с точки зрения его потенциальных клиентов. Каким они увидят этот класс – не слишком ли он сложен? Может быть что-то упущено? Как тесно он взаимосвязан с другими классами? Соответствует ли его имя выполняемым функциям? Все это важно и может быть сведено в следующие четыре основные характеристики правильно сформированного проектного класса:

- полный и достаточный;
- простой;
- обладает высокой внутренней связностью;

обладает низкой связанностью с другими классами.

7.4.6 Наследование

При проектировании наследование играет намного более важную роль, чем при анализе. В анализе наследование использовалось, только если между классами анализа имело место четкое и явно выраженное отношение «*является*». Однако при проектировании наследование может применяться в тактических целях для повторного использования кода. Это совсем иная стратегия, поскольку наследование фактически используется для упрощения реализации дочернего класса, а не для выражения бизнес-отношения между родителем и потомком.

Сравнение агрегации и наследования

Наследование – очень мощный метод. Оно является ключевым механизмом формирования полиморфизма в строго типизированных языках программирования, таких как `Java`, `C#` и `C++`. Однако неопытные ОО проектировщики и программисты

часто используют его неправильно. Необходимо осознавать, что наследование имеет определенные нежелательные характеристики. Это самая строгая из возможных форма связанности двух или более классов.

- В иерархии классов инкапсуляция низкая. Изменения базового класса передаются вниз по иерархии и приводят к изменениям подклассов. Это явление называют проблемой «хрупкости базового класса», когда изменения базового класса имеют огромное влияние на другие классы системы.
- Это очень жесткий тип отношения. Во всех широко используемых ОО языках программирования отношения наследования постоянны во время выполнения. Создавая и уничтожая отношения во время выполнения, можно изменять иерархии агрегации и композиции, но иерархии наследования остаются неизменными. Это делает наследование самым жестким типом отношений между классами.

Типичный пример решения задачи моделирования ролей в организации, выполненного неопытным разработчиком. На первый взгляд все вполне приемлемо, однако здесь есть проблемы. Рассмотрим предложенный вариант. Объект `john` типа `Programmer` (программист) необходимо повысить до типа `Manager` (менеджер). Вам должно быть понятно, что изменить класс Джона (`john`) во время выполнения нельзя. Поэтому единственный способ повысить Джона – создать новый объект `Manager` (названный `john:Manager`), скопировать в него из объекта `john:Programmer` все существующие данные и затем удалить `john:Programmer` для сохранения целостности приложения. Конечно, это очень сложно и совершенно не соответствует тому, как все происходит на самом деле.

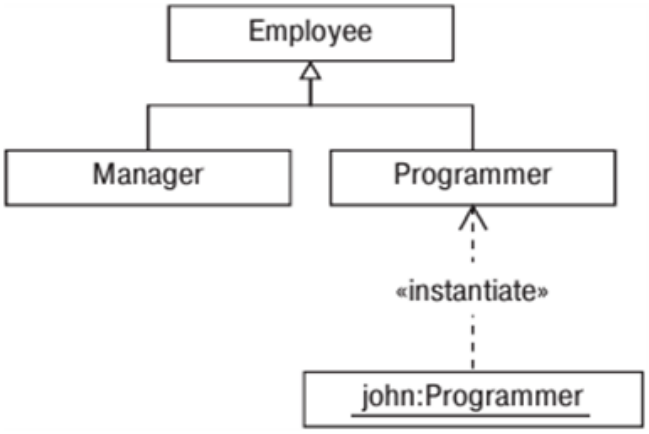


Рисунок 7.69 *Неверная модель*

В сущности, в модели на рис. 7.69 *допущена фундаментальная семантическая ошибка*. Служащий (`Employee`) – это именно должность (`Job`) или это указывает на то, что служащий занимает некоторую должность? Ответ на этот вопрос приводит к решению проблемы (см. рис. 7.70).

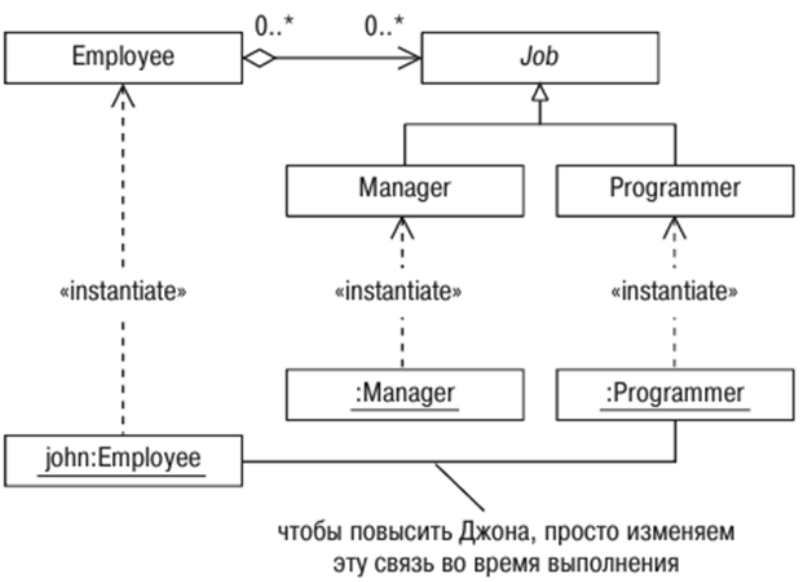


Рисунок 7.70 *Верная модель*

Множественное наследование

Иногда возникает потребность наследования от нескольких родителей. Такое наследование называют *множественным*. Его поддерживают не все ОО языки программирования, например в `Java` и `C#` допускается только единичное наследование. На практике отсутствие поддержки множественного наследования не является проблемой, поскольку его всегда можно заменить единичным наследованием и делегированием. Даже несмотря на то, что иногда множественное наследование предлагает более элегантное решение задачи проектирования, оно может использоваться, только если целевой язык реализации его поддерживает.

О множественном наследовании важно знать следующее.

- Все участвующие родительские классы должны быть семантически не связанными. В случае наличия какого-либо совмещения в семантике базовых классов между ними возможны непредвиденные взаимодействия. Это может привести к необычному поведению подкласса. Мы говорим, что базовые классы должны быть ортогональными (расположенными под прямым углом друг относительно друга).
- Между подклассом и всеми его суперклассами должны действовать принцип *замещаемости* и принцип «*является разновидностью*».
- У суперклассов не должно быть общих родителей. В противном случаев иерархии наследования образуется цикл и возникает множество путей наследования одних и тех же возможностей от более абстрактных классов. У языков программирования, поддерживающих множественное наследование (таких как *C++*), есть специальные, особые для каждого языка способы разрешения циклов в иерархии наследования.

Недостатки использования наследования

- это самая строгая из возможных форм связанности между двумя классами;
- в рамках иерархии наследования инкапсуляция слаба, что приводит к проблеме «хрупкости базового класса» – изменения базового класса передаются вниз по иерархии;
- является очень негибкой формой отношений в большинстве языков программирования – отношение устанавливается во время компиляции и остается неизменным во время выполнения.
- Подклассы всегда должны представлять «разновидность», а не «исполняемую роль» – для представления «исполняемой роли» должна использоваться агрегация.
- Множественное наследование позволяет классам иметь более одного родителя.
- Из всех широко используемых ОО языков программирования только `C++` поддерживает множественное наследование.

Рекомендации к проектированию

- все родительские классы при множественном наследовании должны быть семантически не связанными;
- между классом и всеми его родителями должно быть установлено отношение «является разновидностью»;
- к классу и его родителям должен применяться принцип замещаемости;
- у самих родительских классов не должно быть общих родителей;
- используйте смешанные классы – простые классы, разработанные для смешивания с другими классами при множественном наследовании; это надежное и мощное средство.

Сравнение реализации интерфейса и наследования

Реализация интерфейса – «реализует определяемый им контракт».

Теперь стоит обсудить, в чем разница между реализацией интерфейса и наследованием. Семантика реализации интерфейса – «*реализует определённый контракт*», а семантика наследования – «*является*». Принцип замещаемости применим как к наследованию, так и к реализации интерфейса. Таким образом, оба типа отношений могут формировать полиморфизм.

Семантика наследования – «является».

Чтобы проиллюстрировать разницу между реализацией интерфейса и наследованием, мы предлагаем альтернативное решение для системы управления библиотекой на основании наследования (рис. 7.71). Оно кажется вполне приемлемым и в некотором отношении даже более простым, но здесь есть некоторые проблемы. Прежде всего обсудим, почему эта основанная на наследовании модель не совсем подходит. Здесь очень четко определяется, что объекты `Book` и `CD` имеют тип `BorrowableItem`. Но разве этой способности `Book` и `CD` быть выданными на время достаточно для описания их типа? Наверное, ее можно было бы рассматривать как один из аспектов их поведения,

который оказался бы общим в контексте библиотечной системы. Семантически правильнее рассматривать `BorrowableItem` как отдельную роль, которую `Book` и `CD` играют в `Library`, а не как общий надтип.

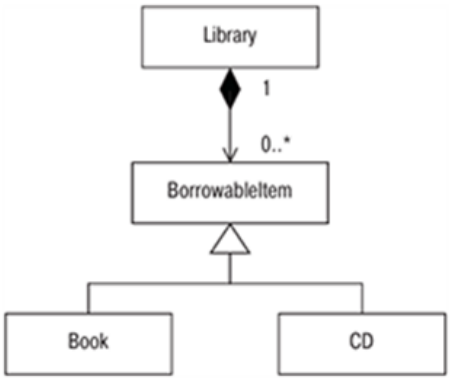


Рисунок 7.71 *Модель, основанная на наследовании, имеет недостаток*

Чтобы конкретизировать недостаток модели, изображенной на рис. 7.71, добавим в систему `Library` класс `Journal` (журнал). `Journal` – это периодическое издание, такое как «`Nature`» (Природа), не выдаваемое на время. В результате получается основанная на наследовании модель, представленная на рис. 7.72. Обратите внимание, что теперь `Library` должен обслуживать два списка объектов – те, которые могут, и те, которые не могут выдаваться на время. Такое решение работоспособно, но не очень изящно, поскольку в нем смешиваются два очень разных понятия системы `Library`:

- хранящиеся объекты;
- объекты, выдаваемые на время.

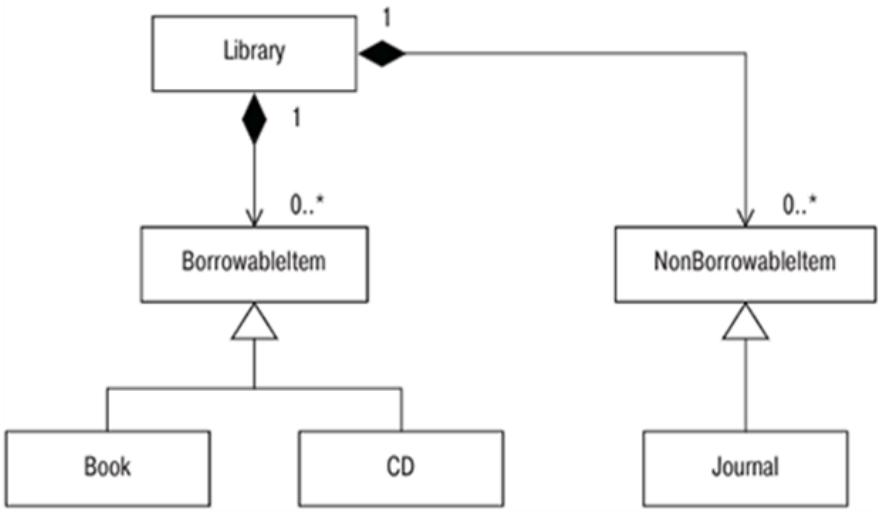


Рисунок 7.72 *Модель `Library` с двумя списками объектов*

Эту модель можно несколько усовершенствовать, введя дополнительный уровень иерархии наследования, как показано на рис. 7.73. Использование класса `LibraryItem` (библиотечная позиция) избавляет от одного из отношений композиции. Такое решение в принципе подходит, поскольку в нем используется только

наследование. Протокол «*выдаваемый на время*» вынесен в отдельный уровень иерархии наследования. Это обычное решение проблемы такого рода. Модель, использующая и интерфейсы, и наследование, обеспечивает более элегантное элегантное решение (рис. 7.74). Основанное на интерфейсах решение имеет следующие преимущества:

- каждая позиция в `Library` является `LibraryItem`;
- понятие «*возможность выдачи на время*» вынесена в отдельный интерфейс, `Borrowable`, который может применяться к классам `LibraryItem` в случае необходимости;
- сокращается число классов – пять классов и один интерфейс в противоположность семи классам в другом решении;
- меньше отношений композиции – одно в противоположность двум в другом решении;
- более простая, состоящая всего из двух уровней, иерархия наследования;
- меньше отношений наследования – три в противоположность пяти в другом решении.

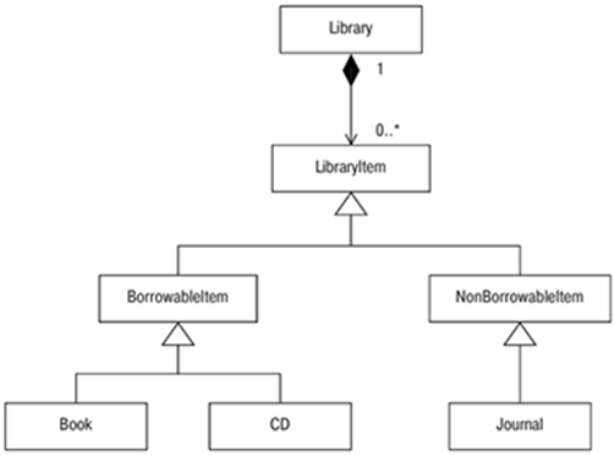


Рисунок 7.73 *Усовершенствованная модель* `Library`

В общем, основанное на интерфейсе решение проще и обладает лучшей семантикой. Такие характеристики, как `catalogNumber` (номер каталога), которые есть у всех `LibraryItem`, были вынесены в базовый класс, чтобы обеспечить возможность их наследования. А протокол «возможность выдачи на время» определен отдельно в интерфейсе `Borrowable`.

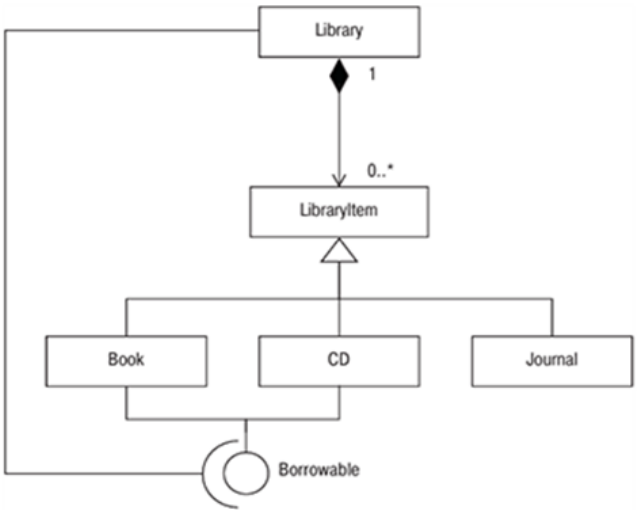


Рисунок 7.74 *Модель* `Library`*, использующая и интерфейсы, и наследование*

Чтобы проиллюстрировать гибкость интерфейсов, давайте пойдем в этом примере на шаг вперед. Предположим, что необходимо экспортировать данные классов `Book` и `Journal` (но не `CD`) в `XML` файлы. Прикладные драйверы в этом случае должны обеспечивать возможность обмена информацией с другими библиотеками и представления каталога печатных материалов в Веб. Было спроектировано следующее решение:

- для осуществления экспорта в `XML` введен класс `XMLExporter`;
- введен интерфейс `XMLExportable`, определяющий протокол для работы с `XMLExporter`, который должен быть у каждой экспортируемой позиции. Имеются следующие нефункциональные требования:
- языком реализации должен быть `Java`;
- для обработки `XML` должна использоваться библиотека `JDOM`.

(`JDOM` – простая, но мощная библиотека `Java` для работы с `XML` документами). Протокол `XMLExportable` – это всего лишь одна операция `getElement()`, возвращающая представление экспортируемого элемента в виде класса `Element`, описанного в библиотеке `JDOM`. Класс `XMLExporter` использует `JDOM` для записи объектов `Element` в `XML` файл. Полностью решение представлено на рис. 7.75.

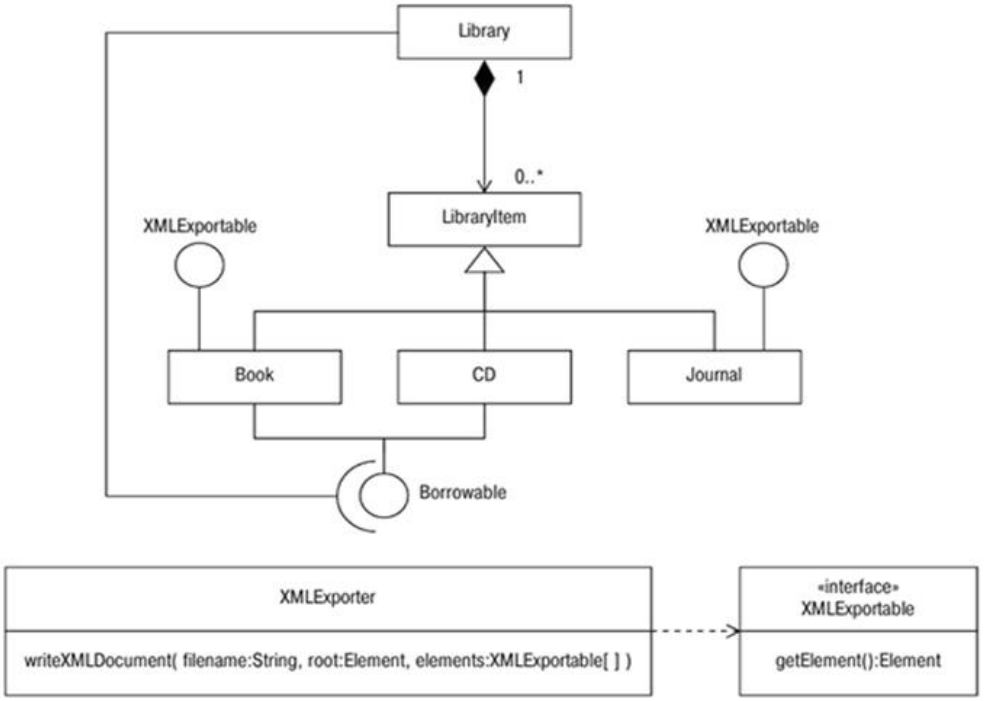


Рисунок 7.75 *Усовершенствованная модель* `Library` *с возможностью экспорта данных в* `XML` *файлы*

Интерфейсы используются для описания общих протоколов классов, которые обычно не связываются через наследование. В данном решении удалось разделить вопросы хранения (отношение композиции), выдачи на время (`Borrowable`) и экспортируемости (`XMLExportable`). Интерфейсы использовались для определения общих протоколов классов, которые не должны быть связаны через наследование.

7.4.7 Порты

Порт группирует семантически связанный набор предоставляемых и требуемых интерфейсов. Он указывает на конкретную точку взаимодействия классификатора и его окружения. Пример на рис. 7.76 иллюстрирует нотацию порта. Здесь показан класс `Book`, имеющий порт `presentation` (представление). Этот порт состоит из требуемого интерфейса `DisplayMedium` (устройство отображения) и предоставляемого интерфейса `Display` (отображение). Имя порта является необязательным. На рисунке показано два варианта нотации порта: слева – обычная, а справа – более краткий альтернативный вариант. Однако эта альтернатива применима, только если у порта один тип предоставляемого интерфейса (у него по-прежнему может быть нуль или более требуемых интерфейсов). Имя типа указывается после имени порта, как показано на рисунке. Порты являются очень удобным способом структурирования предоставляемых и требуемых интерфейсов классификатора. Их также можно использовать для упрощения диаграммы. Например, на рис. 7.77 показан класс `Viewer` (средство просмотра), соединяющийся с портом `presentation` класса `Book`. Чтобы обеспечить возможность соединения портов, их предоставляемый и требуемый интерфейсы должны совпадать. Использование портов, очевидно, обеспечивает намного

более краткое представление, чем отображение всех предоставляемых и требуемых интерфейсов, но может и усложнять чтение диаграмм. У портов может быть видимость. Когда порт изображается перекрывающим границу классификатора, он является открытым. Это означает, что предоставляемый и требуемый интерфейсы открытые (`public`). Если прямоугольник порта находится внутри границы классификатора (как показано на рис. 7.78), видимость порта принимает два значения: или защищенный (`protected`) (по умолчанию), или закрытый (`private`). Фактическая видимость графически не отображается, но записывается в спецификации порта.

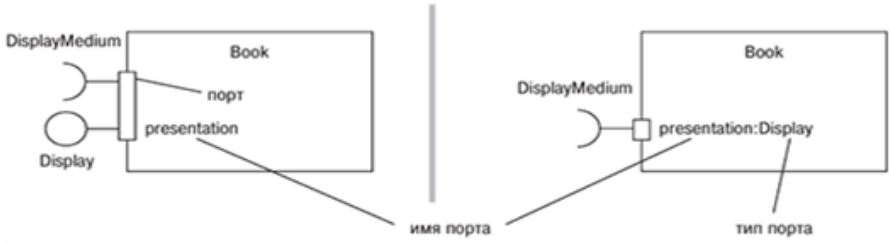


Рисунок 7.76 *Два варианта нотации порта*

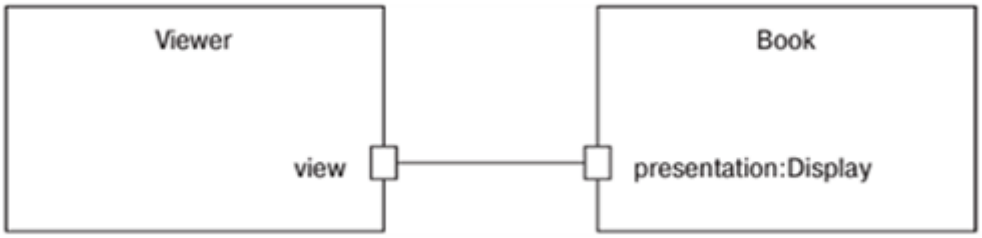


Рисунок 7.77 *Соединение портов*

Рисунок 7.78 *Защищенный порт*

7.4.8 Интерфейсы и компонентноориентированная разработка

Компонентноориентированная разработка заключается в построении программного обеспечения из подключаемых частей.

Интерфейсы – ключ к компонентноориентированной разработке (`componenttbased development, CBD`). Она заключается в построении программного обеспечения из подключаемых частей. Если требуется создать гибкое, ориентированное на компоненты программное обеспечение, для которого по желанию можно подключать новые реализации, при проектировании должны использоваться интерфейсы. Поскольку интерфейсы определяют только контракт, они обеспечивают возможность существования любого количества конкретных реализаций, подчиняющихся этому контракту.

Спецификация UML 2.0 [UML2S] гласит: «*Компонент представляет модульную часть системы, которая инкапсулирует ее содержимое, и реализация компонента замещаема в рамках его окружения*». Компонент как черный ящик, внешнее поведение которого полностью определяется его предоставляемыми и требуемыми интерфейсами. Поэтому один компонент может быть заменен другим, поддерживающим тот же протокол.

Компонент – модульная и замещаемая часть системы, инкапсулирующая ее содержимое. Компоненты могут иметь атрибуты и операции и участвовать в отношениях ассоциации и обобщения. Компоненты – это структурированные классификаторы. У них может быть внутренняя структура, включающая части и соединители.

Структурированный классификатор – это классификатор, имеющий внутреннюю структуру. Структурированный классификатор (structured classifier) – это просто классификатор (такой как класс), имеющий внутреннюю структуру. Эта структура моделируется как части, объединенные с помощью соединителей. Взаимодействие структурированного классификатора с его окружением моделируется его интерфейсами и портами.

Часть – это роль, которую могут выполнять один или более экземпляров классификатора в контексте структурированного классификатора. Каждая часть может иметь:

- *имя роли* – описательное имя роли, исполняемой экземплярами в контексте структурированного классификатора;
- *тип* – только экземпляры этого типа (или подтипа этого типа) могут играть эту роль;
- *кратность* – число экземпляров, которые могут играть роль в любой конкретный момент времени.

Соединители – это отношения между ролями (частями). Соединители и части существуют только в рамках контекста конкретного структурированного классификатора.

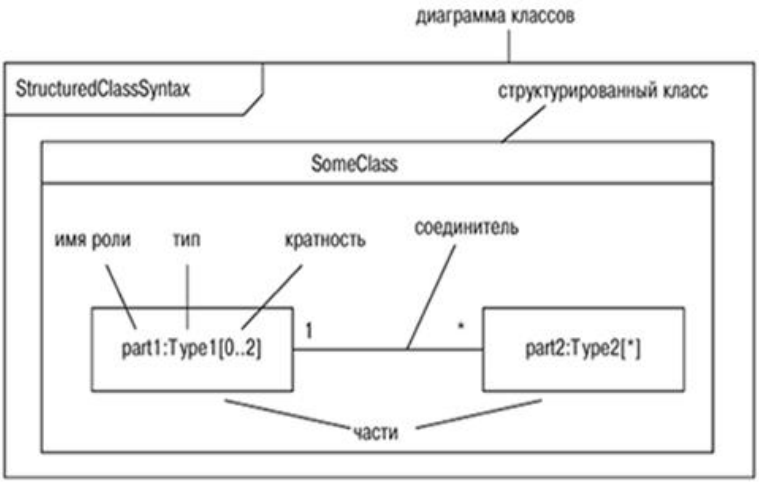


Рисунок 7.79 *Синтаксис структурированного классификатора*

Компоненты могут представлять что-то, экземпляр чего может быть создан во время выполнения, например EJB (Enterprise JavaBean). Или ими может быть представлена абсолютно логическая конструкция, такая как подсистема, экземпляры которой создаются только косвенно через создание экземпляров ее частей. Компонент может быть представлен одним или более артефактами. Артефакт представляет некоторую физическую сущность, например исходный файл. В частности, компонент EJB мог бы быть представлен файлом JAR (Java Archive – Java архив). На диаграмме компонентов могут быть показаны компоненты, зависимости между ними и то, как компонентам назначаются классификаторы. Компонент отображается в виде прямоугольника со стереотипом «component» (компонент) и/или пиктограммой компонента в верхнем правом углу, как на рис. 7.80. У компонентов могут быть предоставляемые и требуемые интерфейсы и порты. Компонент может иметь внутреннюю структуру. Части можно показать вложенными внутрь компонента (рис. 7.81) или находящимися снаружи и соединенными с ним отношением зависимости. Обе формы синтаксическии эквивалентны, хотя первая нотация нам кажется более наглядной.

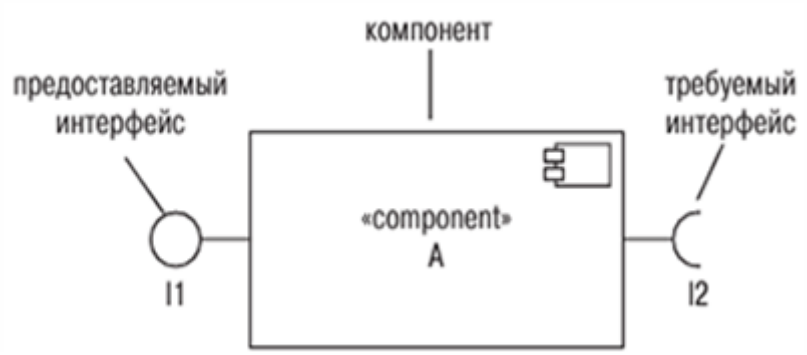


Рисунок 7.80 *Нотация компонента*

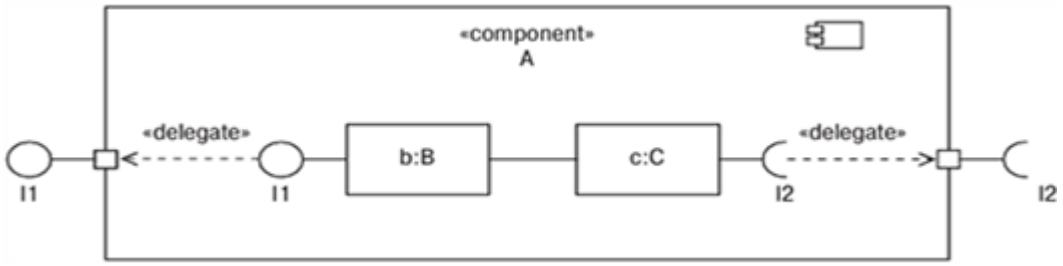


Рисунок 7.81 *Внутренняя структура компонента*

Если у компонента есть внутренняя структура, как правило, он будет делегировать обязанности, определенные его интерфейсами, своим внутренним частям. На рис. 7.81 компонент A предоставляет интерфейс I1 и требует интерфейс I2. Он инкапсулирует две части типа b и c. Он делегирует поведение, описанное его предоставляемым и требуемым интерфейсами b и c соответственно.

7.4.9 Моделирование параллелизма

Параллелизм – один из ключевых вопросов, рассматриваемых при проектировании.

Параллелизм означает параллельное выполнение частей системы. Это один из ключевых вопросов, рассматриваемых при проектировании. UML 2 обеспечивает хорошую поддержку параллелизма:

- активные классы;
- ветвления и объединения на диаграммах деятельности;
- различные представления временных диаграмм;
- ортогональные составные состояния на диаграммах состояний;

Активные классы

Параллелизм – каждый активный объект имеет собственный поток выполнения.

Основной принцип моделирования параллелизма – каждый поток управления или параллельный процесс моделируется как активный объект. Под последним понимают объект, инкапсулирующий собственный поток управления. Активные объекты являются экземплярами активных классов. Активные объекты и активные классы изображаются на диаграммах как обычные классы и объекты, но с двойной рамкой справа и слева, как показано на рис. 7.84. Параллелизм обычно имеет большое значение для встроенных систем, таких как программное обеспечение, управляющее мини фотолабораторией или банкоматом. Для изучения параллелизма рассмотрим очень простую встроенную систему – систему безопасности. Она отслеживает ряд датчиков для обнаружения пожара или проникновения в помещение взломщиков. При срабатывании датчика система включает сигнализацию. Модель прецедентов для системы безопасности показана на рис. 7.82. Описания прецедентов системы приведены на рис. 7.83. Прецедент `ActivateFireOnly` (активировать только пожарные датчики) не рассматривается, поскольку основное внимание в этом разделе направлено на аспекты параллелизма системы. Кроме того, эти прецеденты довольно абстрактны и отражают лишь суть того, что должна делать система сигнализации. Более подробно ее поведение будет рассмотрено позже. Теперь надо найти классы. Для встроенных систем превосходным источником классов могут быть аппаратные средства, на которых система выполняется. На практике наилучшей программной архитектурой обычно является та, которая максимально близка к архитектуре физического оборудования. В рассматриваемом случае сигнализация состоит из четырех компонентов: блока управления, сирены, набора пожарных датчиков и набора датчиков безопасности. Если заглянуть в блок управления, там можно найти плату контроллера для каждого из типов датчиков.

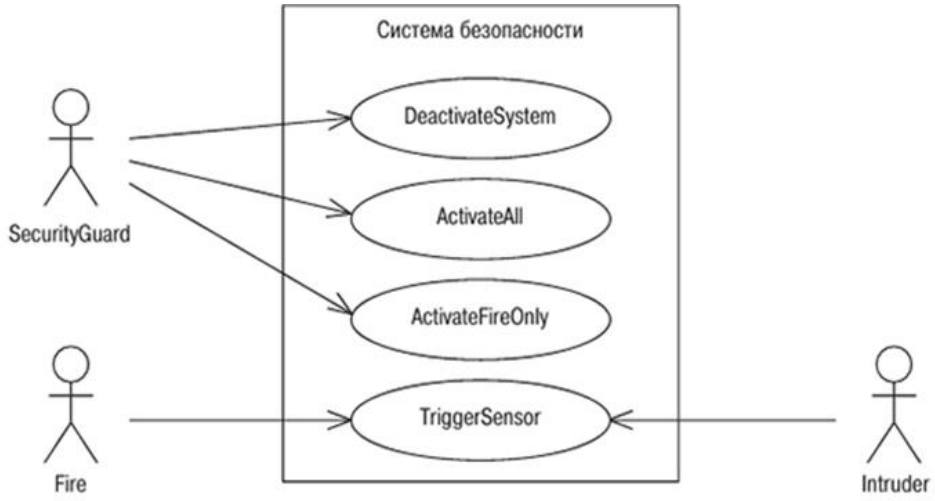


Рисунок 7.82 *Модель прецендентов для системы безопасности*

Прецедент: DeactivateSystem
ID: 1
Краткое описание: Деактивирует систему.
Главные актеры: SecurityGuard
Второстепенные актеры: Нет.
Предусловия: 1. У SecurityGuard есть ключ активации.
Основной поток: 1. SecurityGuard применяет ключ активации для выключения системы. 2. Система прекращает отслеживание датчиков безопасности и пожарных датчиков.
Постусловия: 1. Система безопасности деактивирована. 2. Система безопасности не отслеживает датчики.
Альтернативные потоки: Нет.

Прецедент: ActivateAll (АктивироватьВсе)
ID: 2
Краткое описание: Активирует систему.
Главные актеры: SecurityGuard
Второстепенные актеры: Нет.
Предусловия: 1. У SecurityGuard есть ключ активации.
Основной поток: 1. SecurityGuard применяет ключ активации для включения системы. 2. Система начинает отслеживать датчики безопасности и пожарные датчики. 3. Система воспроизводит звук сирены, демонстрируя готовность.
Постусловия: 1. Система безопасности активирована. 2. Система безопасности отслеживает датчики.
Альтернативные потоки: Нет.

Прецедент: TriggerSensor
ID: 3
Краткое описание: Срабатывает датчик.
Главные актеры: Fire Intruder
Второстепенные актеры: Нет.
Предусловия: 1. Система безопасности активирована.
Основной поток: 1. Если актер Fire инициирует пожарный датчик (FireSensor). 1.1. Сирена воспроизводит сигнал пожарной тревоги. 2. Если актер Intruder инициирует датчик безопасности (SecuritySensor). 2.1. Сирена воспроизводит сигнал тревоги.
Постусловия: 1. Звучит сигнал Сирены.
Альтернативные потоки: Нет.

Рисунок 7.83 *Описание прецедентов системы безопасности*

Исходя из прецедентов и информации о физическом оборудовании можно получить диаграмму классов для этой системы, представленную на рис. 7.84.

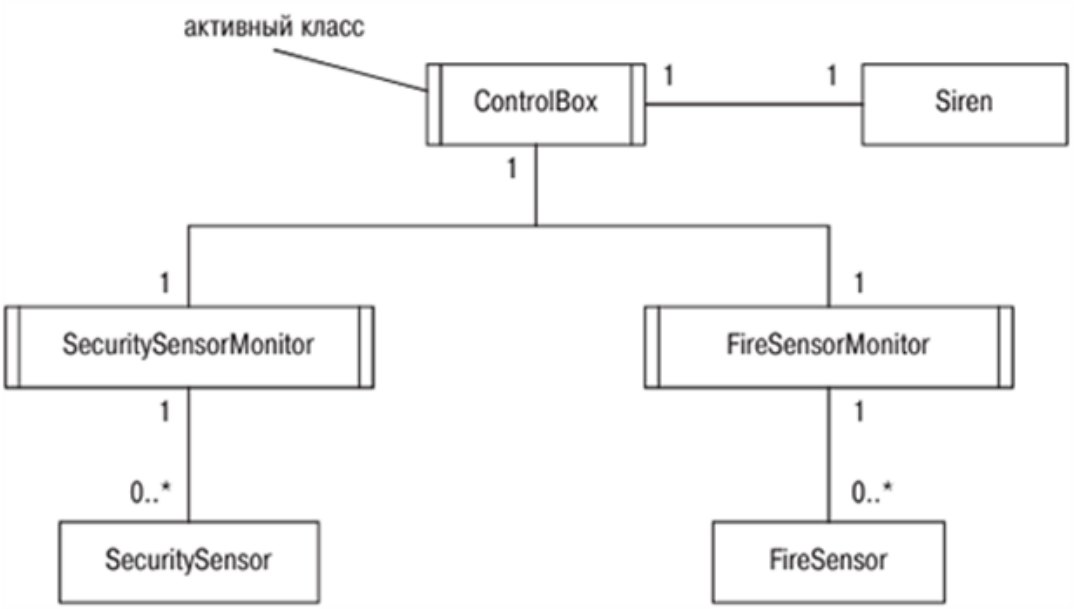


Рисунок 7.84 *Диаграмма классов системы безопасности*

Взаимодействия подсистем на диаграммах взаимодействий подсистем можно показать взаимодействия между частями системы. После создания физической архитектуры подсистем и интерфейсов может оказаться полезным смоделировать взаимодействия между подсистемами. Это обеспечивает очень удобное высокоуровневое представление того, как архитектура реализует прецеденты, без перехода к более низкоуровневым деталям взаимодействий отдельных объектов. Каждая подсистема рассматривается как черный ящик, который просто предоставляет и требует сервисы, описанные как предоставляемые и требуемые интерфейсы. О взаимодействиях объектов внутри подсистемы вообще не надо беспокоиться.

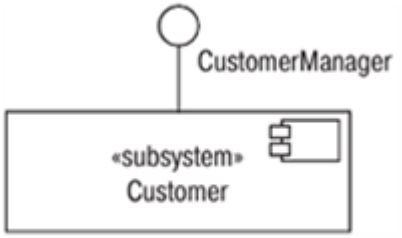


Рисунок 7.85 *Подсистема* `Customer`

На рис. 7.85 показана подсистема `Customer` с единственным интерфейсом `CustomerManager`. Рисунок 7.86 является частью диаграммы последовательностей, представляющей взаимодействие актера с данной подсистемой. Обратите внимание, как изображен интерфейс: в прямоугольнике, примыкающем снизу к пиктограмме подсистемы. Поскольку интерфейс является частью подключаемой подсистемы, он может иметь собственную линию жизни, и сообщения могут направляться непосредственно к этой линии жизни.

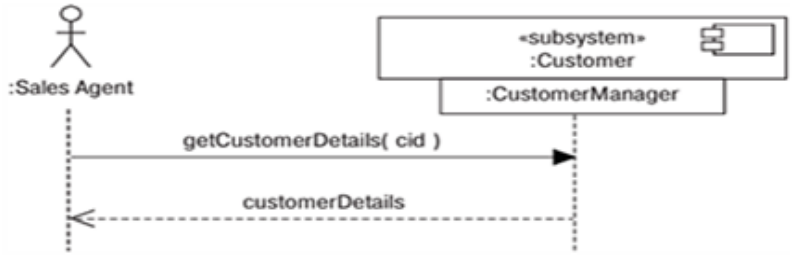


Рисунок 7.86 *Диаграмма взаимодействия актера с подсистемой* `Customer`

7.4.10 Временные диаграммы

Одним из слабых мест UML 1 было моделирование систем реального времени. Это такие системы, в которых временные соотношения критически важны и события должны следовать друг за другом в рамках определенного временного окна. Мы говорим «временное окно», а не «время», потому что абсолютное время для нас как разработчиков неприемлемо. Когда в модели задается время, на самом деле задается время плюс/минус некоторая погрешность, определяемая внешними факторами, такими как точность системных часов. Обычно это не является проблемой, за исключением систем с очень точными временными ограничениями.

Временные диаграммы моделируют временные ограничения.

В UML 1 временные ограничения можно было обозначать на разных диаграммах, но не было отдельной диаграммы, предназначенной именно для моделирования временных соотношений. UML 2 предоставляет разработчикам моделей систем реального времени временные диаграммы. Это разновидность диаграммы взаимодействий, основное внимание в которой направлено на моделирование временных ограничений. Таким образом, она идеально подходит для моделирования этого аспекта систем реального времени. Временные диаграммы, аналогичные временным диаграммам UML, в течение многих лет успешно используются в электронной промышленности для моделирования временных ограничений электронных схем. Временные диаграммы очень просты. Время откладывается на горизонтальной оси слева направо. Линии жизни и их состояния (или определённые условия, накладываемые на линии жизни) располагаются вертикально. Переходы между состояниями линий жизни и условиями представляются в виде графика. На рис. 7.87 приведена простая временная диаграмма для класса `Siren`. Эта диаграмма иллюстрирует, что происходит, когда формируется событие «вторжение», а затем событие «пожар». Конечно, это самый пессимистичный сценарий, но смоделировать его важно, чтобы понять взаимодействие функций обнаружения взломщика и пожара. Вот последовательный анализ данной временной диаграммы.

`t = 0: :Siren` находится в состоянии `Off` (выключен).

`t = 10:` Происходит событие `intruder`, и `:Siren` переходит в состояние `SoundingIntruderAlarm` (воспроизведение сигнала тревоги вторжения).

`t = 25:` Сигнал тревоги вторжения может звучать не более 15 минут согласно местным правилам подачи сигналов тревоги. `:Siren` переходит в состояние `Resting`

(покой). Он должен находиться в этом состоянии 30 минут (опять же по местным законам).

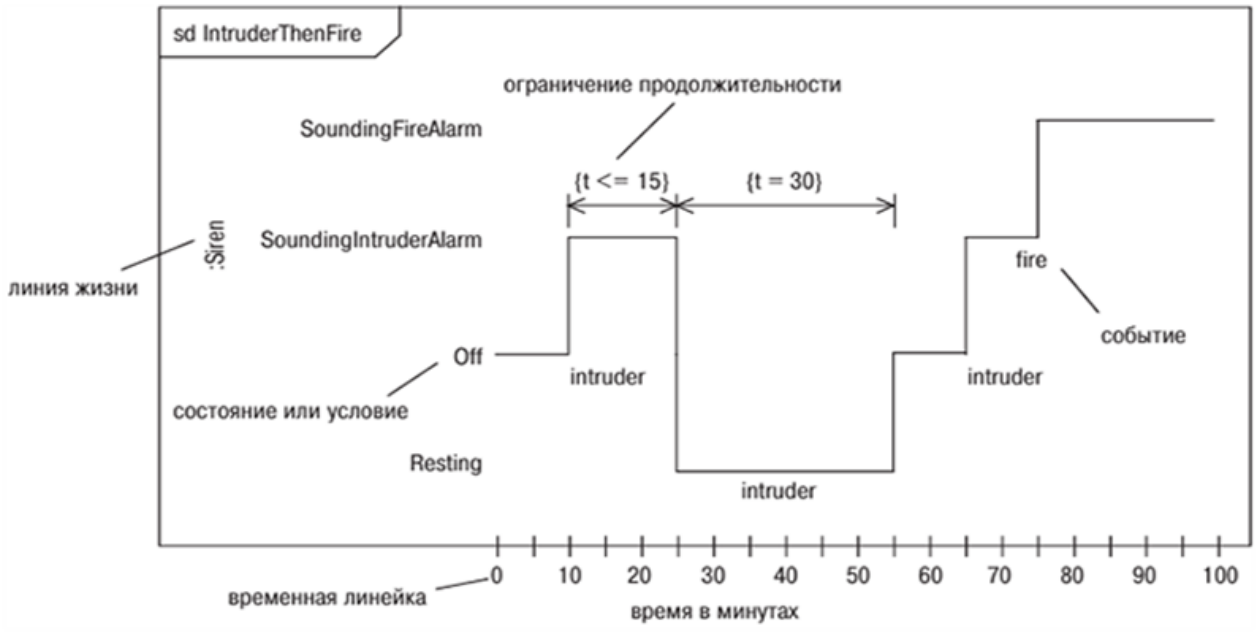


Рисунок 7.87 *Временная диаграмма для класса* `Siren`

`t = 35`: Происходит событие `intruder`, но `:Siren` в состоянии `Resting`, поэтому не может воспроизводить сигнал тревоги.

`t = 55`: `:Siren` возвращается в состояние `Off`.

`t = 65`: Возникает еще одно событие `intruder`. `:Siren` переходит из состояния `Off` в состояние `SoundingIntruderAlarm`.

`t = 75`: Происходит событие `fire`. `:Siren` переходит из состояния `SoundingIntruderAlarm` в состояние `SoundingFireAlarm` (воспроизведение сигнала пожарной тревоги).

`t = 100`: Взаимодействие завершается, `:Siren` остается в состоянии `SoundingFireAlarm`.

Временные диаграммы также можно представить в более компактной форме, когда состояния располагаются горизонтально. На рис. 7.88 в такой форме показана временная диаграмма с рис. 7.87. В такой компактной форме акцент обычно смещается больше на состояния и относительное время, а не на представление абсолютного времени, как это моделирует временная линейка.

Временные диаграммы могут использоваться для представления изменений состояния объекта во времени. Временные диаграммы также могут использоваться для иллюстрации временных ограничений во взаимодействиях между двумя или более линиями жизни.

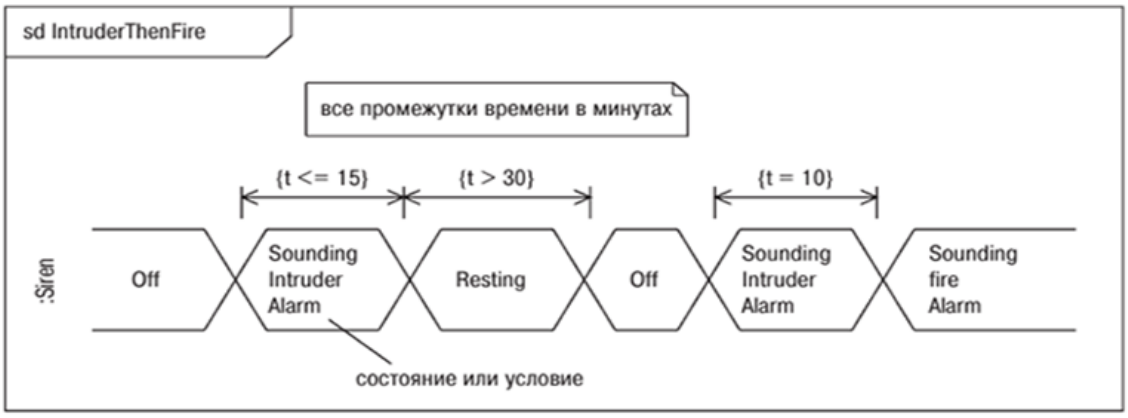


Рисунок 7.88 *Компактная форма временной диаграммы с рис. 7.87*

8 РЕФАКТОРИНГ

Рефакторинг является одной из ключевых практик в современной разработке программного обеспечения, направленной на постепенное улучшение существующего кода без изменения его внешнего поведения. В процессе него не добавляются новые функции, а лишь перерабатывается уже существующий код.

Рефакторинг программного кода преследует следующие цели:

- **Улучшение читаемости.** Прозрачный и легко читаемый код позволяет разработчикам быстрее ориентироваться в проекте, что упрощает поддержку и разработку новых функций.
- **Устранение дублирования.** Рефакторинг помогает обнаружить и объединить повторяющиеся участки кода, что сокращает трудозатраты на их поддержку и вероятность ошибок, появляющихся при необходимости изменить логику в нескольких местах.
- **Упрощение структур.** Зачастую код со временем усложняется за счет внесенных дополнений и исправлений. Упрощение может помочь вернуть код к более понятной, поддерживаемой форме.
- **Улучшение архитектуры.** Рефакторинг может привести к постепенному эволюционному улучшению структуры программы, сделать ее более гибкой и адаптируемой к новым условиям.
- **Оптимизация производительности.** В некоторых случаях рефакторинг направлен на устранение узких мест в коде, что может положительно отразиться на скорости работы программы.

Принципы рефакторинга кода

Сохранение функциональности. Изменения в коде выполняются таким образом, чтобы не нарушить его работоспособность. Даже если в коде много «некрасивых» решений, приоритет отдается стабильности работы приложения. Также изменения, выполняемые в процессе рефакторинга, не должны видоизменять внешний интерфейс программы и ее поведение с точки зрения пользователя.

Поочередные небольшие изменения. Выполнение преобразований небольшими последовательными шагами позволяет избежать ошибок, которые могут возникнуть при

крупных и радикальных переработках. Эта практика также облегчает отслеживание внесенных изменений и откат нежелательных эффектов.

Надежное покрытие тестами. Перед началом рефакторинга необходимо иметь обширный набор тестов, который покрывает все аспекты функциональности приложения. Тесты должны запускаться автоматически после каждого изменения, чтобы убедиться в отсутствии регрессии.

Регулярность и документирование. Не стоит дожидаться момента, когда код окончательно замусорится. Оптимальный подход — это выполнение преобразований как постоянного и регулярного процесса разработки. Хотя они чаще всего не изменяют функциональности, важно описывать, что и почему было изменено, чтобы коллеги-разработчики понимали происходящие трансформации кода.

Изучение изменяемых участков кода. Прежде чем рефакторить часть программы, разработчику нужно понимать ее функцию и влияние на остальные части системы. Изменения должны проводиться поэтапно, с постоянной интеграцией в основную кодовую базу.

В каких случаях необходим рефакторинг кода:

- **Код сложен и труден для понимания.** Плохая читаемость кода свидетельствует о высокой сложности его конструкций и отсутствии четкой структуры. Рефакторинг в этом случае направлен на упрощение структуры кода и на улучшение его логичности и организации. Таким образом код становится понятнее, что ускоряет разработку и облегчает обучение новых членов команды.
- **В коде много дублирования.** Частые повторения одинаковых или похожих фрагментов кода увеличивают вероятность ошибок при внесении изменений, так как каждый случай дублирования требует одинаковых корректировок. Это не только утяжеляет программу, но и влечет риски для ее поддержки и расширения. Рефакторинг с целью избавления от дублирования часто включает создание общих методов или классов. Сокращение дублированных частей упрощает процессы тестирования и сопровождения кода.
- **Трудоемкое внесение новых изменений.** Сложные зависимости между компонентами, запутанные условия и избыточные области кода могут серьезно осложнить процесс разработки. Рефакторинг помогает выстроить программу таким образом, чтобы внесение новшеств стало более простым и менее рискованным. Переосмысление и реструктуризация системы могут значительно повысить ее расширяемость и гибкость.
- **Код содержит устаревшие решения.** Устаревший код может стать причиной проблем с безопасностью, производительностью и интеграцией с другими системами. Рефакторинг с учетом последних достижений в области технологий дает возможность модернизировать приложение. Обновление кодовой базы обеспечивает лучшую поддержку, совместимость и предоставляет возможность использования нового функционала.
- **Производительность системы снижена.** Анализ показывает, что медленная работа часто связана не с инфраструктурными ограничениями, а с неоптимальным

кодом. Рефакторинг с акцентом на производительность может включать оптимизацию алгоритмов, устранение избыточных вычислений и уменьшение числа операций ввода-вывода. Улучшение эффективности кода часто ведет к лучшему пользовательскому опыту и сокращению ресурсов, необходимых для поддержки приложения.

8.1 Этапы рефакторинга кода

1. **Планирование.** Перед тем как приступить к рефакторингу, необходимо определить проблемные участки кода, которые требуют улучшений. Эти участки могут быть обнаружены в ходе код-ревью, как результат обратной связи от разработчиков, проверки кода статическими анализаторами или в результате выявления слабых мест при тестировании и эксплуатации программы.
2. **Обеспечение безопасности.** Перед началом преобразований крайне важно обеспечить кодовую базу набором тестов, который позволит контролировать, чтобы изменения, внесенные в процессе рефакторинга, не привели к потере функциональности. Как минимум необходимо иметь unit-тесты, покрывающие основную функциональность изменяемых участков кода.
3. **Выполнение рефакторинга.** Рефакторинг следует проводить итеративно, внося маленькие изменения за раз. Это помогает избежать серьезных ошибок и облегчает откат изменений в случае их неудачи. К таким шагам относятся: переименование переменных, разбиение больших функций на меньшие, удаление дублированного кода, улучшение структуры условных операторов и применение принципов SOLID.
4. **Тестирование после каждого изменения.** После каждой итерации необходимо запускать ранее подготовленные тесты для проверки, что внесенные изменения не навредили функциональности программы. Регулярное подтверждение правильности работы кода после каждого изменения дает уверенность в качестве рефакторинга и исправности системы.
5. **Документирование изменений.** Каждое существенное изменение в коде следует сопровождать соответствующими записями в системе контроля версий. Комментарии к коммитам должны быть информативными и понятными для других участников команды. Также стоит обновлять или создавать документацию, если изменения затрагивают архитектурные решения или принципы работы с код-базой.
6. **Ревью и слияние с основной веткой.** После завершения преобразований и успешного тестирования следует провести код-ревью с коллегами. Оно не только выявит возможные недочеты в рефакторинге, но и поможет разделить знания об изменениях в команде. После одобрения ревьюерами изменения можно сливать с основной веткой кодовой базы.
7. **Мониторинг и анализ.** После внедрения изменений в продакшен необходимо внимательно мониторить поведение системы на предмет производительности, стабильности и других ключевых метрик работы программы. Этот анализ может

не только показать прямой эффект от внесенных изменений, но и выявить неочевидные проблемы, которые не были обнаружены в ходе тестирования.

Рефакторинг кода – это не разовый проект, а скорее постоянная практика, цель которой – достичь максимальной чистоты и поддерживаемости кодовой базы в долгосрочной перспективе. Чем лучше встроен этот процесс в регулярные рабочие циклы команды разработки, тем выше уровень качества и надежности итогового продукта.

Даже у опытных программистов нередко бывает предчувствие, что в программе что-то не так, но не удается определить что именно. Фаулер замечательным образом систематизировал методы рефакторинга, а также отметил признаки плохого кода — так называемые «запахи», однако более фундаментальными для объектно-ориентированного подхода являются принципы SOLID.

8.2 ПРИНЦИПЫ SOLID

SOLID – это акроним, обозначающий пять основных принципов объектно-ориентированного программирования (ООП), которые помогают разработчикам создавать гибкие, расширяемые и поддерживаемые программные системы.

Эти принципы были представлены *Робертом Мартином* (также известным как *Дядюшка Боб*), а аббревиатуру позже ввел в обиход *Майкл Фэзерс*, и являются одними из наиболее важных руководящих принципов для разработки качественного кода. Зачем нужны SOLID принципы? Они помогают разработчикам создавать высококачественное ПО, которое легко поддерживать и модифицировать. Кроме того, они способствуют созданию кода, который легко читать, понимать и тестировать, и помогают избегать проблем, связанных с плохим проектированием и зависимостями между компонентами системы.

Вот что входит в принципы SOLID:

1. *Single Responsibility Principle (Принцип единственной ответственности).*
2. *Open Closed Principle (Принцип открытости/закрытости).*
3. *Liskov's Substitution Principle (Принцип подстановки Барбары Лисков).*
4. *Interface Segregation Principle (Принцип разделения интерфейса).*
5. *Dependency Inversion Principle (Принцип инверсии зависимостей).*

8.2.1 Принцип единственной ответственности (Single Responsibility Principle)

Принцип декларирует, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс, а все его сервисы должны быть направлены исключительно на обеспечение этой обязанности.

Следование принципу заключается обычно в декомпозиции сложных классов, которые делают сразу много вещей, на простые, ответственность которых очень специализирована. Но также и объединении в отдельный класс однотипной функциональности, которая может оказаться распределённой по многим классам, может рассматриваться как следование этому принципу.

Проектирование классов с направленностью на обеспечение единственной обязанности упрощает дальнейшие модификации и сопровождение, так как проще разобраться в одном блоке функциональности, нежели распутывать сложные

взаимосвязи между различными функциональными блоками. Также при модификации логики в одном месте приложения снижаются риски возникновения проблем в других «неожиданных» его местах.

Следование SRP весьма полезная практика с точки зрения повторного использования кода. Сложные объекты с комплексными зависимостями обычно очень сложно использовать повторно, особенно если нужна только часть реализованного в них функционала. А небольшие классы с чётко очерченным функционалом, напротив, проще использовать повторно, так как они не избыточные и редко тянут за собой существенный объём зависимостей.

Наиболее ярким анти-паттерном, нарушающим принцип единственной ответственности, является использование God-объектов, которые «слишком много знают» или «слишком много умеют». Возникают такие «божественные объекты» обычно из-за любви разработчиков к абстракции — если возводить абстракцию в абсолют, то вполне можно любой объект реального мира отразить в приложении в виде экзепляра некого универсального класса. На словах это даже может выглядеть логично, но на практике почти всегда это приводит к проблемам сопровождаемости. Обычно такие объекты становятся центральной частью системы, а их модификация крайне сложна, так как становится очень сложно предсказать, как изенение кода для решения текущей задачи может сказаться на ранее реализованной функциональности.

На самом деле, как и любые другие принципы, SRP требует сознательного и осмысленного применения. Чрезмерная декомпозиция может оказаться и вредной, если она приводит к большей сложности или усложняет сопровождение.

Представьте себе модуль, который обрабатывает заказы. Если заказ верно сформирован, он сохраняет его в базу данных и высылает письмо для подтверждения заказа:

```
public class OrderProcessor {

    public void process(Order order){
        if (order.isValid() && save(order)) {
            sendConfirmationEmail(order);
        }
    }

    private boolean save(Order order) {
        MySqlConnection connection = new MySqlConnection("database.url");
        return true;
    }

    private void sendConfirmationEmail(Order order) {
        String name = order.getCustomerName();
        String email = order.getCustomerEmail();


    }
```

```
}
```

Такой модуль может измениться по трем причинам. Во-первых может стать другой логика обработки заказа, во-вторых, способ его сохранения (тип базы данных), в-третьих – способ отправки письма подтверждения (скажем, вместо `email` нужно отправлять `SMS`). Принцип единственной обязанности подразумевает, что три аспекта этой проблемы на самом деле – три разные обязанности. А значит, должны находиться в разных классах или модулях. Объединение нескольких сущностей, которые могут меняться в разное время и по разным причинам, считается плохим проектным решением. Гораздо лучше разделить модуль на три отдельных, каждый из которых будет выполнять одну единственную функцию:

```
public class MySQLOrderRepository {
    public boolean save(Order order) {
        MySqlConnection connection = new MySqlConnection("database.url");
        return true;
    }
}

public class ConfirmationEmailSender {
    public void sendConfirmationEmail(Order order) {
        String name = order.getCustomerName();
        String email = order.getCustomerEmail();
    }
}

public class OrderProcessor {
    public void process(Order order){
        MySQLOrderRepository repository = new
MySQLOrderRepository();
        ConfirmationEmailSender mailSender = new
ConfirmationEmailSender();
        if (order.isValid() && repository.save(order)) {
            mailSender.sendConfirmationEmail(order);
        }
    }
}
```

8.2.2 Принцип открытости/закрытости (Open/Closed Principle)

Принцип декларирует, что программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода.

В этом контексте открытость для расширения – это возможность добавить для класса, модуля или функции новое поведение, если необходимость в этом возникнет, а закрытость для изменений – это запрет на изменение исходного кода программных сущностей. На первый взгляд, это звучит сложно и противоречиво. Но если разобраться, то принцип вполне логичен.

Следование принципу OCP заключается в том, что программное обеспечение изменяется не через изменение существующего кода, а через добавление нового кода.

То есть созданный изначально код остаётся «нетронутым» и стабильным, а новая функциональность внедряется либо через наследование реализации, либо через использование абстрактных интерфейсов и полиморфизм.

Принцип открытости/закрытости Мейера основывается на идее, что разработанная изначально реализация класса в дальнейшем не модифицируется (разве что исправляются ошибки), а любые изменения производятся через создание нового класса, который обычно наследуется от изначального. Согласно определению Мейера реализация интерфейса может быть унаследована и переиспользована, но интерфейс может и измениться в новой реализации.

Позже был сформулирован полиморфный принцип открытости/закрытости. Он основывается на строгой реализации интерфейсов и на наследовании от абстрактных базовых классов или на полиморфизме. Созданный изначально интерфейс должен быть закрыт для модификаций, а новые реализации как минимум соответсвуют этому изначальному интерфейсу, но могут поддерживать и другие, более расширенные.

Продолжая наш пример с заказом, предположим, что нам нужно выполнять какие-то действия перед обработкой заказа и после отправки письма с подтверждением. Вместо того, чтобы менять сам класс `OrderProcessor`, мы расширим его и добьемся решения поставленной задачи, не нарушая принцип OCP:

```
    public class OrderProcessorWithPreAndPostProcessing extends
OrderProcessor {

        @Override
        public void process(Order order) {
            beforeProcessing();
            super.process(order);
            afterProcessing();
        }

        private void beforeProcessing() {

        }

        private void afterProcessing() {

        }
    }
```

8.2.3 Принцип подстановки Барбары Лисков (Liskov Substitution Principle)

Принцип в формулировке Роберта Мартина декларирует, что функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом. Оригинальное определение Барбары Лисков более формальное и заметно сложнее для восприятия: «В том случае, если `q(x)` — свойство, верное по отношению к объектам `x` некого типа `T`, то свойство `q(y)` тоже будет верным относительно ряда объектов y, которые относятся к типу `S`, при этом `S` — подтип некого типа `T`».

Следование принципу LSP заключается в том, что при построении иерархий наследования создаваемые наследники должны корректно реализовывать поведение базового типа. То есть если базовый тип реализует определённое поведение, то это поведение должно быть корректно реализовано и для всех его наследников.

LSP перекликается с контрактным программированием, определяя точные, формальные и верифицируемые описания интерфейсов. И интерфейсы, реализумые наследниками, должны соответствовать контракту интерфейсов базового класса.

Наследник класса дополняет, но не заменяет поведение базового класса. То есть в любом месте программы замена базового класса на класс-наследник не должна вызывать проблем. Если по каким-то причинам так не получается, то вероятнее всего имеет место либо некорректная реализация, либо неверно выбранная абстракция для наследования.

Соблюдение принципа подстановки Барбары Лисков позволяет гарантировать, что любой созданный нами подкласс будет без проблем использоваться ранее реализованными модулями, которые работали с надклассом. А это существенно упрощает расширение функциональных возможностей системы.

Но LSP, как и любой другой принцип, не является догмой. И иногда следование этому принципу при построении архитектуры может приводить к более ресурсоёмкой реализации, нежели работа с нарушением этого принципа. Но как и с любыми другими правилами – надо осознавать возможные последствия нарушения.

Предположим, у нас есть класс, который отвечает за валидацию заказа и проверяет, все ли из товаров заказа находятся на складе. У данного класса есть метод `isValid` который возвращает *`true`* или *`false`*:

```
public class OrderStockValidator {

    public boolean isValid(Order order) {
        for (Item item : order.getItems()) {
            if (! item.isInStock()) {
                return false;
            }
        }

        return true;
    }
}
```

Также предположим, что некоторые заказы нужно валидировать иначе: проверять, все ли товары заказа находятся на складе и все ли товары упакованы. Для этого мы расширили класс `OrderStockValidator` классом `OrderStockAndPackValidator`:

```
public class OrderStockAndPackValidator extends OrderStockValidator{
    @Override
    public boolean isValid(Order order) {
        for (Item item : order.getItems()) {
            if ( !item.isInStock() || !item.isPacked() ){
                throw new IllegalStateException(
```

```
                        String.format("Order %d is not valid!",
order.getId())
                    );
                }
            }

            return true;
        }
    }
```

Однако в данном классе мы нарушили принцип LSP, так как вместо того, чтобы вернуть *`false`*, если заказ не прошел валидацию, наш метод бросает исключение `IllegalStateException`. Клиенты данного кода не рассчитывают на такое: они ожидают возвращения *`true`* или *`false`*. Это может привести к ошибкам в работе программы.

8.2.4 Принцип разделения интерфейса (Interface Segregation Principle)

Принцип в формулировке Роберта Мартина декларирует, что клиенты не должны зависеть от методов, которые они не используют. То есть если какой-то метод интерфейса не используется клиентом, то изменения этого метода не должны приводить к необходимости внесения изменений в клиентский код.

Следование принципу ISP заключается в создании интерфейсов, которые достаточно специфичны и требуют только необходимый минимум реализаций методов. Избыточные интерфейсы, напротив, могут требовать от реализующего класса создание большого количества методов, причём даже таких, которые не имеют смысла в контексте класса.

В чём-то принцип разделения интерфейса перекликается с принципом единственной ответственности — интерфейсы не должны быть избыточно «толстыми», если вдруг в приложении формируется слишком объёмный интерфейс, то есть высокая вероятность, что так происходит из-за того, что в этом интерфейсе слишком много разных ответственностей, а значит логичнее всего провести декомпозицию сложного интерфейса на несколько простых.

Принцип разделения интерфейса снижает сложность поддержки и развития приложения. Чем проще и минималистичнее используемый интерфейс, тем менее ресурсоёмкой является его реализация в новых классах, тем меньше причин его модифицировать, но и в случае модификации она будет значительно проще.

Рассмотрим пример. Разработчик Алекс создал интерфейс "отчет" и добавил два метода: `generateExcel()` и `generatedPdf()`. Теперь клиент А хочет использовать этот интерфейс, но он намерен использовать отчеты только в PDF-формате, а не в Excel. Устроит ли его такая функциональность?

Нет. Он должен будет реализовать два метода, один из которых по большому счету не нужен и существует только благодаря Алексу — дизайнеру программного обеспечения. Клиент воспользуется либо другим интерфейсом, либо оставит поле для Excel пустым.

Так в чем же решение? Оно состоит в разделении существующего интерфейса на два более мелких. Один – отчет в формате PDF, второй – отчет в формате Excel. Это даст пользователю возможность использовать только необходимый для него функционал.

8.2.5 Принцип инверсии зависимостей (Dependency Inversion Principle)

Принцип декларирует, что модули верхних уровней не должны зависеть от модулей нижних уровней, а оба типа модулей должны зависеть от абстракций; сами абстракции не должны зависеть от деталей, а вот детали должны зависеть от абстракций.

Следование принципу инверсии зависимостей «заставляет» реализовывать высокоуровневые компоненты без встраивания зависимостей от конкретных низкоуровневых классов, что, например, сильно упрощает замену используемых зависимостей как по бизнес-требованиям, так и для целей тестирования. При этом зависимость формируется не от конкретной реализации, а от абстракции — реализуемого зависимостью интерфейса.

Например, мы реализуем хранение документов в веб-приложении. На первый взгляд, кажется логичным добавить зависимость от модулей работы с файловой системой непосредственно в класс, отвечающий за высокоуровневую работу с этими документами. Но в перспективе такая зависимость может создать проблемы — например, нам потребуется хранить данные не только на диске, но и в облаке. Если зависимость внедрена от реализации, то мы столкнёмся с необходимостью её переработки. Если же зависимость выведена на уровень абстракции (интерфейса), то нам будет достаточно реализовать функционал работы с облаком, соответствующий ранее созданному интерфейсу работы с файлами.

Принцип инверсии зависимостей часто упрощает следованию принципу подстановки Барбары Лисков. Выделение абстракций и встраивание зависимостей от них снижает вероятность того, что в новом классе мы не полностью реализуем контракт базового класса, который мы расширяем в рамках нового.

9 ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Писать понятный код — сложно.

Каждая строчка кода увеличивает нагрузку на читателя, поэтому много кода читать сложно. С другой стороны, если в коде недостаточно деталей и объяснений, то понять его смысл тоже очень сложно. При написании программ нам надо соблюдать баланс между лаконичностью и полнотой, между абстракциями и конкретикой.

Найти этот баланс нам помогают решения, которые сообщество воспринимает как стандартные, шаблонные. Иногда такие решения получают имена и становятся паттернами или архитектурными подходами.

Люди плохо умеют прогнозировать будущее. Нам сложно предсказать, как будет меняться программа или требования к ней. Единственное, что мы знаем точно – требования (а значит, и программа) будут меняться.

Грамотная архитектура помогает спроектировать и развивать систему так, чтобы её было проще и удобнее расширять и изменять.

- Если общение между модулями регламентировано, их реализацию проще заменить на другую.
- Если общение с внешним миром регламентировано, меньше шансов для утечки данных.
- Если код разделён грамотно, программу проще тестировать.
- Если код организован понятно, уходит меньше времени на добавление новых фич и поиск багов в старых.
- Если архитектура широко известна, погружение в проект проходит быстрее.

Какие есть архитектурные подходы

Мы можем условно разделить архитектурные подходы по их целям и зоне действия. Часть подходов распределяют ответственность между модулями. Они определяют, какие модули и за что будут отвечать.

Самый известный из таких подходов — ***Model-View-Controller***. Он выделяет 3 «типа» задач, в соответствии с которыми назначает модулям роли.

Другие определяют, насколько каждый из модулей близок к бизнес-логике. Таким подходам важно, какая часть кода занимается непосредственно задачей приложения, а какая – инфраструктурными задачами.

Например, в приложении для обработки фотографий бизнес-логикой были бы функции фильтров, а инфраструктурными задачами – обращение к API камеры телефона.

В зависимости от степени близости к бизнес-логике такие подходы делят код на «слои». Самый распространённый подход среди таких – это трёхслойная архитектура.

Третьи управляют потоками данных в приложении. Они определяют, как модули общаются друг с другом: напрямую, опосредованно или с помощью специальных сервисов типа шины событий.

9.1 Паттерны проектирования

Некоторые проблемы слишком малы для выделения в архитектурный подход, но достаточно часто встречаются, чтобы породить стандартные решения. Такие стандартные решения называются паттернами или шаблонами.

Термин *«design patterns»* можно перевести с английского как паттерны/шаблоны/образцы проектирования. Они применяются в процессе создания информационных систем и являются формализованными описаниями регулярно возникающих задач проектирования, эффективными решениями таких задач и рекомендациями по использованию полученных решений в тех или иных ситуациях.

Стоит отметить, что адекватное моделирование рассматриваемой предметной области выступает в качестве важнейшей начальной стадии в процессе работы с паттернами. Данный этап нужен не только для правильной (грамотно формализованной) постановки задачи, но и для определения подходящих паттернов. Стоит отметить, что адекватное моделирование рассматриваемой предметной области выступает в качестве важнейшей начальной стадии в процессе работы с паттернами. Данный этап нужен не только для правильной (грамотно формализованной) постановки задачи, но и для определения подходящих паттернов.

Правильно применяя паттерны проектирования, разработчик получает массу преимуществ. Например, построенная с помощью паттернов модель будет менее сложна и более наглядна в изучении, относительно обычных моделей. Кроме того, она будет представлять собой структурированное выделение важных для решения задачи элементов. Вместе с тем, полученная модель позволит на глубочайшем уровне проработать архитектуру создаваемой системы при помощи специального языка.

Использования паттернов проектирования делает систему более устойчивой к смене требований. При этом дальнейшая доработка системы становится гораздо проще. Применение паттернов очень полезно и при интеграции информационных систем организации.

Комплекс паттернов проектирования можно смело назвать словарем разработчика. Это универсальное средство, позволяющее налаживать коммуникацию в процессе работы.

Можно выделить три типа паттернов: *порождающие, структурные и поведенческие*.

9.2 Порождающие (`Creational`)

Это группа шаблонов на основе механизма создания объектов, нацеленная на инкапсуляцию процесса инстанцирования объекта, благодаря которой этот процесс гибче, а его связанность с клиентским кодом слабее. Они убирают лишнее дублирование, делают процесс создания объектов короче и прямолинейнее.

Среди порождающих паттернов можно выделить:

- **Прототип (`Prototype`)**. Такой шаблон задаёт типы создаваемых объектов посредством экземпляра-прототипа. Более того, копируя прототип, паттерн формирует новые объекты. При использовании этого шаблона вы сможете отойти от реализации и придерживаться принципа «программирование через интерфейсы». В роли возвращающего типа выступает указанный интерфейс/абстрактный класс на вершине иерархии. При этом классы-наследники могут подставить в это место наследника, выполняющего реализацию данного типа. Иными словами, прототип позволяет создать объект посредством клонирования, а не с помощью конструктора.
- **Factory (`Фабрика`)**. Это не считается паттерном в строгом смысле слова. Правильнее будет обозначить Factory как подход, в рамках которого логика создания объектов выносится в отдельный класс.
- **Фабричный метод (`Factory Method`)**. Шаблон, который определят общий интерфейс для формирования объектов в суперклассе. Данный метод даёт подклассам возможность изменять вид создаваемых объектов.
- **Абстрактная фабрика (`Abstract factory`)**. Это порождающий паттерн проектирования, который предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, без непосредственной спецификации их конкретных классов. Реализация такого шаблона обеспечивается за счёт создания абстрактного класса Factory. Этот класс выступает в качестве интерфейса для создания компонентов системы (скажем,

применительно к оконному интерфейсу он может формировать окна и кнопки). После всего этого пишутся классы, которые реализуют данный интерфейс.

- **Одиночка (`Singleton`).** Очередной порождающий паттерн. С помощью него обеспечивается наличие единственного экземпляра класса с глобальной точкой доступа в однопоточном приложении.
- **Строитель (`Builder`).** Полезный порождающий паттерн, который, по сути, является методом создания составного объекта. Он дифференцирует сложный объект на конструирование и представление. Благодаря этому при выполнении одной и той же операции конструирования вы можете получить разные представления.

Остановимся подробнее на каждом из них.

9.2.1 Паттерн `Prototype` (прототип)

Назначение паттерна `Prototype`

Паттерн `Prototype` (прототип) можно использовать в следующих случаях:

- Система должна оставаться независимой как от процесса создания новых объектов, так и от типов порождаемых объектов. Непосредственное использование выражения *new* в коде приложения считается нежелательным.
- Необходимо создавать объекты, точные классы которых становятся известными уже на стадии выполнения программы.

Паттерн `Factory Method` также делает систему независимой от типов порождаемых объектов, но для этого он вводит параллельную иерархию классов: для каждого типа создаваемого объекта должен присутствовать соответствующий класс-фабрика, что может быть нежелательно. Паттерн `Prototype` лишен этого недостатка.

Описание паттерна `Prototype`

Для создания новых объектов паттерн `Prototype` использует прототипы. Прототип – это уже существующий в системе объект, который поддерживает операцию клонирования, то есть умеет создавать копию самого себя. Таким образом, для создания объекта некоторого класса достаточно выполнить операцию `clone()` соответствующего прототипа.

Паттерн `Prototype` реализует подобное поведение следующим образом: все классы, объекты которых нужно создавать, должны быть подклассами одного общего абстрактного базового класса. Этот базовый класс должен объявлять интерфейс метода `clone()`. Также здесь могут объявляться виртуальными и другие общие методы, например, `initialize()` в случае, если после клонирования нужна инициализация вновь созданного объекта. Все производные классы должны реализовывать метод `clone()`. В языке C++ для создания копий объектов используется конструктор копирования, однако, в общем случае, создание объектов при помощи операции копирования не является обязательным.

UML-диаграмма классов паттерна `Prototype`

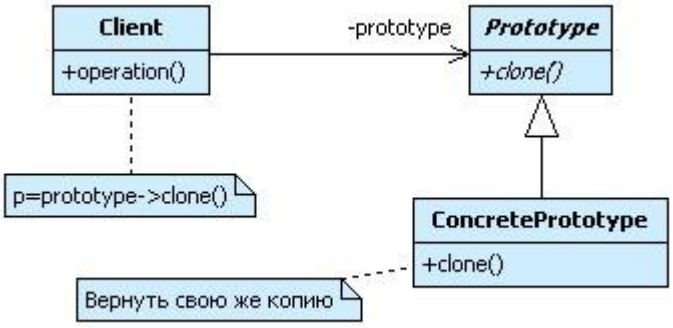


Рисунок 9.1

Для порождения объекта некоторого типа в системе должен существовать его прототип. Прототип представляет собой объект того же типа, единственным назначением которого является создание подобных ему объектов. Обычно для удобства все существующие в системе прототипы организуются в специальные коллекции-хранилища или реестры прототипов. Такое хранилище может иметь реализацию в виде ассоциативного массива, каждый элемент которого представляет пару «Идентификатор типа» – «Прототип». Реестр прототипов позволяет добавлять или удалять прототип, а также создавать объект по идентификатору типа. Именно операции динамического добавления и удаления прототипов в хранилище обеспечивают дополнительную гибкость системе, позволяя управлять процессом создания новых объектов.

Достоинства паттерна `Prototype`

- Для создания новых объектов клиенту необязательно знать их конкретные классы.
- Возможность гибкого управления процессом создания новых объектов за счет возможности динамических добавления и удаления прототипов в реестр.

Недостатки паттерна `Prototype`

- Каждый тип создаваемого продукта должен реализовывать операцию клонирования `clone()`. В случае, если требуется **глубокое копирование** объекта (объект содержит ссылки или указатели на другие объекты), это может быть непростой задачей.

9.2.2 Паттерн `Factory Method` (фабричный метод)

Назначение паттерна `Factory Method`

В системе часто требуется создавать объекты самых разных типов. Паттерн `Factory Method` (фабричный метод) может быть полезным в решении следующих задач:

- Система должна оставаться расширяемой путем добавления объектов новых типов. Непосредственное использование выражения `new` является нежелательным, так как в этом случае код создания объектов с указанием конкретных типов может получиться разбросанным по всему приложению. Тогда

такие операции как добавление в систему объектов новых типов или замена объектов одного типа на другой будут затруднительными. Паттерн `Factory Method` позволяет системе оставаться независимой как от самого процесса порождения объектов, так и от их типов.

- Заранее известно, когда нужно создавать объект, но неизвестен его тип.

Описание паттерна `Factory Method`

Для того, чтобы система оставалась независимой от различных типов объектов, паттерн `Factory Method` использует механизм полиморфизма - классы всех конечных типов наследуют от одного абстрактного базового класса, предназначенного для полиморфного использования. В этом базовом классе определяется единый интерфейс, через который пользователь будет оперировать объектами конечных типов.

Для обеспечения относительно простого добавления в систему новых типов паттерн `Factory Method` локализует создание объектов конкретных типов в специальном классе-фабрике. Методы этого класса, посредством которых создаются объекты конкретных классов, называются фабричными. Существуют две разновидности паттерна `Factory Method`:

Обобщенный конструктор, когда в том же самом полиморфном базовом классе, от которого наследуют производные классы всех создаваемых в системе типов, определяется статический фабричный метод. В качестве параметра в этот метод должен передаваться идентификатор типа создаваемого объекта.

UML-диаграмма классов паттерна `Factory Method`. Обобщенный конструктор

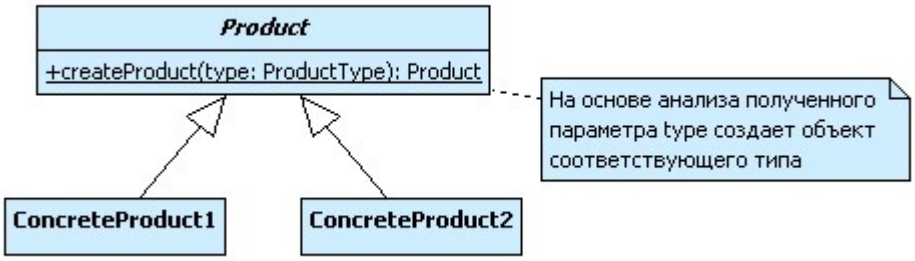


Рисунок 9.2

Классический вариант фабричного метода, когда интерфейс фабричных методов объявляется в независимом классе-фабрике, а их реализация определяется конкретными подклассами этого класса.

UML-диаграмма классов паттерна `Factory Method`. Классическая реализация

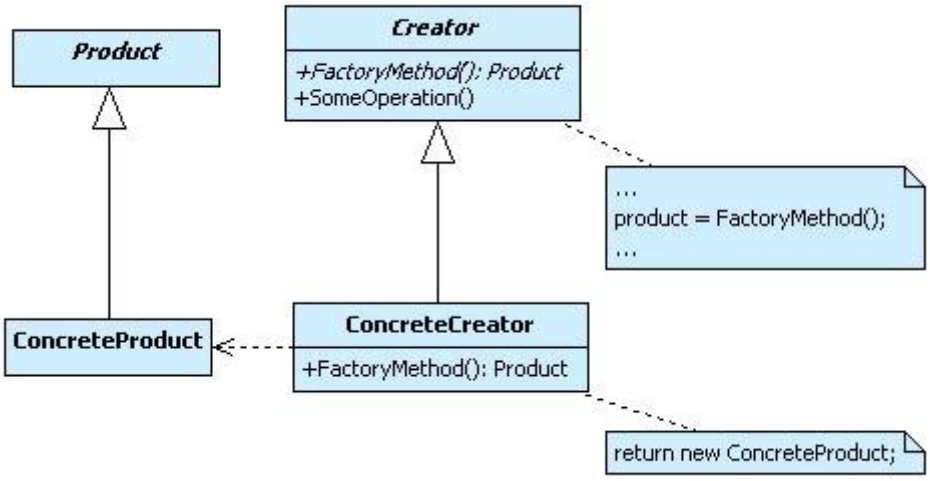


Рисунок 9.3

Шаги реализации

1. Приведите все создаваемые продукты к общему интерфейсу.
2. В классе, который производит продукты, создайте пустой фабричный метод. В качестве возвращаемого типа укажите общий интерфейс продукта.
3. Затем пройдитесь по коду класса и найдите все участки, создающие продукты. Поочерёдно замените эти участки вызовами фабричного метода, перенося в него код создания различных продуктов.
4. В фабричный метод, возможно, придётся добавить несколько параметров, контролирующих, какой из продуктов нужно создать.
5. На этом этапе фабричный метод, скорее всего, будет выглядеть удручающе. В нём будет жить большой условный оператор, выбирающий класс создаваемого продукта. Но не волнуйтесь, мы вот-вот исправим это.
6. Для каждого типа продуктов заведите подкласс и переопределите в нём фабричный метод. Переместите туда код создания соответствующего продукта из суперкласса.
7. Если создаваемых продуктов слишком много для существующих подклассов создателя, вы можете подумать о введении параметров в фабричный метод, которые позволят возвращать различные продукты в пределах одного подкласса.
8. Например, у вас есть класс `Почта` с подклассами `АвиаПочта` и `НаземнаяПочта`, а также классы продуктов `Самолёт`, `Грузовик` и `Поезд`. Авиа соответствует `Самолётам`, но для `НаземнойПочты` есть сразу два продукта. Вы могли бы создать новый подкласс почты для поездов, но проблему можно решить и по-другому. Клиентский код может передавать в фабричный метод `НаземнойПочты` аргумент, контролирующий тип создаваемого продукта.

9. Если после всех перемещений фабричный метод стал пустым, можете сделать его абстрактным. Если в нём что-то осталось – не беда, это будет его реализацией по умолчанию.

Достоинства паттерна `Factory Method`

- Избавляет класс от привязки к конкретным классам продуктов.
- Выделяет код производства продуктов в одно место, упрощая поддержку кода.
- Упрощает добавление новых продуктов в программу.
- Реализует *принцип открытости/закрытости*.

Недостатки паттерна `Factory Method`

- В случае классического варианта паттерна даже для порождения единственного объекта необходимо создавать соответствующую фабрику

9.2.3 Абстрактная фабрика (`Abstract Factory`)

Представьте, что вы пишете симулятор мебельного магазина. Ваш код содержит:

1. Семейство зависимых продуктов. Скажем, `Кресло + Диван + Столик`.
2. Несколько вариаций этого семейства. Например, продукты `Кресло`, `Диван` и `Столик` представлены в трёх разных стилях: `Ар-деко`, `Викторианском` и `Модерне`.

Вам нужен такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами того же семейства. Это важно, так как клиенты расстраиваются, если получают несочетающуюся мебель.

Кроме того, вы не хотите вносить изменения в существующий код при добавлении новых продуктов или семейств в программу. Поставщики часто обновляют свои каталоги, и вы бы не хотели менять уже написанный код каждый раз при получении новых моделей мебели.

Для начала паттерн `Абстрактная фабрика` предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства. Так, все вариации кресел получат общий интерфейс `Кресло`, все диваны реализуют интерфейс `Диван` и так далее.

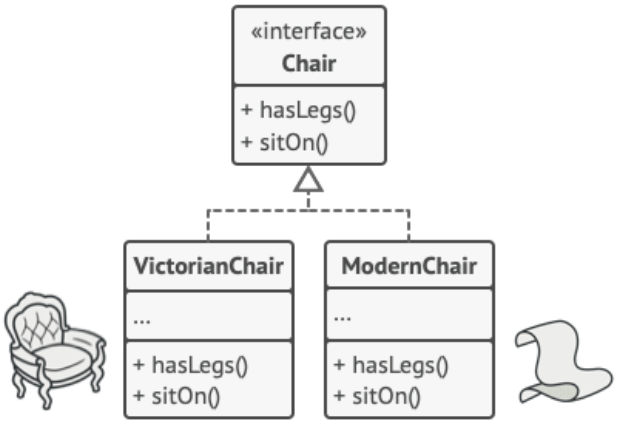


Рисунок 9.4

Далее вы создаёте *абстрактную фабрику* – общий интерфейс, который содержит методы создания всех продуктов семейства (например, `создатьКресло`, `создатьДиван` и `создатьСтолик`). Эти операции должны возвращать **абстрактные** типы продуктов, представленные интерфейсами, которые мы выделили ранее – `Кресла`, `Диваны` и `Столики`.

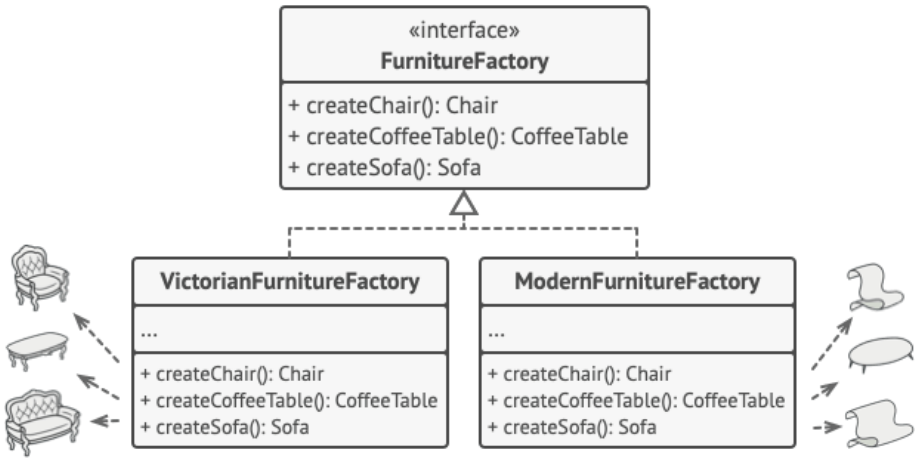


Рисунок 9.5

Как насчёт вариаций продуктов? Для каждой вариации семейства продуктов мы должны создать свою собственную фабрику, реализовав абстрактный интерфейс. Фабрики создают продукты одной вариации. Например, `ФабрикаМодерн` будет возвращать только `КреслаМодерн`, `ДиваныМодерн` и `СтоликиМодерн`.

Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы. Это позволит подавать в ваши классы любой тип фабрики и производить любые продукты, ничего не ломая.

Например, клиентский код просит фабрику сделать стул. Он не знает, какого типа была эта фабрика. Он не знает, получит викторианский или модерновый стул. Для него важно, чтобы на стуле можно было сидеть и чтобы этот стул отлично смотрелся с диваном той же фабрики.

Осталось прояснить последний момент: кто создаёт объекты конкретных фабрик, если клиентский код работает только с интерфейсами фабрик? Обычно программа создаёт конкретный объект фабрики при запуске, причём тип фабрики выбирается, исходя из параметров окружения или конфигурации.

Структура

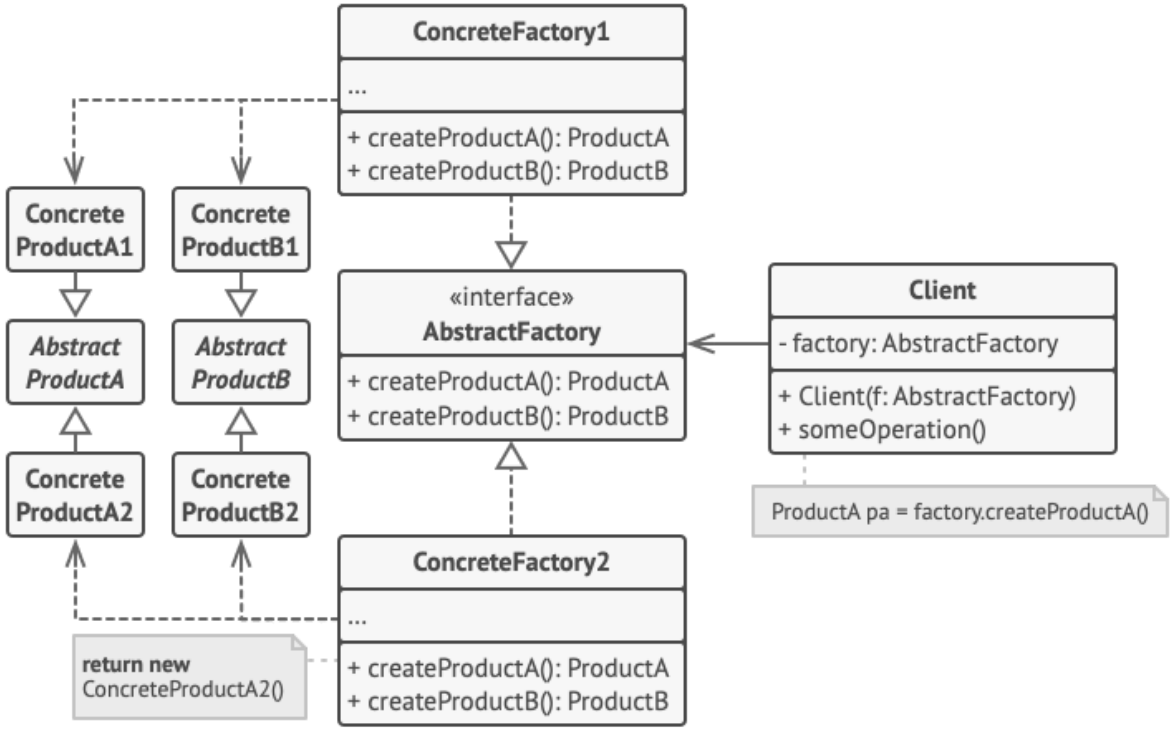


Рисунок 9.6

1. **Абстрактные продукты** объявляют интерфейсы продуктов, которые связаны друг с другом по смыслу, но выполняют разные функции.
2. **Конкретные продукты** – большой набор классов, которые относятся к различным абстрактным продуктам (кресло/столик), но имеют одни и те же вариации (Викторианский/Модерн).
3. **Абстрактная фабрика** объявляет методы создания различных абстрактных продуктов (кресло/столик).
4. **Конкретные фабрики** относятся каждая к своей вариации продуктов (Викторианский/Модерн) и реализуют методы абстрактной фабрики, позволяя создавать все продукты определённой вариации.
5. Несмотря на то, что конкретные фабрики порождают конкретные продукты, сигнатуры их методов должны возвращать соответствующие абстрактные продукты. Это позволит клиентскому коду, использующему фабрику, не привязываться к конкретным классам продуктов. Клиент сможет работать с любыми вариациями продуктов через абстрактные интерфейсы.

Реализация паттерна `Абстрактная фабрика`

В этом примере `Абстрактная фабрика` создаёт кросс-платформенные элементы интерфейса и следит за тем, чтобы они соответствовали выбранной операционной системе.

Кросс-платформенная программа может показывать одни и те же элементы интерфейса, выглядящие чуточку по-другому в различных операционных системах. В такой программе важно, чтобы все создаваемые элементы всегда соответствовали

текущей операционной системе. Вы бы не хотели, чтобы программа, запущенная на Windows, вдруг начала показывать чекбоксы в стиле macOS.

Абстрактная фабрика объявляет список создающих методов, которые клиентский код может использовать для получения тех или иных разновидностей элементов интерфейса. Конкретные фабрики относятся к различным операционным системам и создают элементы, совместимые с этой системой.

В самом начале программа определяет, какая из фабрик соответствует текущей операционке. Затем создаёт эту фабрику и отдаёт её клиентскому коду. В дальнейшем клиент будет работать только с этой фабрикой, чтобы исключить несовместимость возвращаемых продуктов.

Клиентский код не зависит от конкретных классов фабрик и элементов интерфейса. Он общается с ними через абстрактные интерфейсы. Благодаря этому клиент может работать с любой разновидностью фабрик и элементов интерфейса.

Чтобы добавить в программу новую вариацию элементов (например, для поддержки Linux), вам не нужно трогать клиентский код. Достаточно создать ещё одну фабрику, производящую эти элементы.

```
// Этот паттерн предполагает, что у вас есть несколько семейств
// продуктов, находящихся в отдельных иерархиях классов
// (Button/Checkbox). Продукты одного семейства должны иметь
// общий интерфейс.
interface Button is
    method paint()

// Семейства продуктов имеют те же вариации (macOS/Windows).
class WinButton implements Button is
    method paint() is
        // Отрисовать кнопку в стиле Windows.

class MacButton implements Button is
    method paint() is
        // Отрисовать кнопку в стиле macOS.


interface Checkbox is
    method paint()

class WinCheckbox implements Checkbox is
    method paint() is
        // Отрисовать чекбокс в стиле Windows.

class MacCheckbox implements Checkbox is
    method paint() is
        // Отрисовать чекбокс в стиле macOS.


// Абстрактная фабрика знает обо всех абстрактных типах
// продуктов.
```

```
interface GUIFactory is
    method createButton():Button
    method createCheckbox():Checkbox


// Каждая конкретная фабрика знает и создаёт только продукты
// своей вариации.
class WinFactory implements GUIFactory is
    method createButton():Button is
        return new WinButton()
    method createCheckbox():Checkbox is
        return new WinCheckbox()

// Несмотря на то, что фабрики оперируют конкретными классами,
// их методы возвращают абстрактные типы продуктов. Благодаря
// этому фабрики можно взаимозаменять, не изменяя клиентский
// код.
class MacFactory implements GUIFactory is
    method createButton():Button is
        return new MacButton()
    method createCheckbox():Checkbox is
        return new MacCheckbox()


// Для кода, использующего фабрику, не важно, с какой конкретно
// фабрикой он работает. Все получатели продуктов работают с
// ними через общие интерфейсы.
class Application is
    private field factory: GUIFactory
    private field button: Button
    constructor Application(factory: GUIFactory) is
        this.factory = factory
    method createUI()
        this.button = factory.createButton()
    method paint()
        button.paint()


// Приложение выбирает тип конкретной фабрики и создаёт её
// динамически, исходя из конфигурации или окружения.
class ApplicationConfigurator is
    method main() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            factory = new WinFactory()
        else if (config.OS == "Mac") then
            factory = new MacFactory()
        else
            throw new Exception("Error! Unknown operating system.")
```

```
Application app = new Application(factory)
```

Шаги реализации

1. Создайте таблицу соотношений типов продуктов к вариациям семейств продуктов.
2. Сведите все вариации продуктов к общим интерфейсам.
3. Определите интерфейс абстрактной фабрики. Он должен иметь фабричные методы для создания каждого из типов продуктов.
4. Создайте классы конкретных фабрик, реализовав интерфейс абстрактной фабрики. Этих классов должно быть столько же, сколько и вариаций семейств продуктов.
5. Измените код инициализации программы так, чтобы она создавала определённую фабрику и передавала её в клиентский код.
6. Замените в клиентском коде участки создания продуктов через конструктор вызовами соответствующих методов фабрики.

Достоинства паттерна `Абстрактная фабрика`

- Гарантирует сочетаемость создаваемых продуктов.
- Избавляет клиентский код от привязки к конкретным классам продуктов.
- Выделяет код производства продуктов в одно место, упрощая поддержку кода.
- Упрощает добавление новых продуктов в программу.
- Реализует принцип открытости/закрытости.

Недостатки паттерна `Абстрактная фабрика`

- Усложняет код программы из-за введения множества дополнительных классов.
- Требует наличия всех типов продуктов в каждой вариации.

9.2.4 Паттерн `Singleton`

Назначение паттерна `Singleton`

Часто в системе могут существовать сущности только в единственном экземпляре, например, система ведения системного журнала сообщений или драйвер дисплея. В таких случаях необходимо уметь создавать единственный экземпляр некоторого типа, предоставлять к нему доступ извне и запрещать создание нескольких экземпляров того же типа.

Паттерн `Singleton` предоставляет такие возможности.

Описание паттерна `Singleton`

Архитектура паттерна `Singleton` основана на идее использования глобальной переменной, имеющей следующие важные свойства:

1. Такая переменная доступна всегда. Время жизни глобальной переменной – от запуска программы до ее завершения.

2. Предоставляет глобальный доступ, то есть, такая переменная может быть доступна из любой части программы.

Однако, использовать глобальную переменную некоторого типа непосредственно невозможно, так как существует проблема обеспечения единственности экземпляра, а именно, возможно создание нескольких переменных того же самого типа (например, стековых).

Для решения этой проблемы паттерн `Singleton` возлагает контроль над созданием единственного объекта на сам класс. Доступ к этому объекту осуществляется через статическую функцию-член класса, которая возвращает указатель или ссылку на него. Этот объект будет создан только при первом обращении к методу, а все последующие вызовы просто возвращают его адрес. Для обеспечения уникальности объекта, конструкторы и оператор присваивания объявляются закрытыми.

UML-диаграмма классов паттерна `Singleton`

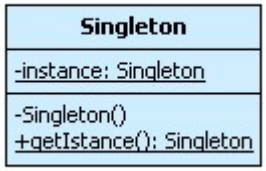


Рисунок 9.7

Паттерн `Singleton` часто называют усовершенствованной глобальной переменной.

Реализация паттерна `Singleton`

Классическая реализация `Singleton`

Рассмотрим наиболее часто встречающуюся реализацию паттерна `Singleton`.

```
// Singleton.h
class Singleton
{
  private:
    static Singleton * p_instance;
    // Конструкторы и оператор присваивания недоступны клиентам
    Singleton() {}
    Singleton( const Singleton& );
    Singleton& operator=( Singleton& );
  public:
    static Singleton * getInstance() {
        if(!p_instance)
            p_instance = new Singleton();
        return p_instance;
    }
```

```
};

// Singleton.cpp
#include "Singleton.h"

Singleton* Singleton::p_instance = 0;
```

Клиенты запрашивают единственный объект класса через статическую функцию-член `getInstance()`, которая при первом запросе динамически выделяет память под этот объект и затем возвращает указатель на этот участок памяти. Впоследствии клиенты должны сами позаботиться об освобождении памяти при помощи оператора `delete`.

Последняя особенность является серьезным недостатком классической реализации шаблона `Singleton`. Так как класс сам контролирует создание единственного объекта, было бы логичным возложить на него ответственность и за разрушение объекта. Этот недостаток отсутствует в реализации `Singleton`, впервые предложенной Скоттом Мэйерсом.

`Singleton` Мэйерса

```
// Singleton.h
class Singleton
{
private:
    Singleton() {}
    Singleton( const Singleton&);
    Singleton& operator=( Singleton& );
public:
    static Singleton& getInstance() {
        static Singleton  instance;
        return instance;
    }
};
```

Внутри `getInstance()` используется статический экземпляр нужного класса. Стандарт языка программирования C++ гарантирует автоматическое уничтожение статических объектов при завершении программы. Досрочного уничтожения и не требуется, так как объекты `Singleton` обычно являются долгоживущими объектами. Статическая функция-член `getInstance()` возвращает не указатель, а ссылку на этот объект, тем самым, затрудняя возможность ошибочного освобождения памяти клиентами.

Приведенная реализация паттерна `Singleton` использует так называемую отложенную инициализацию (`lazy initialization`) объекта, когда объект класса инициализируется не при старте программы, а при первом вызове `getInstance()`. В данном случае это обеспечивается тем, что статическая переменная `instance` объявлена внутри функции – члена класса `getInstance()`, а не как статический член данных этого класса. Отложенную инициализацию, в первую очередь, имеет смысл

использовать в тех случаях, когда инициализация объекта представляет собой дорогостоящую операцию и не всегда используется.

К сожалению, у реализации Мэйерса есть недостатки: сложности создания объектов производных классов и невозможность безопасного доступа нескольких клиентов к единственному объекту в многопоточной среде.

Улучшенная версия классической реализации `Singleton`

С учетом всего вышесказанного классическая реализация паттерна `Singleton` может быть улучшена.

```
// Singleton.h
class Singleton;  // опережающее объявление

class SingletonDestroyer
{
  private:
    Singleton* p_instance;
  public:
    ~SingletonDestroyer();
    void initialize( Singleton* p );
};

class Singleton
{
  private:
    static Singleton* p_instance;
    static SingletonDestroyer destroyer;
  protected:
    Singleton() { }
    Singleton( const Singleton& );
    Singleton& operator=( Singleton& );
   ~Singleton() { }
    friend class SingletonDestroyer;
  public:
    static Singleton& getInstance();
};

// Singleton.cpp
#include "Singleton.h"

Singleton * Singleton::p_instance = 0;
SingletonDestroyer Singleton::destroyer;

SingletonDestroyer::~SingletonDestroyer() {
    delete p_instance;
}
void SingletonDestroyer::initialize( Singleton* p ){
    p_instance = p;
}
```

```
Singleton& Singleton::getInstance() {
    if(!p_instance)     {
        p_instance = new Singleton();
        destroyer.initialize( p_instance);
    }
    return *p_instance;
}
```

Ключевой особенностью этой реализации является наличие класса `SingletonDestroyer`, предназначенного для автоматического разрушения объекта `Singleton`. Класс `Singleton` имеет статический член `SingletonDestroyer`, который инициализируется при первом вызове `Singleton::getInstance()` создаваемым объектом `Singleton`. При завершении программы этот объект будет автоматически разрушен деструктором `SingletonDestroyer` (для этого `SingletonDestroyer` объявлен другом класса `Singleton`).

Для предотвращения случайного удаления пользователями объекта класса `Singleton`, деструктор теперь уже не является общедоступным как ранее. Он объявлен защищенным.

Использование нескольких взаимозависимых одиночек

До сих пор предполагалось, что в программе используется один одиночка либо несколько несвязанных между собой. При использовании взаимосвязанных одиночек появляются новые вопросы:

- Как гарантировать, что к моменту использования одного одиночки, экземпляр другого зависимого уже создан?
- Как обеспечить возможность безопасного использования одного одиночки другим при завершении программы? Другими словами, как гарантировать, что в момент разрушения первого одиночки в его деструкторе еще возможно использование второго зависимого одиночки (то есть второй одиночка к этому моменту еще не разрушен)?

Управлять порядком создания одиночек относительно просто. Следующий код демонстрирует один из возможных методов.

```
// Singleton.h
class Singleton1
{
  private:
    Singleton1() { }
    Singleton1( const Singleton1& );
    Singleton1& operator=( Singleton1& );
  public:
    static Singleton1& getInstance() {
        static Singleton1 instance;
        return instance;
    }
};
```

```
class Singleton2
{
  private:
    Singleton2( Singleton1& instance): s1( instance)
{ }
    Singleton2( const Singleton2& );
    Singleton2& operator=( Singleton2& );
    Singleton1& s1;
  public:
    static Singleton2& getInstance() {
        static Singleton2 instance(
Singleton1::getInstance());
        return instance;
    }
};

// main.cpp
#include "Singleton.h"

int main()
{
    Singleton2& s = Singleton2::getInstance();
    return 0;
}
```

Объект `Singleton1` гарантированно инициализируется раньше объекта `Singleton2`, так как в момент создания объекта `Singleton2` происходит вызов `Singleton1::getInstance()`.

Гораздо сложнее управлять временем жизни одиночек. Существует несколько способов это сделать, каждый из них обладает своими достоинствами и недостатками и заслуживают отдельного рассмотрения.

Несмотря на кажущуюся простоту паттерна `Singleton` (используется всего один класс), его реализация не является тривиальной.

Результаты применения паттерна `Singleton`

Достоинства паттерна `Singleton`

- Класс сам контролирует процесс создания единственного экземпляра.
- Паттерн легко адаптировать для создания нужного числа экземпляров.
- Возможность создания объектов классов, производных от `Singleton`.

Недостатки паттерна `Singleton`

- В случае использования нескольких взаимозависимых одиночек их реализация может резко усложниться.

9.2.5 Строитель (`Builder`)

Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

Представьте сложный объект, требующий кропотливой пошаговой инициализации множества полей и вложенных объектов. Код инициализации таких

объектов обычно спрятан внутри монструозного конструктора с десятком параметров. Либо ещё хуже – распылён по всему клиентскому коду.

Например, давайте подумаем о том, как создать объект `Дом`. Чтобы построить стандартный дом, нужно поставить 4 стены, установить двери, вставить пару окон и положить крышу. Но что, если вы хотите дом побольше да посветлее, имеющий сад, бассейн и прочее добро?

Самое простое решение – расширить класс `Дом`, создав подклассы для всех комбинаций параметров дома. Проблема такого подхода – это громадное количество классов, которые вам придётся создать. Каждый новый параметр, вроде цвета обоев или материала кровли, заставит вас создавать всё больше и больше классов для перечисления всех возможных вариантов.

Чтобы не плодить подклассы, вы можете подойти к решению с другой стороны. Вы можете создать гигантский конструктор `Дома`, принимающий уйму параметров для контроля над создаваемым продуктом. Действительно, это избавит вас от подклассов, но приведёт к другой проблеме.

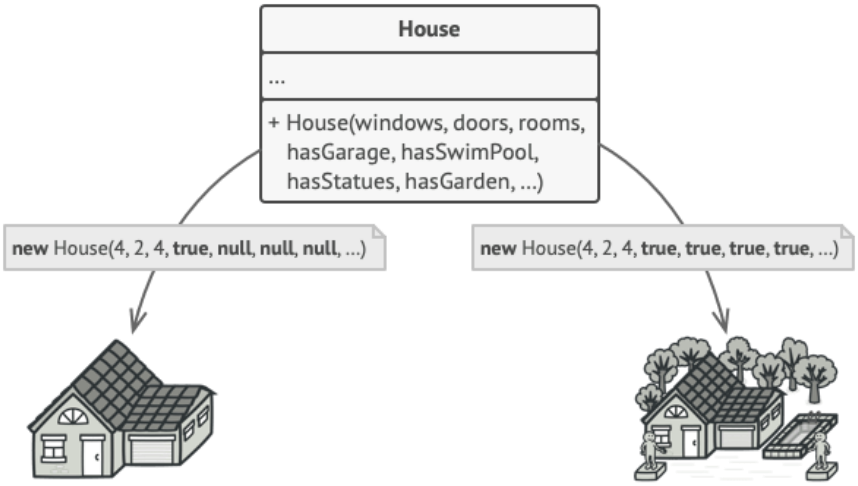


Рисунок 9.8

Большая часть этих параметров будет простаивать, а вызовы конструктора будут выглядеть монструозно из-за *длинного списка параметров*. К примеру, далеко не каждый дом имеет бассейн, поэтому параметры, связанные с бассейнами, будут простаивать бесполезно в 99% случаев.

Паттерн Строитель предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, которые следует называть строителями.

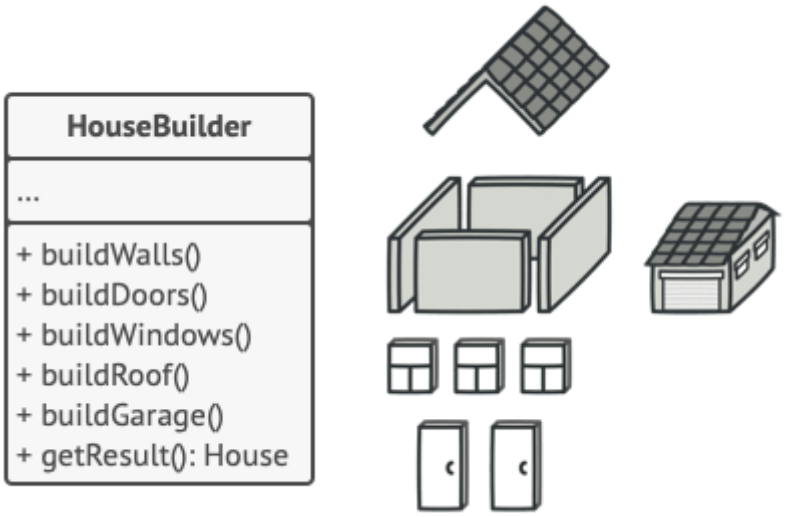


Рисунок 9.9

Паттерн предлагает разбить процесс конструирования объекта на отдельные шаги (например, `построитьСтены`, `вставитьДвери` и другие). Чтобы создать объект, вам нужно поочерёдно вызывать методы строителя. Причём не нужно запускать все шаги, а только те, что нужны для производства объекта определённой конфигурации.

Зачастую один и тот же шаг строительства может отличаться для разных вариаций производимых объектов. Например, деревянный дом потребует строительства стен из дерева, а каменный – из камня.

В этом случае вы можете создать несколько классов строителей, выполняющих одни и те же шаги по-разному. Используя этих строителей в одном и том же строительном процессе, вы сможете получать на выходе различные объекты.

Вы можете пойти дальше и выделить вызовы методов строителя в отдельный класс, называемый директором. В этом случае директор будет задавать порядок шагов строительства, а строитель – выполнять их.

Структура

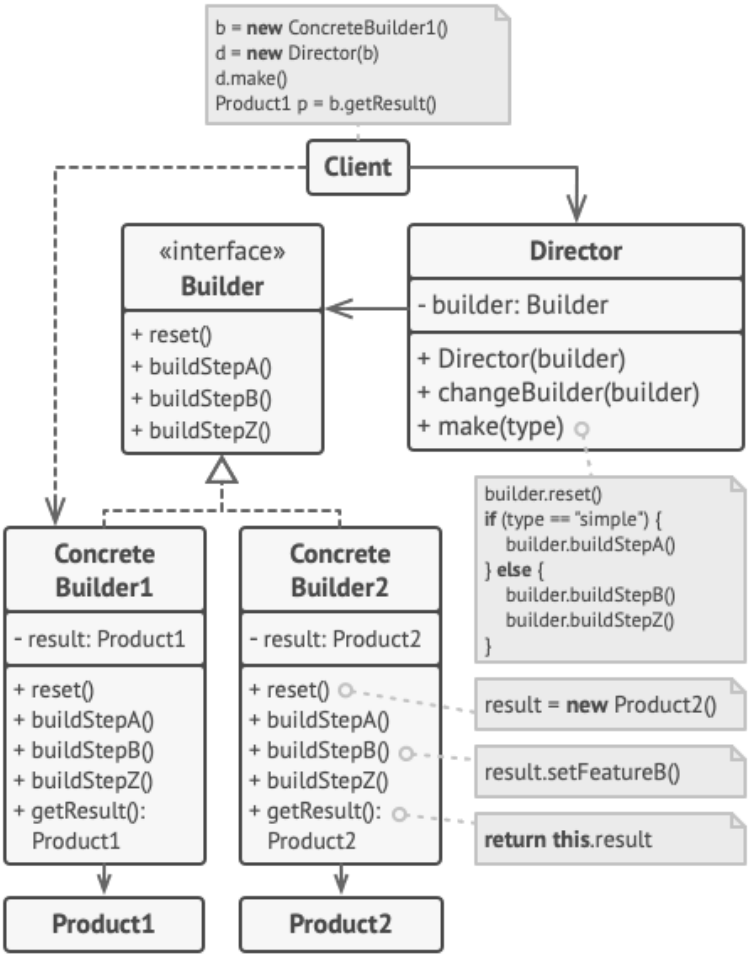


Рисунок 9.10

1. **Интерфейс строителя** объявляет шаги конструирования продуктов, общие для всех видов строителей.
2. **Конкретные строители** реализуют строительные шаги, каждый по-своему. Конкретные строители могут производить разнородные объекты, не имеющие общего интерфейса.
3. **Продукт** – создаваемый объект. Продукты, сделанные разными строителями, не обязаны иметь общий интерфейс.
4. **Директор** определяет порядок вызова строительных шагов для производства той или иной конфигурации продуктов.
5. Обычно **Клиент** подаёт в конструктор директора уже готовый объект-строитель, и в дальнейшем данный директор использует только его. Но возможен и другой вариант, когда клиент передаёт строителя через параметр строительного метода директора. В этом случае можно каждый раз применять разных строителей для производства различных представлений объектов.

Пример использования паттерна `Builder`

В этом примере **Строитель** используется для пошагового конструирования автомобилей, а также технических руководств к ним.

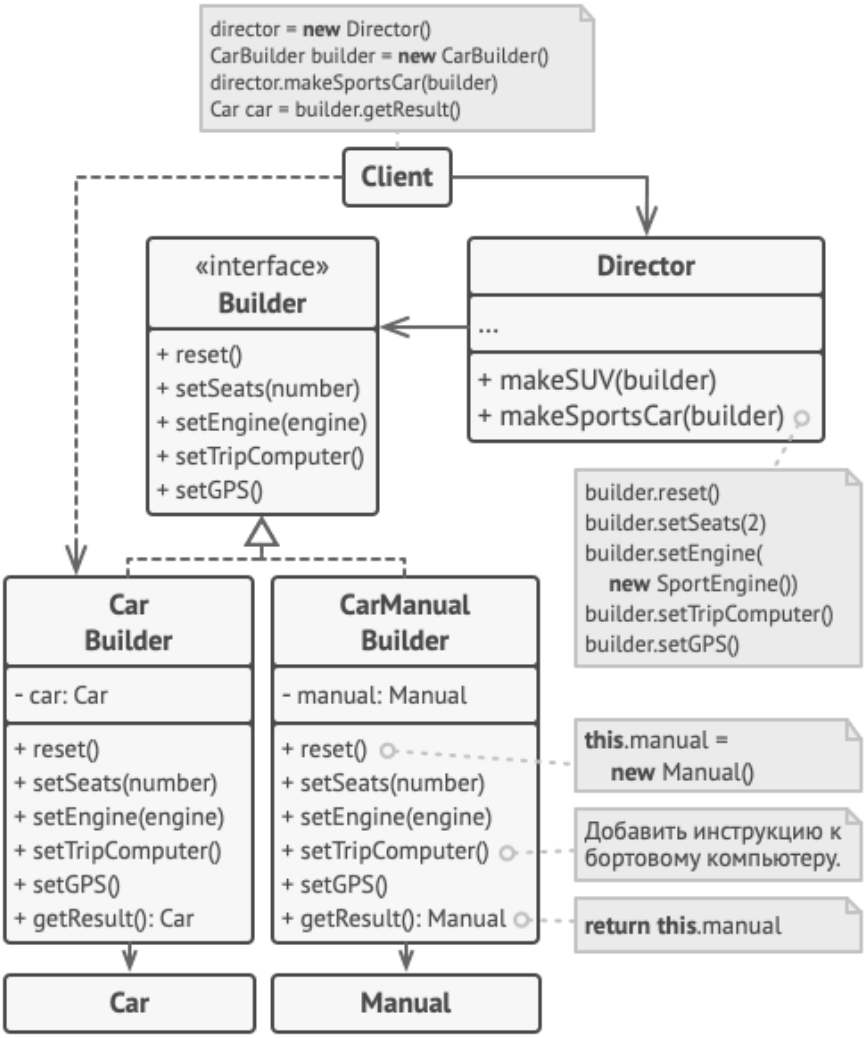


Рисунок 9.11

Автомобиль – это сложный объект, который может быть сконфигурирован сотней разных способов. Вместо того, чтобы настраивать автомобиль через конструктор, мы вынесем его сборку в отдельный класс-строитель, предусмотрев методы для конфигурации всех частей автомобиля.

Клиент может собирать автомобили, работая со строителем напрямую. Но, с другой стороны, он может поручить это дело директору. Это объект, который знает, какие шаги строителя нужно вызвать, чтобы получить несколько самых популярных конфигураций автомобилей.

Но к каждому автомобилю нужно ещё и руководство, совпадающее с его конфигурацией. Для этого мы создадим ещё один класс строителя, который вместо конструирования автомобиля, будет печатать страницы руководства к той детали, которую мы встраиваем в продукт. Теперь, пропустив оба типа строителей через одни и те же шаги, мы получим автомобиль и подходящее к нему руководство пользователя.

Очевидно, что бумажное руководство и железный автомобиль – это две разных вещи, не имеющих ничего общего. По этой причине мы должны получать результат напрямую от строителей, а не от директора. Иначе нам пришлось бы жёстко привязать директора к конкретным классам автомобилей и руководств.

```
// Строитель может создавать различные продукты, используя один
// и тот же процесс строительства.
class Car is
    // Автомобили могут отличаться комплектацией: типом
    // двигателя, количеством сидений, могут иметь или не иметь
    // GPS и систему навигации и т. д. Кроме того, автомобили
    // могут быть городскими, спортивными или внедорожниками.

class Manual is
    // Руководство пользователя для данной конфигурации
    // автомобиля.


// Интерфейс строителя объявляет все возможные этапы и шаги
// конфигурации продукта.
interface Builder is
    method reset()
    method setSeats(...)
    method setEngine(...)
    method setTripComputer(...)
    method setGPS(...)

// Все конкретные строители реализуют общий интерфейс по-своему.
class CarBuilder implements Builder is
    private field car:Car
    method reset()
        // Поместить новый объект Car в поле "car".
    method setSeats(...) is
        // Установить указанное количество сидений.
    method setEngine(...) is
        // Установить поданный двигатель.
    method setTripComputer(...) is
        // Установить поданную систему навигации.
    method setGPS(...) is
        // Установить или снять GPS.
    method getResult():Car is
        // Вернуть текущий объект автомобиля.

// В отличие от других порождающих паттернов, где продукты
// должны быть частью одной иерархии классов или следовать
// общему интерфейсу, строители могут создавать совершенно
// разные продукты, которые не имеют общего предка.
class CarManualBuilder implements Builder is
    private field manual:Manual
    method reset()
        // Поместить новый объект Manual в поле "manual".
```

```
    method setSeats(...) is
        // Описать, сколько мест в машине.
    method setEngine(...) is
        // Добавить в руководство описание двигателя.
    method setTripComputer(...) is
        // Добавить в руководство описание системы навигации.
    method setGPS(...) is
        // Добавить в инструкцию инструкцию GPS.
    method getResult():Manual is
        // Вернуть текущий объект руководства.


// Директор знает, в какой последовательности нужно заставлять
// работать строителя, чтобы получить ту или иную версию
// продукта. Заметьте, что директор работает со строителем через
// общий интерфейс, благодаря чему он не знает тип продукта,
// который изготовляет строитель.
class Director is
    method constructSportsCar(builder: Builder) is
        builder.reset()
        builder.setSeats(2)
        builder.setEngine(new SportEngine())
        builder.setTripComputer(true)
        builder.setGPS(true)


// Директор получает объект конкретного строителя от клиента
// (приложения). Приложение само знает, какого строителя нужно
// использовать, чтобы получить определённый продукт.
class Application is
    method makeCar() is
        director = new Director()

        CarBuilder builder = new CarBuilder()
        director.constructSportsCar(builder)
        Car car = builder.getResult()

        CarManualBuilder builder = new CarManualBuilder()
        director.constructSportsCar(builder)

        // Готовый продукт возвращает строитель, так как
        // директор чаще всего не знает и не зависит от
        // конкретных классов строителей и продуктов.
        Manual manual = builder.getResult()
```

Шаги реализации

1. Убедитесь в том, что создание разных представлений объекта можно свести к общим шагам.
2. Опишите эти шаги в общем интерфейсе строителей.

3. Для каждого из представлений объекта-продукта создайте по одному классу-строителю и реализуйте их методы строительства.
4. Не забудьте про метод получения результата. Обычно конкретные строители определяют собственные методы получения результата строительства. Вы не можете описать эти методы в интерфейсе строителей, поскольку продукты не обязательно должны иметь общий базовый класс или интерфейс. Но вы всегда сможете добавить метод получения результата в общий интерфейс, если ваши строители производят однородные продукты с общим предком.
5. Подумайте о создании класса директора. Его методы будут создавать различные конфигурации продуктов, вызывая разные шаги одного и того же строителя.
6. Клиентский код должен будет создавать и объекты строителей, и объект директора. Перед началом строительства клиент должен связать определённого строителя с директором. Это можно сделать либо через конструктор, либо через сеттер, либо подав строителя напрямую в строительный метод директора.
7. Результат строительства можно вернуть из директора, но только если метод возврата продукта удалось поместить в общий интерфейс строителей. Иначе вы жёстко привяжете директора к конкретным классам строителей.

Преимущества `Строителя`

- Позволяет создавать продукты пошагово.
- Позволяет использовать один и тот же код для создания различных продуктов.
- Изолирует сложный код сборки продукта от его основной бизнес-логики.

Преимущества и недостатки `Строителя`

- Усложняет код программы из-за введения дополнительных классов.
- Клиент будет привязан к конкретным классам строителей, так как в интерфейсе директора может не быть метода получения результата.

9.3 Структурные (Structural)

Такие шаблоны определяют всевозможные структуры высокого уровня сложности, которые вносят изменения в интерфейс уже созданных объектов или его реализацию. Это позволяет упростить процесс разработки и сделать программу более оптимизированной.

Шаблоны, входящие в эту разновидность, облегчают проектирование за счёт того, что они выявляют простейший метод реализации отношений между субъектами.

Шаблоны, входящие в эту разновидность, облегчают проектирование за счёт того, что они выявляют простейший метод реализации отношений между субъектами.

- **Мост (`Bridge`)**. Применяется в проектировании ПО. Он разделяет абстракцию и реализацию таким образом, чтобы они могли меняться независимо. Данный паттерн работает с помощью инкапсуляции, агрегирования и может применять наследование в целях распределения межклассовой ответственности.
- **Декоратор (`Decorator`)**. Данный шаблон был сформирован для динамического подключения дополнительного поведения к объекту. С его помощью практика создания подклассов получает гибкую альтернативу. Это позволяет сделать функционал более широким.

- **Адаптер (`Adapter`)**. Он необходим для организации применения функций объекта, который нельзя модифицировать, посредством специального интерфейса.
- **Компоновщик (`Composite pattern`)**. Такой шаблон способен объединять объекты в древовидную структуру. Это полезно для представления иерархии от частного к целому. Тем самым Composite pattern даёт клиентам возможность обращаться к отдельным объектам и к группам объектов одинаковым образом.
- **Заместитель (`Proxy`)**. Он предоставляет объект, контролирующий доступ к другому объекту, перехватывая при этом все вызовы. Иными словами, он выступает в качестве контейнера `Фасад` (`Facade`). Он помогает скрыть сложность системы. Принцип работы довольно прост: все возможные внешние вызовы сводятся к одному и тому же объекту, который передаёт эти вызовы соответствующим объектам системы.
- **Приспособленец (`Flyweight`, «легковесный (элемент)»)**. В процессе применения данного паттерна объект представляет себя как уникальный экземпляр в разных частях программы, однако, на самом деле это не так.

Остановимся подробнее на некоторых из них.

9.3.1 Мост (Bridge)

Синонимы: Описатель Тело (Handle Body)

Отделяет абстракцию от реализации, благодаря чему появляется возможность независимо изменять то и другое.

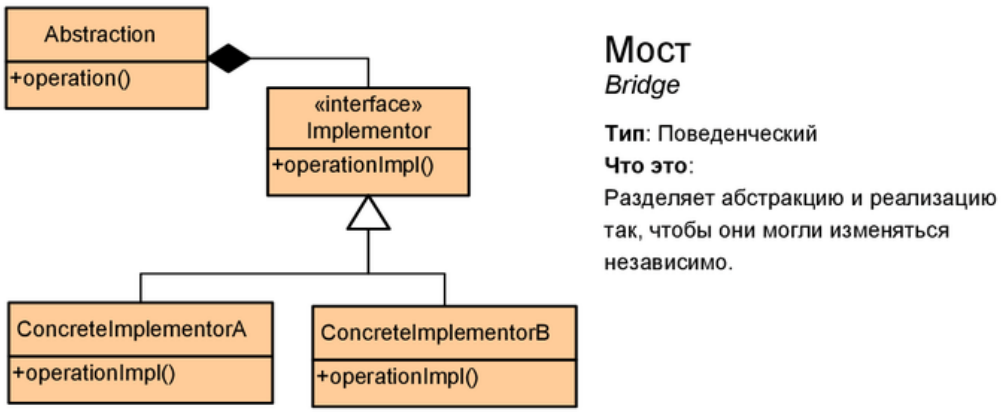


Рисунок 9.12

Применимость

- необходимо избежать постоянной привязки абстракции к реализации. Так, например, бывает, когда реализацию необходимо выбирать во время выполнения программы;
- и абстракции, и реализации должны расширяться новыми подклассами. В таком случае паттерн мост позволяет комбинировать разные абстракции и реализации и изменять их независимо;

- изменения в реализации абстракции не должны сказываться на клиентах, то есть клиентский код не должен перекомпилироваться;

Используйте паттерн мост, когда:

1. хотите избежать постоянной привязки абстракции к реализации. Так, например, бывает, когда реализацию необходимо выбирать во время выполнения программы;
2. абстракции, и реализации должны расширяться новыми подклассами. В таком случае паттерн мост позволяет комбинировать разные абстракции и реализации и изменять их независимо;
3. изменения в реализации абстракции не должны сказываться на клиентах, то есть клиентский код не должен перекомпилироваться;
4. (только для C++!) вы хотите полностью скрыть от клиентов реализацию абстракции. В C++ представление класса видимо через его интерфейс;
5. число классов начинает быстро расти (что создаёт проблему), Это признак того, что иерархию следует разделить на две части;
6. вы хотите разделить одну реализацию между несколькими объектами (быть может, применяя подсчет ссылок), и этот факт необходимо скрыть от клиента.

Отношения

Объект `Abstraction` перенаправляет своему объекту `Implementor` запросы клиента.

Шаблон мост(англ. Bridge) – структурный шаблон проектирования, используемый в проектировании программного обеспечения чтобы «разделять абстракцию и реализацию так, чтобы они могли изменяться независимо». Шаблон мост использует инкапсуляцию, агрегирование и может использовать наследование для того, чтобы разделить ответственность между классами.

Казалось бы, чем не устраивает механизм наследования? Почему был придуман этот паттерн? Все очень просто – коллективная разработка. Действительно, в больших проектах, это обыденное дело – программирование/проектирование абстракций и реализаций. Причем, порой бывает просто необходимым совершенно независимо развивать/модифицировать эти два, казалось бы связанных друг с другом понятия. При использовании механизма наследования это очень затруднительно. Любые изменения в интерфейсе абстракции тут же должны быть имплементированы в реализации.

Описание паттерна `Bridge`

Паттерн Bridge разделяет абстракцию и реализацию на две отдельные иерархии классов так, что их можно изменять независимо друг от друга.

Первая иерархия определяет интерфейс абстракции, доступный пользователю. Для случая проектируемого нами логгера абстрактный базовый класс `Logger` мог бы объявить интерфейс метода `log()` для вывода сообщений. Класс `Logger` также содержит указатель на реализацию `pimpl`, который инициализируется должным образом при создании логгера конкретного типа. Этот указатель используется для перенаправления пользовательских запросов в реализацию. Заметим, в общем случае подклассы `ConsoleLogger`, `FileLogger` и `SocketLogger` могут расширять интерфейс класса `Logger`.

Все детали реализации, связанные с особенностями среды скрываются во второй иерархии. Базовый класс `LoggerImpl` объявляет интерфейс операций, предназначенных для отправки сообщений на экран, файл и удаленный компьютер, а подклассы `ST_LoggerImpl` и `MT_LoggerImpl` его реализуют для однопоточной и многопоточной среды соответственно. В общем случае, интерфейс `LoggerImpl` необязательно должен в точности соответствовать интерфейсу абстракции. Часто он выглядит как набор низкоуровневых примитивов.

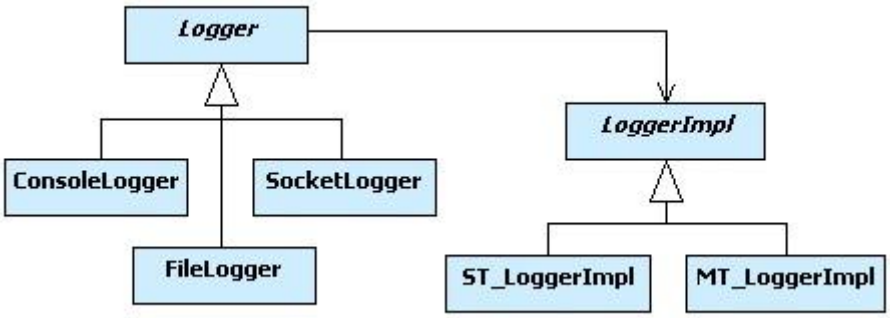


Рисунок 9.13

Паттерн `Bridge` позволяет легко изменить реализацию во время выполнения программы. Для этого достаточно перенастроить указатель `pimpl` на объект-реализацию нужного типа. Применение паттерна Bridge также позволяет сократить общее число подклассов в системе, что делает ее более простой в поддержке.

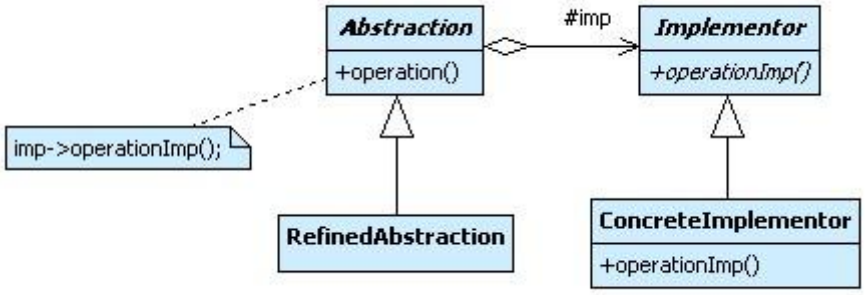


Рисунок 9.14

```
// Logger.h - Абстракция
#include <string>

// Опережающее объявление
class LoggerImpl;

class Logger
{
  public:
    Logger( LoggerImpl* p );
    virtual ~Logger( );
```

```
    virtual void log( string & str ) = 0;
  protected:
    LoggerImpl * pimpl;
};

class ConsoleLogger : public Logger
{
  public:
    ConsoleLogger();
    void log( string & str );
};

class FileLogger : public Logger
{
  public:
    FileLogger( string & file_name );
    void log( string & str );
  private:
    string file;
};

class SocketLogger : public Logger
{
  public:
    SocketLogger( string & remote_host, int remote_port );
    void log( string & str );
  private:
    string host;
    int    port;
};
```

// Logger.cpp - Абстракция

```
#include "Logger.h"
#include "LoggerImpl.h"

Logger::Logger( LoggerImpl* p ) : pimpl(p)
{ }

Logger::~Logger( )
{
    delete pimpl;
}

ConsoleLogger::ConsoleLogger() : Logger(
      #ifdef MT
        new MT_LoggerImpl()
      #else
        new ST_LoggerImpl()
      #endif
    )
```

```
{ }

void ConsoleLogger::log( string & str )
{
    pimpl->console_log( str);
}

FileLogger::FileLogger( string & file_name ) : Logger(
      #ifdef MT
        new MT_LoggerImpl()
      #else
        new ST_LoggerImpl()
      #endif
    ), file(file_name) { }

void FileLogger::log( string & str )
{
    pimpl->file_log( file, str);
}

SocketLogger::SocketLogger( string & remote_host,
                            int remote_port ) : Logger(
      #ifdef MT
        new MT_LoggerImpl()
      #else
        new ST_LoggerImpl()
      #endif
    ), host(remote_host), port(remote_port)
{ }

void SocketLogger::log( string & str )
{
    pimpl->socket_log( host, port, str);
}


// LoggerImpl.h - Реализация
#include <string>

class LoggerImpl
{
  public:
    virtual ~LoggerImpl( ) {}
    virtual void console_log( string & str ) = 0;
    virtual void file_log(
                    string & file, string & str ) = 0;
    virtual void socket_log(
           tring & host, int port, string & str ) = 0;
};

class ST_LoggerImpl : public LoggerImpl
```

```
{
  public:
    void console_log( string & str );
    void file_log   ( string & file, string & str );
    void socket_log (
            string & host, int port, string & str );
};

class MT_LoggerImpl : public LoggerImpl
{
  public:
    void console_log( string & str );
    void file_log   ( string & file, string & str );
    void socket_log (
            string & host, int port, string & str );
};
```

// LoggerImpl.cpp - Реализация

```
#include <iostream>
#include "LoggerImpl.h"


void ST_LoggerImpl::console_log( string & str )
{
  cout << "Single-threaded console logger" << endl;
}

void ST_LoggerImpl::file_log( string & file, string & str )
{
  cout << "Single-threaded file logger" << endl;
}

void ST_LoggerImpl::socket_log(string & host, int port, string & str)
{
  cout << "Single-threaded socket logger" << endl;
};

void MT_LoggerImpl::console_log( string & str )
{
  cout << "Multithreaded console logger" << endl;
}

void MT_LoggerImpl::file_log( string & file, string & str )
{
  cout << "Multithreaded file logger" << endl;
}

void MT_LoggerImpl::socket_log(
                    string & host, int port, string & str )
{
```

```
  cout << "Multithreaded socket logger" << endl;
}

// Main.cpp
#include <string>
#include "Logger.h"

int main()
{
  Logger * p = new FileLogger( string("log.txt"));
  p->log( string("message"));
  delete p;
  return 0;
}
```

Отметим несколько важных моментов приведенной реализации паттерна `Bridge`:

1. При модификации реализации клиентский код перекомпилировать не нужно.
2. Пользователь класса `Logger` не видит никаких деталей его реализации.

Результаты применения паттерна `Bridge`

Достоинства паттерна `Bridge`

- Проще расширять систему новыми типами за счет сокращения общего числа родственных подклассов.
- Возможность динамического изменения реализации в процессе выполнения программы.
- Паттерн `Bridge` полностью скрывает реализацию от клиента. В случае модификации реализации пользовательский код не требует перекомпиляции.

9.3.2 Адаптер (`Adapter`)

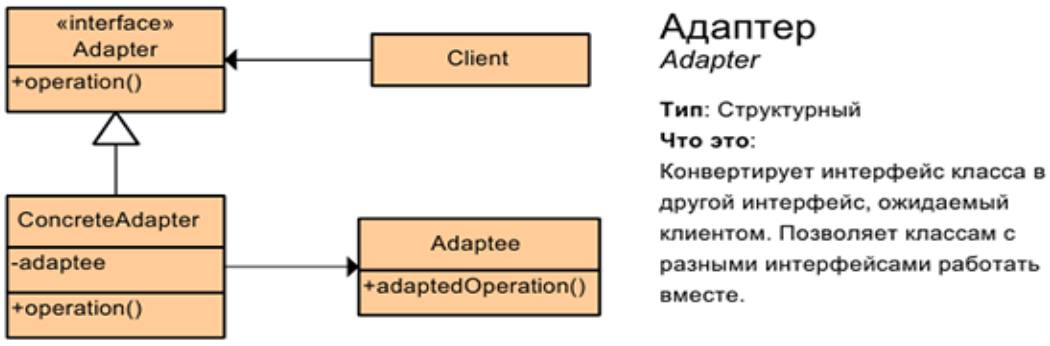


Рисунок 9.15

Назначение паттерна `Adapter`

Часто в новом программном проекте не удается повторно использовать уже существующий код. Например, имеющиеся классы могут обладать нужной функциональностью, но иметь при этом несовместимые интерфейсы. В таких случаях следует использовать паттерн `Adapter` (адаптер).

Паттерн Adapter, представляющий собой программную обертку над существующими классами, преобразует их интерфейсы к виду, пригодному для последующего использования.

Рассмотрим простой пример, когда следует применять паттерн Adapter. Пусть мы разрабатываем систему климат-контроля, предназначенной для автоматического поддержания температуры окружающего пространства в заданных пределах. Важным компонентом такой системы является температурный датчик, с помощью которого измеряют температуру окружающей среды для последующего анализа. Для этого датчика уже имеется готовое программное обеспечение от сторонних разработчиков, представляющее собой некоторый класс с соответствующим интерфейсом. Однако использовать этот класс непосредственно не удастся, так как показания датчика снимаются в градусах Фаренгейта. Нужен адаптер, преобразующий температуру в шкалу Цельсия.

Контейнеры queue, priority_queue и stack библиотеки стандартных шаблонов STL реализованы на базе последовательных контейнеров list, deque и vector, адаптируя их интерфейсы к нужному виду. Именно поэтому эти контейнеры называют контейнерами-адаптерами.

Описание паттерна Adapter

Пусть класс, интерфейс которого нужно адаптировать к нужному виду, имеет имя Adaptee. Для решения задачи преобразования его интерфейса паттерн Adapter вводит следующую иерархию классов:

- Виртуальный базовый класс Target. Здесь объявляется пользовательский интерфейс подходящего вида. Только этот интерфейс доступен для пользователя.
- Производный класс Adapter, реализующий интерфейс Target. В этом классе также имеется указатель или ссылка на экземпляр Adaptee. Паттерн Adapter использует этот указатель для перенаправления клиентских вызовов в Adaptee. Так как интерфейсы Adaptee и Target несовместимы между собой, то эти вызовы обычно требуют преобразования.

Реализация паттерна Adapter

Классическая реализация паттерна Adapter

Приведем реализацию паттерна Adapter. Для примера выше адаптируем показания температурного датчика системы климат-контроля, переведя их из градусов Фаренгейта в градусы Цельсия (предполагается, что код этого датчика недоступен для модификации).

```
#include <iostream>

// Уже существующий класс температурного датчика окружающей среды
class FahrenheitSensor
{
  public:
    // Получить показания температуры в градусах Фаренгейта
    float getFahrenheitTemp() {
      float t = 32.0;
```

```
      // ... какой то код
      return t;
    }
};

class Sensor
{
  public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;
};

class Adapter : public Sensor
{
  public:
    Adapter( FahrenheitSensor* p ) : p_fsensor(p) {
    }
   ~Adapter() {
      delete p_fsensor;
    }
    float getTemperature() {
      return (p_fsensor->getFahrenheitTemp()-32.0)*5.0/9.0;
    }
  private:
    FahrenheitSensor* p_fsensor;
};

int main()
{
  Sensor* p = new Adapter( new FahrenheitSensor);
  cout << "Celsius temperature = " << p->getTemperature() << endl;
  delete p;
  return 0;
}
```

Реализация паттерна `Adapter` на основе закрытого наследования

Пусть наш температурный датчик системы климат-контроля поддерживает функцию юстировки для получения более точных показаний. Эта функция не является обязательной для использования, возможно, поэтому соответствующий метод `adjust()` объявлен разработчиками защищенным в существующем классе `FahrenheitSensor`.

Разрабатываемая нами система должна поддерживать настройку измерений. Так как доступ к защищенному методу через указатель или ссылку запрещен, то классическая реализация паттерна `Adapter` здесь уже не подходит. Единственное решение - наследовать от класса `FahrenheitSensor`. Интерфейс этого класса должен оставаться недоступным пользователю, поэтому наследование должно быть закрытым.

Цели, преследуемые при использовании открытого и закрытого наследования различны. Если открытое наследование применяется для наследования интерфейса и реализации, то закрытое наследование - только для наследования реализации.

```
#include <iostream>

class FahrenheitSensor
{
  public:
    float getFahrenheitTemp() {
      float t = 32.0;
      // ...
      return t;
    }
  protected:
    void adjust() {} // Настройка датчика (защищенный метод)
};

class Sensor
{
  public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;
    virtual void adjust() = 0;
};

class Adapter : public Sensor, private FahrenheitSensor
{
  public:
    Adapter() { }
    float getTemperature() {
      return (getFahrenheitTemp()-32.0)*5.0/9.0;
    }
    void adjust() {
      FahrenheitSensor::adjust();
    }
};

int main()
{
  Sensor * p = new Adapter();
  p->adjust();
  cout << "Celsius temperature = " << p->getTemperature() << endl;
  delete p;
  return 0;
}
```

Результаты применения паттерна `Adapter`

Достоинства паттерна `Adapter`

- Паттерн `Adapter` позволяет повторно использовать уже имеющийся код, адаптируя его несовместимый интерфейс к виду, пригодному для использования.

Недостатки паттерна `Adapter`

- Задача преобразования интерфейсов может оказаться непростой в случае, если клиентские вызовы и (или) передаваемые параметры не имеют функционального соответствия в адаптируемом объекте.

9.3.3 Декоратор (`Decorator`)

Вы работаете над библиотекой оповещений, которую можно подключать к разнообразным программам, чтобы получать уведомления о важных событиях.

Основой библиотеки является класс `Notifier` с методом `send`, который принимает на вход строку-сообщение и высылает её всем администраторам по электронной почте. Сторонняя программа должна создать и настроить этот объект, указав кому отправлять оповещения, а затем использовать его каждый раз, когда что-то случается.

В какой-то момент стало понятно, что одних email-оповещений пользователям мало. Некоторые из них хотели бы получать извещения о критических проблемах через SMS. Другие хотели бы получать их в виде сообщений Facebook. Корпоративные пользователи хотели бы видеть сообщения в Slack.

Сначала вы добавили каждый из этих типов оповещений в программу, унаследовав их от базового класса `Notifier`. Теперь пользователь выбирал один из типов оповещений, который и использовался в дальнейшем.

Но затем кто-то резонно спросил, почему нельзя выбрать несколько типов оповещений сразу? Ведь если вдруг в вашем доме начался пожар, вы бы хотели получить оповещения по всем каналам, не так ли?

Вы попытались реализовать все возможные комбинации подклассов оповещений. Но после того, как вы добавили первый десяток классов, стало ясно, что такой подход невероятно раздувает код программы.

Наследование – это первое, что приходит в голову многим программистам, когда нужно расширить какое-то существующее поведение. Но механизм наследования имеет несколько досадных проблем.

- Он **статичен**. Вы не можете изменить поведение существующего объекта. Для этого вам надо создать новый объект, выбрав другой подкласс.
- Он **не разрешает наследовать** поведение нескольких классов одновременно. Из-за этого вам приходится создавать множество подклассов-комбинаций для получения совмещённого поведения.

Одним из способов обойти эти проблемы является замена наследования агрегацией либо композицией. Это когда один объект содержит ссылку на другой и делегирует ему работу, вместо того чтобы самому наследовать его поведение. Как раз на этом принципе построен паттерн `Декоратор`.

Рисунок 9.16

Декоратор имеет альтернативное название – *обёртка*. Оно более точно описывает суть паттерна: вы помещаете целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё.

Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать – чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно – результат будет иметь объединённое поведение всех обёрток сразу.

В примере с оповещениями мы оставим в базовом классе простую отправку по электронной почте, а расширенные способы отправки сделаем декораторами.

Сторонняя программа, выступающая клиентом, во время первичной настройки будет заворачивать объект оповещений в те обёртки, которые соответствуют желаемому способу оповещения.

Последняя обёртка в списке и будет тем объектом, с которым клиент будет работать в остальное время. Для остального клиентского кода, по сути, ничего не изменится, ведь все обёртки имеют точно такой же интерфейс, что и базовый класс оповещений.

Таким же образом можно изменять не только способ доставки оповещений, но и форматирование, список адресатов и так далее. К тому же клиент может «дообернуть» объект любыми другими обёртками, когда ему захочется.

Структура

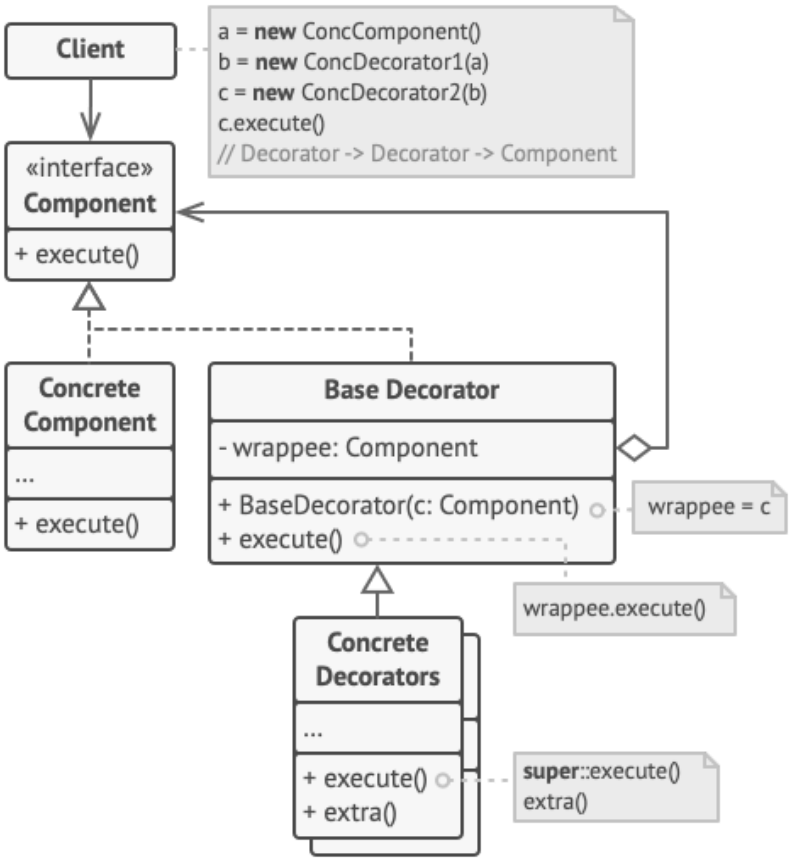


Рисунок 9.17

- **Компонент** задаёт общий интерфейс обёрток и оборачиваемых объектов.
- **Конкретный компонент** определяет класс оборачиваемых объектов. Он содержит какое-то базовое поведение, которое потом изменяют декораторы.
- **Базовый декоратор** хранит ссылку на вложенный объект-компонент. Им может быть как конкретный компонент, так и один из конкретных декораторов. Базовый декоратор делегирует все свои операции вложенному объекту. Дополнительное поведение будет жить в конкретных декораторах.
- **Конкретные декораторы** – это различные вариации декораторов, которые содержат добавочное поведение. Оно выполняется до или после вызова аналогичного поведения обёрнутого объекта.
- **Клиент** может оборачивать простые компоненты и декораторы в другие декораторы, работая со всеми объектами через общий интерфейс компонентов.

Пример использования `Декоратора`

В этом примере **`Декоратор`** защищает финансовые данные дополнительными уровнями безопасности прозрачно для кода, который их использует.

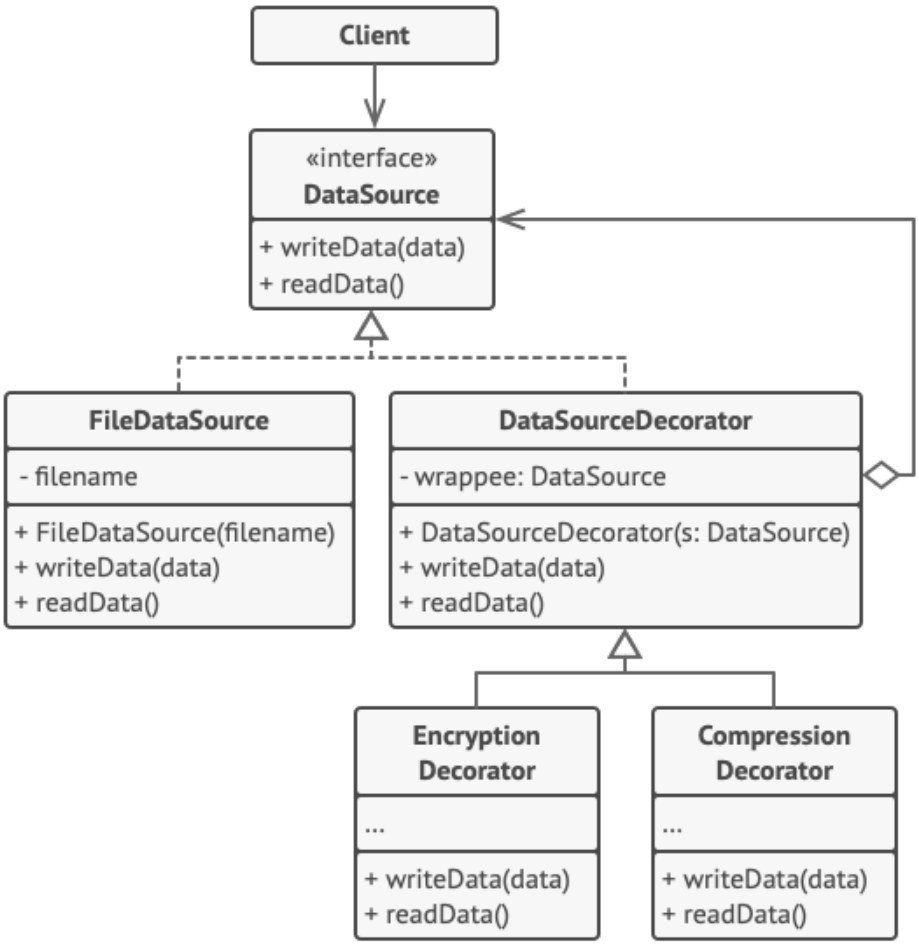


Рисунок 9.18

Пример шифрования и компрессии данных с помощью обёрток.

Приложение оборачивает класс данных в шифрующую и сжимающую обёртки, которые при чтении выдают оригинальные данные, а при записи – зашифрованные и сжатые.

Декораторы, как и сам класс данных, имеют общий интерфейс. Поэтому клиентскому коду не важно, с чем работать – с «чистым» объектом данных или с «обёрнутым».

```
// Общий интерфейс компонентов.
interface DataSource is
    method writeData(data)
    method readData():data

// Один из конкретных компонентов реализует базовую
// функциональность.
class FileDataSource implements DataSource is
    constructor FileDataSource(filename) { ... }

    method writeData(data) is
        // Записать данные в файл.

    method readData():data is
        // Прочитать данные из файла.

// Родитель всех декораторов содержит код обёртывания.
class DataSourceDecorator implements DataSource is
    protected field wrappee: DataSource

    constructor DataSourceDecorator(source: DataSource) is
        wrappee = source

    method writeData(data) is
        wrappee.writeData(data)

    method readData():data is
        return wrappee.readData()

// Конкретные декораторы добавляют что-то своё к базовому
// поведению обёрнутого компонента.
class EncryptionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Зашифровать поданные данные.
        // 2. Передать зашифрованные данные в метод writeData
        // обёрнутого объекта (wrappee).

    method readData():data is
        // 1. Получить данные из метода readData обёрнутого
        // объекта (wrappee).
        // 2. Расшифровать их, если они зашифрованы.
        // 3. Вернуть результат.
```

```
// Декорировать можно не только базовые компоненты, но и уже
// обёрнутые объекты.
class CompressionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Запаковать поданные данные.
        // 2. Передать запакованные данные в метод writeData
        // обёрнутого объекта (wrappee).

    method readData():data is
        // 1. Получить данные из метода readData обёрнутого
        // объекта (wrappee).
        // 2. Распаковать их, если они запакованы.
        // 3. Вернуть результат.


// Вариант 1. Простой пример сборки и использования декораторов.
class Application is
    method dumbUsageExample() is
        source = new FileDataSource("somefile.dat")
        source.writeData(salaryRecords)
        // В файл были записаны чистые данные.

        source = new CompressionDecorator(source)
        source.writeData(salaryRecords)
        // В файл были записаны сжатые данные.

        source = new EncryptionDecorator(source)
        // Сейчас в source находится связка из трёх объектов:
        // Encryption > Compression > FileDataSource

        source.writeData(salaryRecords)
        // В файл были записаны сжатые и зашифрованные данные.


// Вариант 2. Клиентский код, использующий внешний источник
// данных. Класс SalaryManager ничего не знает о том, как именно
// будут считаны и записаны данные. Он получает уже готовый
// источник данных.
class SalaryManager is
    field source: DataSource

    constructor SalaryManager(source: DataSource) { ... }

    method load() is
        return source.readData()

    method save() is
        source.writeData(salaryRecords)
    // ...Остальные полезные методы...
```

```
// Приложение может по-разному собирать декорируемые объекты, в
// зависимости от условий использования.
class ApplicationConfigurator is
    method configurationExample() is
        source = new FileDataSource("salary.dat")
        if (enabledEncryption)
            source = new EncryptionDecorator(source)
        if (enabledCompression)
            source = new CompressionDecorator(source)

        logger = new SalaryManager(source)
        salary = logger.load()
    // ...
```

Шаги реализации

1. Убедитесь, что в вашей задаче есть один основной компонент и несколько опциональных дополнений или надстроек над ним.
2. Создайте интерфейс компонента, который описывал бы общие методы как для основного компонента, так и для его дополнений.
3. Создайте класс конкретного компонента и поместите в него основную бизнес-логику.
4. Создайте базовый класс декораторов. Он должен иметь поле для хранения ссылки на вложенный объект-компонент. Все методы базового декоратора должны делегировать действие вложенному объекту.
5. И конкретный компонент, и базовый декоратор должны следовать одному и тому же интерфейсу компонента.
6. Теперь создайте классы конкретных декораторов, наследуя их от базового декоратора. Конкретный декоратор должен выполнять свою добавочную функцию, а затем (или перед этим) вызывать эту же операцию обёрнутого объекта.
7. Клиент берёт на себя ответственность за конфигурацию и порядок обёртывания объектов.

Преимущества паттерна `Декоратор`

- Большая гибкость, чем у наследования.
- Позволяет добавлять обязанности на лету.
- Можно добавлять несколько новых обязанностей сразу.
- Позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.

Недостатки паттерна `Декоратор`

- Трудно конфигурировать многократно обёрнутые объекты.
- Обилие крошечных классов.

9.3.4 Компоновщик (`Composite`)

Паттерн Компоновщик имеет смысл только тогда, когда основная модель вашей программы может быть структурирована в виде дерева.

Например, есть два объекта: `Продукт` и `Коробка`. `Коробка` может содержать несколько `Продуктов` и других `Коробок` поменьше. Те, в свою очередь, тоже содержат либо `Продукты`, либо `Коробки` и так далее.

Теперь предположим, ваши `Продукты` и `Коробки` могут быть частью заказов. Каждый заказ может содержать как простые `Продукты` без упаковки, так и составные `Коробки`. Ваша задача состоит в том, чтобы узнать цену всего заказа.

Если решать задачу в лоб, то вам потребуется открыть все коробки заказа, перебрать все продукты и посчитать их суммарную стоимость. Но это слишком хлопотно, так как типы коробок и их содержимое могут быть вам неизвестны. Кроме того, наперёд неизвестно и количество уровней вложенности коробок, поэтому перебрать коробки простым циклом не выйдет.

Компоновщик предлагает рассматривать `Продукт` и `Коробку` через единый интерфейс с общим методом получения стоимости.

`Продукт` просто вернёт свою цену. `Коробка` спросит цену каждого предмета внутри себя и вернёт сумму результатов. Если одним из внутренних предметов окажется коробка поменьше, она тоже будет перебирать своё содержимое, и так далее, пока не будут посчитаны все составные части.

Для вас, клиента, главное, что теперь не нужно ничего знать о структуре заказов. Вы вызываете метод получения цены, он возвращает цифру, а вы не тонете в горах картона и скотча.

Структура

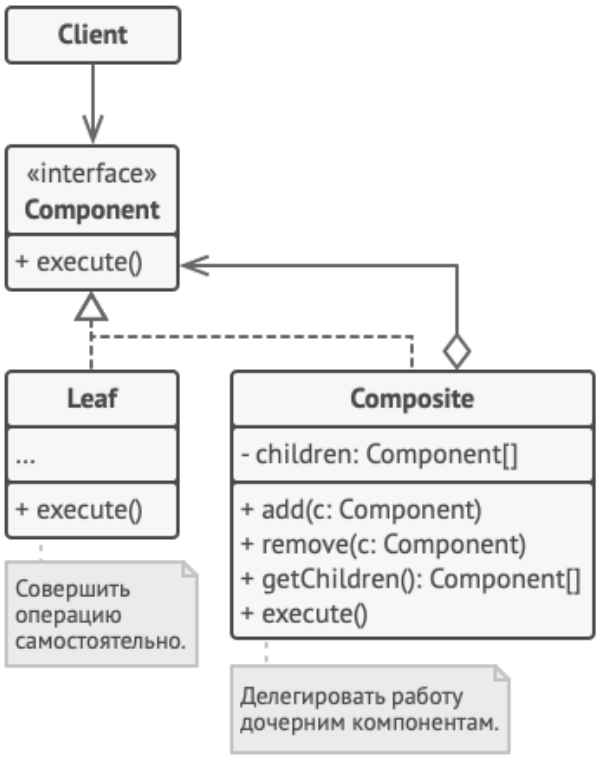


Рисунок 9.19

1. **Компонент** определяет общий интерфейс для простых и составных компонентов дерева.

2. **Лист** – это простой компонент дерева, не имеющий ответвлений. Из-за того, что им некому больше передавать выполнение, классы листьев будут содержать большую часть полезного кода.

3. **Контейнер (или композит)** – это составной компонент дерева. Он содержит набор дочерних компонентов, но ничего не знает об их типах. Это могут быть как простые компоненты-листья, так и другие компоненты-контейнеры. Но это не является проблемой, если все дочерние компоненты следуют единому интерфейсу.

Методы контейнера переадресуют основную работу своим дочерним компонентам, хотя и могут добавлять что-то своё к результату.

4. Клиент работает с деревом через общий интерфейс компонентов. Благодаря этому, клиенту не важно, что перед ним находится – простой или составной компонент дерева.

Пример использования `Компоновщика`

В этом примере Компоновщик помогает реализовать вложенные геометрические фигуры.

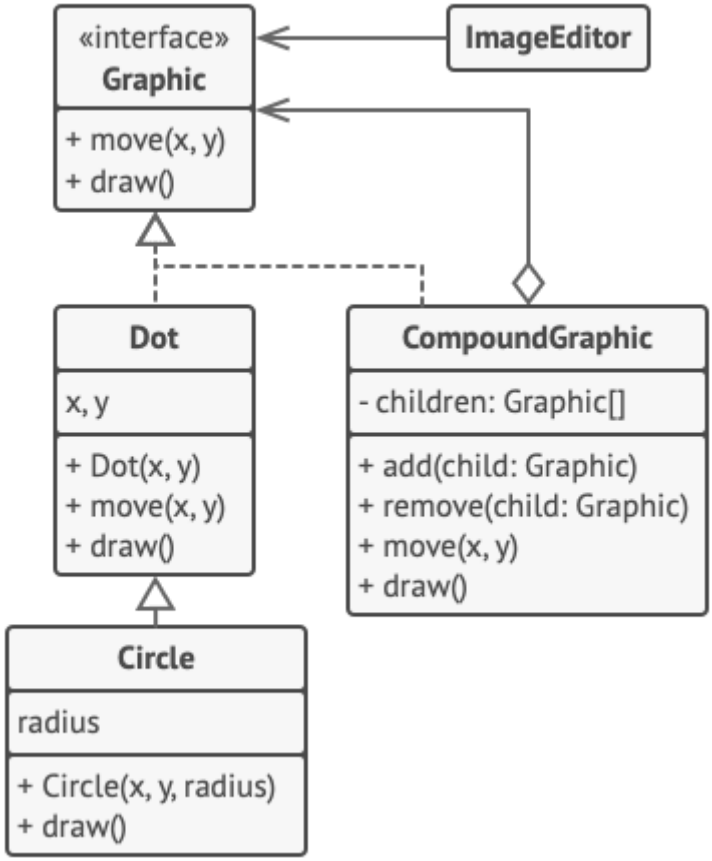


Рисунок 9.20

Класс `CompoundGraphic` может содержать любое количество подфигур, включая такие же контейнеры, как он сам. Контейнер реализует те же методы, что и простые фигуры. Но, вместо непосредственного действия, он передаёт вызовы всем

вложенным компонентам, используя рекурсию. Затем он как бы «суммирует» результаты всех вложенных фигур.

Клиентский код работает со всеми фигурами через общий интерфейс фигур и не знает, что перед ним – простая фигура или составная. Это позволяет клиентскому коду работать с деревьями объектов любой сложности, не привязываясь к конкретным классам объектов, формирующих дерево.

```
// Общий интерфейс компонентов.
interface Graphic is
    method move(x, y)
    method draw()

// Простой компонент.
class Dot implements Graphic is
    field x, y

    constructor Dot(x, y) { ... }

    method move(x, y) is
        this.x += x, this.y += y

    method draw() is
        // Нарисовать точку в координате X, Y.

// Компоненты могут расширять другие компоненты.
class Circle extends Dot is
    field radius

    constructor Circle(x, y, radius) { ... }

    method draw() is
        // Нарисовать окружность в координате X, Y и радиусом R.

// Контейнер содержит операции добавления/удаления дочерних
// компонентов. Все стандартные операции интерфейса компонентов
// он делегирует каждому из дочерних компонентов.
class CompoundGraphic implements Graphic is
    field children: array of Graphic

    method add(child: Graphic) is
        // Добавить компонент в список дочерних.

    method remove(child: Graphic) is
        // Убрать компонент из списка дочерних.

    method move(x, y) is
        foreach (child in children) do
            child.move(x, y)

    method draw() is
```

```
        // 1. Для каждого дочернего компонента:
        //      - Отрисовать компонент.
        //      - Определить координаты максимальной границы.
        // 2. Нарисовать пунктирную границу вокруг всей области.


// Приложение работает единообразно как с единичными
// компонентами, так и с целыми группами компонентов.
class ImageEditor is
    field all: CompoundGraphic

    method load() is
        all = new CompoundGraphic()
        all.add(new Dot(1, 2))
        all.add(new Circle(5, 3, 10))
        // ...

    // Группировка выбранных компонентов в один сложный
    // компонент.
    method groupSelected(components: array of Graphic) is
        group = new CompoundGraphic()
        foreach (component in components) do
            group.add(component)
            all.remove(component)
        all.add(group)
        // Все компоненты будут отрисованы.
        all.draw()
```

Шаги реализации

1. Убедитесь, что вашу бизнес-логику можно представить как древовидную структуру. Попытайтесь разбить её на простые компоненты и контейнеры. Помните, что контейнеры могут содержать как простые компоненты, так и другие вложенные контейнеры.
2. Создайте общий интерфейс компонентов, который объединит операции контейнеров и простых компонентов дерева. Интерфейс будет удачным, если вы сможете использовать его, чтобы взаимозаменять простые и составные компоненты без потери смысла.
3. Создайте класс компонентов-листьев, не имеющих дальнейших ответвлений. Имейте в виду, что программа может содержать несколько таких классов.
4. Создайте класс компонентов-контейнеров и добавьте в него массив для хранения ссылок на вложенные компоненты. Этот массив должен быть способен содержать как простые, так и составные компоненты, поэтому убедитесь, что он объявлен с типом интерфейса компонентов.
5. Реализуйте в контейнере методы интерфейса компонентов, помня о том, что контейнеры должны делегировать основную работу своим дочерним компонентам.
6. Добавьте операции добавления и удаления дочерних компонентов в класс контейнеров.

7. Имейте в виду, что методы добавления/удаления дочерних компонентов можно поместить и в интерфейс компонентов. Да, это нарушит *принцип разделения интерфейса*, так как реализации методов будут пустыми в компонентах-листьях. Но зато все компоненты дерева станут действительно одинаковыми для клиента.

Преимущества паттерна `Компоновщик`

1. Упрощает архитектуру клиента при работе со сложным деревом компонентов.
2. Облегчает добавление новых видов компонентов.

Недостатки паттерна `Компоновщик`

1. Создаёт слишком общий дизайн классов.

9.3.5 Заместитель (Proxy)

Для чего вообще контролировать доступ к объектам? Рассмотрим такой пример: у вас есть внешний ресурсоёмкий объект, который нужен не все время, а изредка.

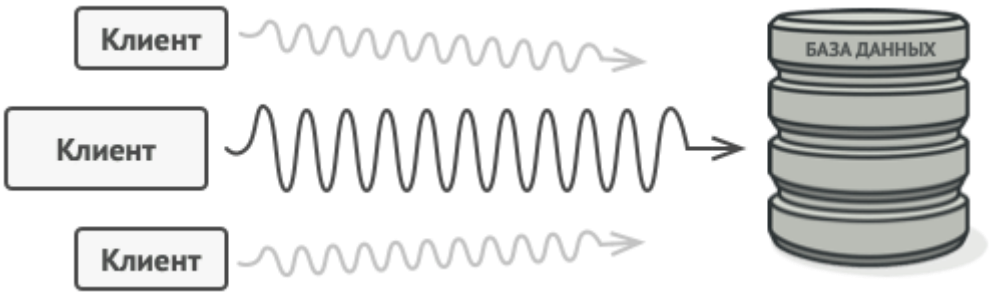


Рисунок 9.21

Мы могли бы создавать этот объект не в самом начале программы, а только тогда, когда он кому-то реально понадобится. Каждый клиент объекта получил бы некий код отложенной инициализации. Но, вероятно, это привело бы к множественному дублированию кода.

В идеале, этот код хотелось бы поместить прямо в служебный класс, но это не всегда возможно. Например, код класса может находиться в закрытой сторонней библиотеке.

Паттерн Заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.

Рисунок 9.22

Но в чём же здесь польза? Вы могли бы поместить в класс заместителя какую-то промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте. А благодаря одинаковому интерфейсу, объект-заместитель можно передать в любой код, ожидающий сервисный объект.

Структура

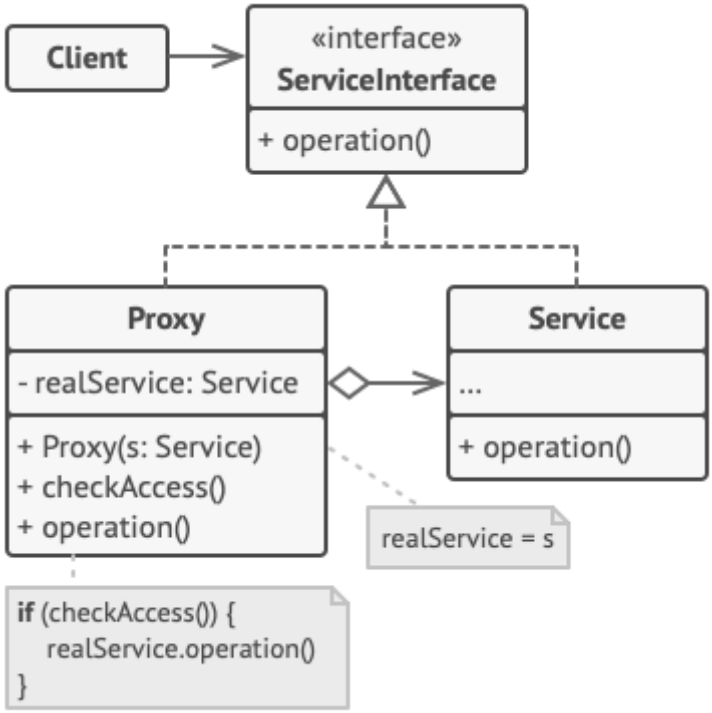


Рисунок 9.23

1. **Интерфейс сервиса** определяет общий интерфейс для сервиса и заместителя. Благодаря этому, объект заместителя можно использовать там, где ожидается объект сервиса.
2. **Сервис** содержит полезную бизнес-логику.
3. **Заместитель** хранит ссылку на объект сервиса. После того как заместитель заканчивает свою работу (например, инициализацию, логирование, защиту или другое), он передаёт вызовы вложенному сервису. Заместитель может сам отвечать за создание и удаление объекта сервиса.
4. **Клиент** работает с объектами через интерфейс сервиса. Благодаря этому, его можно «одурачить», подменив объект сервиса объектом заместителя.

Пример использования паттерна `Proxy`

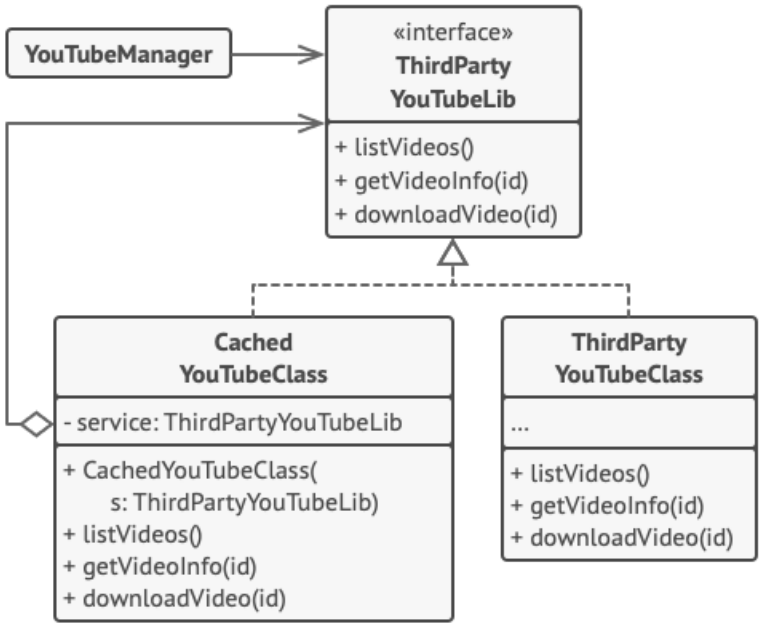


Рисунок 9.24

Оригинальный объект начинал загрузку по сети, даже если пользователь запрашивал одно и то же видео. Заместитель же загружает видео только один раз, используя для этого служебный объект, но в остальных случаях возвращает закешированный файл.

```
// Интерфейс удалённого сервиса.
interface ThirdPartyYouTubeLib is
    method listVideos()
    method getVideoInfo(id)
    method downloadVideo(id)

// Конкретная реализация сервиса. Методы этого класса
// запрашивают у YouTube различную информацию. Скорость запроса
// зависит не только от качества интернет-канала пользователя,
// но и от состояния самого YouTube. Значит, чем больше будет
// вызовов к сервису, тем менее отзывчивой станет программа.
class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib is
    method listVideos() is
        // Получить список видеороликов с помощью API YouTube.

    method getVideoInfo(id) is
        // Получить детальную информацию о каком-то видеоролике.

    method downloadVideo(id) is
        // Скачать видео с YouTube.

// С другой стороны, можно кешировать запросы к YouTube и не
// повторять их какое-то время, пока кеш не устареет. Но внести
// этот код напрямую в сервисный класс нельзя, так как он
```

```
// находится в сторонней библиотеке. Поэтому мы поместим логику
// кеширования в отдельный класс-обёртку. Он будет делегировать
// запросы к сервисному объекту, только если нужно
// непосредственно выслать запрос.
class CachedYouTubeClass implements ThirdPartyYouTubeLib is
    private field service: ThirdPartyYouTubeLib
    private field listCache, videoCache
    field needReset

    constructor CachedYouTubeClass(service: ThirdPartyYouTubeLib) is
        this.service = service

    method listVideos() is
        if (listCache == null || needReset)
            listCache = service.listVideos()
        return listCache

    method getVideoInfo(id) is
        if (videoCache == null || needReset)
            videoCache = service.getVideoInfo(id)
        return videoCache

    method downloadVideo(id) is
        if (!downloadExists(id) || needReset)
            service.downloadVideo(id)

// Класс GUI, который использует сервисный объект. Вместо
// реального сервиса, мы подсунем ему объект-заместитель. Клиент
// ничего не заметит, так как заместитель имеет тот же
// интерфейс, что и сервис.
class YouTubeManager is
    protected field service: ThirdPartyYouTubeLib

    constructor YouTubeManager(service: ThirdPartyYouTubeLib) is
        this.service = service

    method renderVideoPage(id) is
        info = service.getVideoInfo(id)
        // Отобразить страницу видеоролика.

    method renderListPanel() is
        list = service.listVideos()
        // Отобразить список превьюшек видеороликов.

    method reactOnUserInput() is
        renderVideoPage()
        renderListPanel()

// Конфигурационная часть приложения создаёт и передаёт клиентам
// объект заместителя.
class Application is
```

```
method init() is
    YouTubeService = new ThirdPartyYouTubeClass()
    YouTubeProxy = new CachedYouTubeClass(YouTubeService)
    manager = new YouTubeManager(YouTubeProxy)
    manager.reactOnUserInput()
```

Шаги реализации

1. Определите интерфейс, который бы сделал заместитель и оригинальный объект взаимозаменяемыми.
2. Создайте класс заместителя. Он должен содержать ссылку на сервисный объект. Чаще всего, сервисный объект создаётся самим заместителем. В редких случаях заместитель получает готовый сервисный объект от клиента через конструктор.
3. Реализуйте методы заместителя в зависимости от его предназначения. В большинстве случаев, проделав какую-то полезную работу, методы заместителя должны передать запрос сервисному объекту.
4. Подумайте о введении фабрики, которая решала бы, какой из объектов создавать – заместитель или реальный сервисный объект. Но, с другой стороны, эта логика может быть помещена в создающий метод самого заместителя.
5. Подумайте, не реализовать ли вам ленивую инициализацию сервисного объекта при первом обращении клиента к методам заместителя.

Преимущества паттерна `Proxy`

- Позволяет контролировать сервисный объект незаметно для клиента.
- Может работать, даже если сервисный объект ещё не создан.
- Может контролировать жизненный цикл служебного объекта.

Недостатки паттерна `Proxy`

- Усложняет код программы из-за введения дополнительных классов.
- Увеличивает время отклика от сервиса

9.4 Поведенческие шаблоны проектирования (`Behavioral`)

Это группа шаблонов для определения связи и взаимодействия между объектами и классами, повышения гибкости, переиспользуемости, сопровождаемости за счет инкапсуляции поведений в отдельные объекты.

- **Итератор (`iterator`).** Это интерфейс, который даёт доступ к элементам коллекции (массива или контейнера), а также навигацию по ним. В зависимости от конкретной системы итераторы носят разные названия. Если говорить о системах управления, то это курсоры.
- **Интерпретатор (`Interpreter`).** Он решает одну очень известную, но постоянно меняющуюся задачу. Альтернативное название — Little (Small) Language.
- **Цепочка обязанностей (`Chain of responsibility`).** Такой шаблон нужен для организации уровней ответственности в системе.
- **Хранитель (`Memento`).** С его помощью можно выполнить закрепление и сохранение внутреннего состояния объекта без нарушения инкапсуляции. Это

нужно для того, чтобы в последующем можно было восстановить его в это самое состояние.

- **Команда (`Command`).** Он представляет действие и применяется в рамках объектно-ориентированного программирования. Объект команды содержит в себе как само действие, так и его параметры.
- **Посредник (`Mediator`).** Данный паттерн проектирования позволяет обеспечить взаимодействие нескольких объектов. В процессе этого создаётся слабая связанность и объекты избавляются от потребности в явных ссылках друг на друга.
- **Стратегия (Strategy)** Этим шаблоном инкапсулируются взаимозаменяемые поведения – стратегия или алгоритм – и для определения поведения, применяемого во время выполнения, используется делегирование. «Стратегия» основана на принципе открытости/закрытости: как писать расширяемый код, не трогая уже имеющегося.

Остановимся подробнее на некоторых из них.

9.4.1 Паттерн Memento (хранитель)

Назначение паттерна `Memento`

- Не нарушая инкапсуляции, паттерн `Memento` получает и сохраняет за пределами объекта его внутреннее состояние так, чтобы позже можно было восстановить объект в таком же состоянии.
- Является средством для инкапсуляции «контрольных точек» программы.
- Паттерн `Memento` придает операциям «Отмена» (`undo`) или «Откат» (`rollback`) статус «полноценного объекта».

Решаемая проблема

Вам нужно восстановить объект обратно в прежнее состояние (те есть выполнить операции «Отмена» или «Откат»).

Обсуждение паттерна `Memento`

Клиент запрашивает `Memento` (хранителя) у исходного объекта, когда ему необходимо сохранить состояние исходного объекта (установить контрольную точку). Исходный объект инициализирует `Memento` своим текущим состоянием. Клиент является «посыльным» за `Memento`, но только исходный объект может сохранять и извлекать информацию из `Memento` (`Memento` является «непрозрачным» для клиентов и других объектов). Если клиенту в дальнейшем нужно «откатить» состояние исходного объекта, он передает `Memento` обратно в исходный объект для его восстановления.

Реализовать возможность выполнения неограниченного числа операций «Отмена» (`undo`) и «Повтор» (`redo`) можно с помощью стека объектов `Command` и стека объектов `Memento`.

Паттерн проектирования `Memento` определяет трех различных участников:

- **`Originator (хозяин)`** - объект, умеющий создавать хранителя, а также знающий, как восстановить свое внутреннее состояние из хранителя.

- **Caretaker (смотритель)** - объект, который знает, почему и когда хозяин должен сохранять и восстанавливать себя.
- **Memento (хранитель)** – «ящик на замке», который пишется и читается хозяином и за которым присматривает смотритель.

Структура паттерна Memento

UML-диаграмма классов паттерна Memento

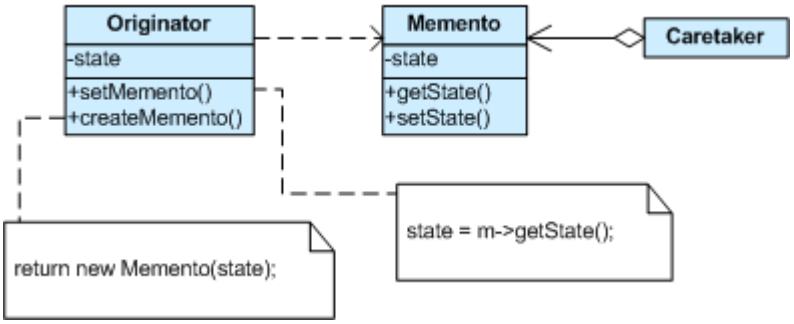


Рисунок 9.25

Пример паттерна Memento

Паттерн Memento фиксирует и сохраняет за пределами объекта его внутреннее состояние так, чтобы позже этот объект можно было бы восстановить в таком же состоянии. Этот паттерн часто используется механиками-любителями для ремонта барабанных тормозов на своих автомобилях. Барабаны удаляются с обеих сторон, чтобы сделать видимыми правые и левые тормоза. При этом разбирается только одна сторона, другая же служит напоминанием (Memento) о том, как части тормозной системы тормозной системы собраны вместе. Только после того, как завершена работа с одной стороны, разбирается другая сторона. При этом в качестве Memento выступает уже первая сторона.

Использование паттерна Memento

- Определите роли «смотрителя» и «хозяина».
- Создайте класс Memento и объявите хозяина другом.
- Смотритель знает, когда создавать «контрольную точку» хозяина.
- Хозяин создает хранителя Memento и копирует свое состояние в этот Memento.
- Смотритель сохраняет хранителя Memento (но смотритель не может заглянуть в Memento).
- Смотритель знает, когда нужно «откатить» хозяина.
- Хозяин восстанавливает себя, используя сохраненное в Memento состояние.

Особенности паттерна Memento

- Паттерны Command и Memento определяют объекты «волшебная палочка», которые передаются от одного владельца к другому и используются позднее. В Command такой «волшебной палочкой» является запрос; в Memento – внутреннее состояние объекта в некоторый момент времени. Полиморфизм важен для

Command, но не важен для Memento потому, что интерфейс Memento настолько «узкий», что его можно передавать как значение.

- Command может использовать Memento для сохранения состояния, необходимого для выполнения отмены действий.
- Memento часто используется совместно с Iterator. Iterator может использовать Memento для сохранения состояния итерации.

Реализация паттерна Memento

Memento – это объект, хранящий «снимок» внутреннего состояния другого объекта. Memento может использоваться для поддержки «многоуровневой» отмены действий паттерна Command. В этом примере перед выполнением команды по изменению объекта Number, текущее состояние этого объекта сохраняется в статическом списке истории хранителей Memento, а сама команда сохраняется в статическом списке истории команд. Undo() просто восстанавливает состояние объекта Number, получаемое из списка истории хранителей. Redo() использует список истории команд. Обратите внимание, Memento "открыт" для Number.

```
#include <iostream.h>
class Number;

class Memento
{
  public:
    Memento(int val)
    {
        _state = val;
    }
  private:
    friend class Number;
    int _state;
};

class Number
{
  public:
    Number(int value)
    {
        _value = value;
    }
    void dubble()
    {
        _value = 2 * _value;
    }
    void half()
    {
        _value = _value / 2;
    }
    int getValue()
    {
```

```
            return _value;
        }
        Memento *createMemento()
        {
            return new Memento(_value);
        }
        void reinstateMemento(Memento *mem)
        {
            _value = mem->_state;
        }
      private:
        int _value;
    };

    class Command
    {
      public:
        typedef void(Number:: *Action)();
        Command(Number *receiver, Action action)
        {
            _receiver = receiver;
            _action = action;
        }
        virtual void execute()
        {
            _mementoList[_numCommands]    =    _receiver-
>createMemento();
            _commandList[_numCommands] = this;
            if (_numCommands > _highWater)
              _highWater = _numCommands;
            _numCommands++;
            (_receiver-> *_action)();
        }
        static void undo()
        {
            if (_numCommands == 0)
            {
                cout << "*** Attempt to run off the end!!
***" << endl;
                return ;
            }
               _commandList[_numCommands - 1]->_receiver-
>reinstateMemento
              (_mementoList[_numCommands - 1]);
            _numCommands--;
        }
        void static redo()
        {
            if (_numCommands > _highWater)
            {
```

```
                cout << "*** Attempt to run off the end!!
***" << endl;
                return ;
            }
       (_commandList[_numCommands]->_receiver-
>*(_commandList[_numCommands]
              ->_action))();
            _numCommands++;
        }
      protected:
        Number *_receiver;
        Action _action;
        static Command *_commandList[20];
        static Memento *_mementoList[20];
        static int _numCommands;
        static int _highWater;
    };

    Command *Command::_commandList[];
    Memento *Command::_mementoList[];
    int Command::_numCommands = 0;
    int Command::_highWater = 0;

    int main()
    {
      int i;
      cout << "Integer: ";
      cin >> i;
      Number *object = new Number(i);

      Command *commands[3];
      commands[1] = new Command(object, &Number::dubble);
      commands[2] = new Command(object, &Number::half);

      cout << "Exit[0], Double[1], Half[2], Undo[3],
Redo[4]: ";
      cin >> i;

      while (i)
      {
        if (i == 3)
          Command::undo();
        else if (i == 4)
          Command::redo();
        else
          commands[i]->execute();
        cout << "   " << object->getValue() << endl;
        cout << "Exit[0], Double[1], Half[2], Undo[3],
Redo[4]: ";
        cin >> i;
      }
```

```
}
```

9.4.2 Итератор (`Iterator`)

Коллекции — самая распространённая структура данных, которую вы можете встретить в программировании. Это набор объектов, собранный в одну кучу по каким-то критериям.

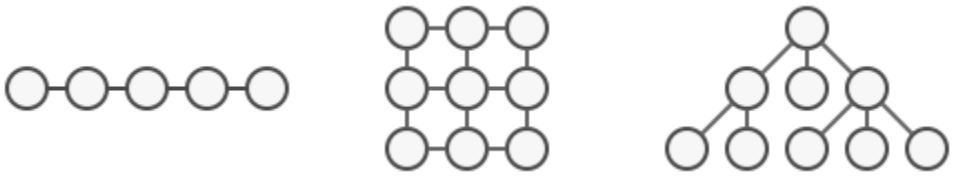


Рисунок 9.26

Большинство коллекций выглядят как обычный список элементов. Но есть и экзотические коллекции, построенные на основе деревьев, графов и других сложных структур данных.

Но как бы ни была структурирована коллекция, пользователь должен иметь возможность последовательно обходить её элементы, чтобы проделывать с ними какие-то действия.

Но каким способом следует перемещаться по сложной структуре данных? Например, сегодня может быть достаточным обход дерева в глубину, но завтра потребуется возможность перемещаться по дереву в ширину. А на следующей неделе и того хуже – понадобится обход коллекции в случайном порядке.

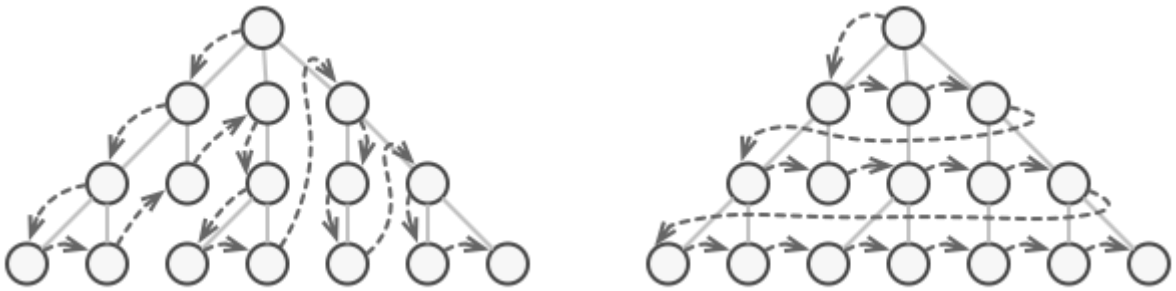


Рисунок 9.27

Добавляя всё новые алгоритмы в код коллекции, вы понемногу размываете её основную задачу, которая заключается в эффективном хранении данных. Некоторые алгоритмы могут быть и вовсе слишком «заточены» под определённое приложение и смотреться дико в общем классе коллекции.

Идея паттерна `Итератор` состоит в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс.

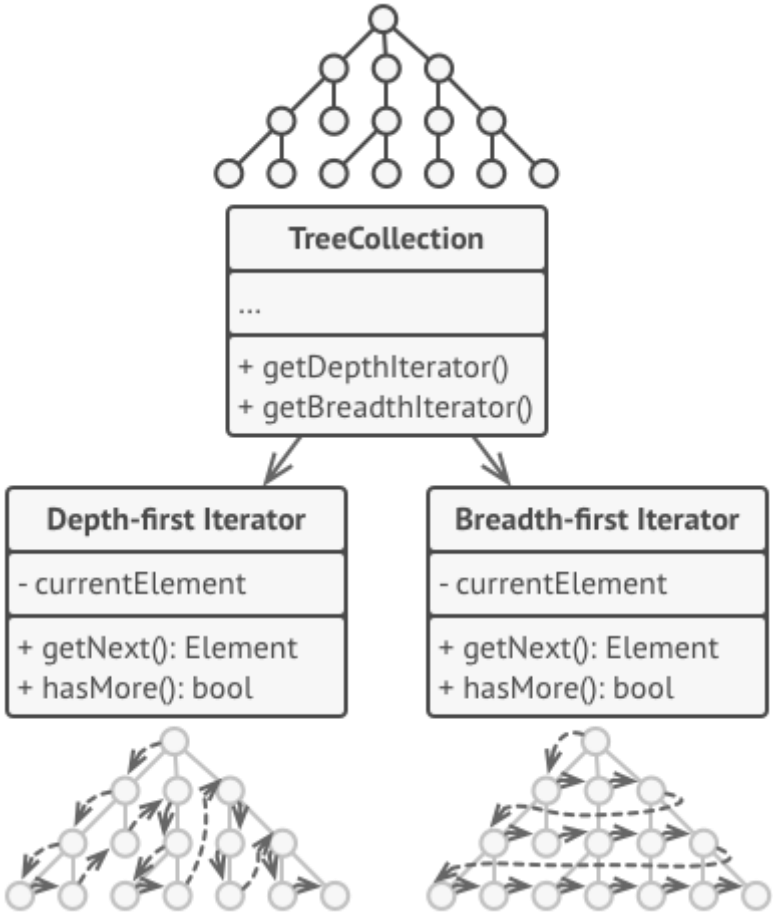


Рисунок 9.28

Итераторы содержат код обхода коллекции. Одну коллекцию могут обходить сразу несколько итераторов.

Объект-итератор будет отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти. Одну и ту же коллекцию смогут одновременно обходить различные итераторы, а сама коллекция не будет даже знать об этом.

К тому же, если вам понадобится добавить новый способ обхода, вы сможете создать отдельный класс итератора, не изменяя существующий код коллекции.

Структура

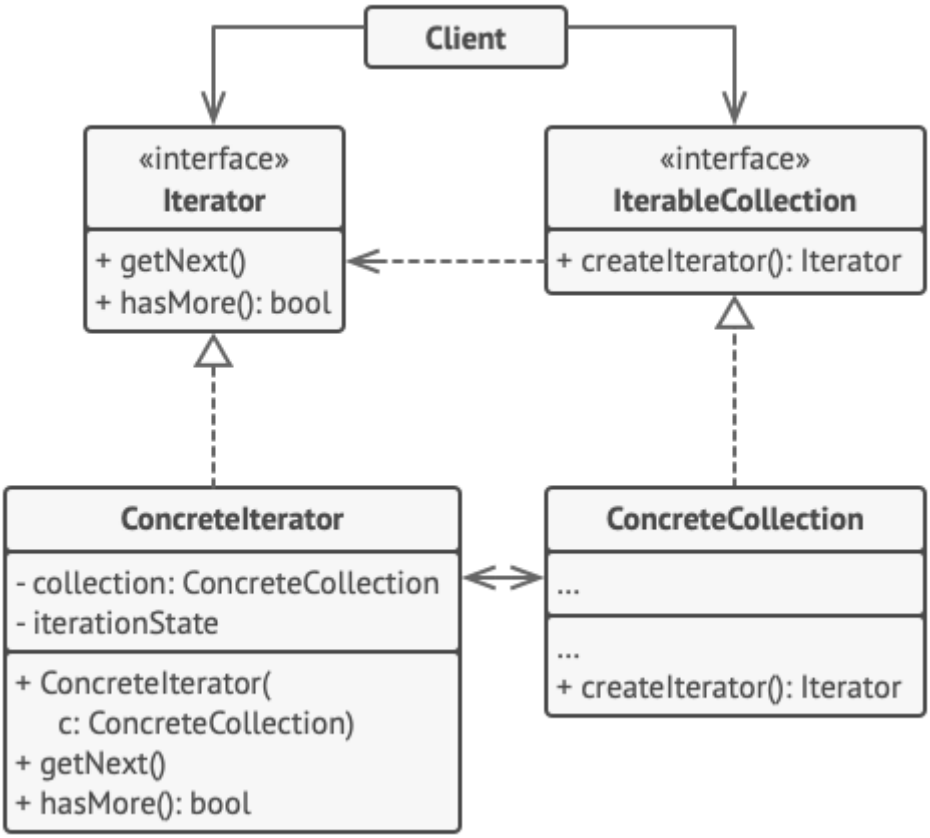


Рисунок 9.29

1. **Итератор** описывает интерфейс для доступа и обхода элементов коллекции.
2. **Конкретный итератор** реализует алгоритм обхода какой-то конкретной коллекции. Объект итератора должен сам отслеживать текущую позицию при обходе коллекции, чтобы отдельные итераторы могли обходить одну и ту же коллекцию независимо.
3. **Коллекция** описывает интерфейс получения итератора из коллекции. Как мы уже говорили, коллекции не всегда являются списком. Это может быть и база данных, и удалённое API, и даже дерево **Компоновщика**. Поэтому сама коллекция может создавать итераторы, так как она знает, какие именно итераторы способны с ней работать.
4. **Конкретная коллекция** возвращает новый экземпляр определённого конкретного итератора, связав его с текущим объектом коллекции. Обратите внимание, что сигнатура метода возвращает интерфейс итератора. Это позволяет клиенту не зависеть от конкретных классов итераторов.
5. **Клиент** работает со всеми объектами через интерфейсы коллекции и итератора. Так клиентский код не зависит от конкретных классов, что позволяет применять различные итераторы, не изменяя существующий код программы. В общем случае клиенты не создают объекты итераторов, а получают их из коллекций. Тем не

менее, если клиенту требуется специальный итератор, он всегда может создать его самостоятельно.

Пример использования паттерна `Итератор`

В этом примере паттерн **Итератор** используется для реализации обхода нестандартной коллекции, которая инкапсулирует доступ к социальному графу Facebook. Коллекция предоставляет несколько итераторов, которые могут по-разному обходить профили людей.

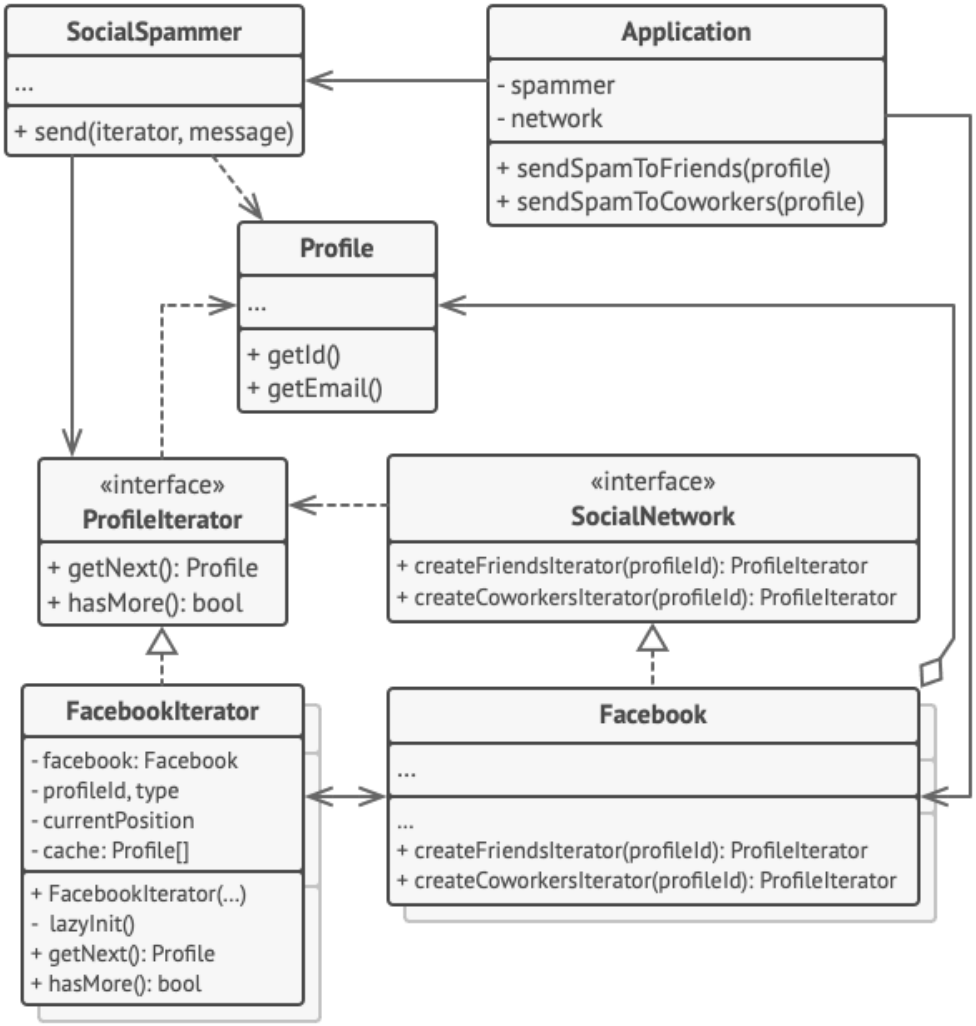


Рисунок 9.30

Так, итератор друзей перебирает всех друзей профиля, а итератор коллег – фильтрует друзей по принадлежности к компании профиля. Все итераторы реализуют общий интерфейс, который позволяет клиентам работать с профилями, не вникая в детали работы с социальной сетью (например, в авторизацию, отправку REST-запросов и т. д.)

Кроме того, Итератор избавляет код от привязки к конкретным классам коллекций. Это позволяет добавить поддержку другого вида коллекций (например, LinkedIn), не меняя клиентский код, который работает с итераторами и коллекциями.

```
// Общий интерфейс коллекций должен определить фабричный метод
```

```
// для производства итератора. Можно определить сразу несколько
// методов, чтобы дать пользователям различные варианты обхода
// одной и той же коллекции.
interface SocialNetwork is
    method createFriendsIterator(profileId):ProfileIterator
    method createCoworkersIterator(profileId):ProfileIterator

// Конкретная коллекция знает, объекты каких итераторов нужно
// создавать.
class Facebook implements SocialNetwork is
    // ...Основной код коллекции...

    // Код получения нужного итератора.
    method createFriendsIterator(profileId) is
        return new FacebookIterator(this, profileId, "friends")
    method createCoworkersIterator(profileId) is
        return new FacebookIterator(this, profileId, "coworkers")

// Общий интерфейс итераторов.
interface ProfileIterator is
    method getNext():Profile
    method hasMore():bool

// Конкретный итератор.
class FacebookIterator implements ProfileIterator is
    // Итератору нужна ссылка на коллекцию, которую он обходит.
    private field facebook: Facebook
    private field profileId, type: string

    // Но каждый итератор обходит коллекцию, независимо от
    // остальных, поэтому он содержит информацию о текущей
    // позиции обхода.
    private field currentPosition
    private field cache: array of Profile

    constructor FacebookIterator(facebook, profileId, type) is
        this.facebook = facebook
        this.profileId = profileId
        this.type = type

    private method lazyInit() is
        if (cache == null)
            cache = facebook.socialGraphRequest(profileId, type)

    // Итератор реализует методы базового интерфейса по-своему.
    method getNext() is
        if (hasMore())
            result = cache[currentPosition]
            currentPosition++
            return result
```

```
    method hasMore() is
        lazyInit()
        return currentPosition < cache.length

// Вот ещё полезная тактика: мы можем передавать объект
// итератора вместо коллекции в клиентские классы. При таком
// подходе клиентский код не будет иметь доступа к коллекциям, а
// значит, его не будут волновать подробности их реализаций. Ему
// будет доступен только общий интерфейс итераторов.
class SocialSpammer is
    method send(iterator: ProfileIterator, message: string) is
        while (iterator.hasMore())
            profile = iterator.getNext()
            System.sendEmail(profile.getEmail(), message)


// Класс приложение конфигурирует классы, как захочет.
class Application is
    field network: SocialNetwork
    field spammer: SocialSpammer

    method config() is
        if working with Facebook
            this.network = new Facebook()
        if working with LinkedIn
            this.network = new LinkedIn()
        this.spammer = new SocialSpammer()

    method sendSpamToFriends(profile) is
        iterator = network.createFriendsIterator(profile.getId())
        spammer.send(iterator, "Very important message")

    method sendSpamToCoworkers(profile) is
        iterator = network.createCoworkersIterator(profile.getId())
        spammer.send(iterator, "Very important message")
```

Шаги реализации

1. Создайте общий интерфейс итераторов. Обязательный минимум – это операция получения следующего элемента коллекции. Но для удобства можно предусмотреть и другое. Например, методы для получения предыдущего элемента, текущей позиции, проверки окончания обхода и прочие.
2. Создайте интерфейс коллекции и опишите в нём метод получения итератора. Важно, чтобы сигнатура метода возвращала общий интерфейс итераторов, а не один из конкретных итераторов.
3. Создайте классы конкретных итераторов для тех коллекций, которые нужно обходить с помощью паттерна. Итератор должен быть привязан только к одному объекту коллекции. Обычно эта связь устанавливается через конструктор.
4. Реализуйте методы получения итератора в конкретных классах коллекций. Они должны создавать новый итератор того класса, который способен работать с

данным типом коллекции. Коллекция должна передавать ссылку на собственный объект в конструктор итератора.

5. В клиентском коде и в классах коллекций не должно остаться кода обхода элементов. Клиент должен получать новый итератор из объекта коллекции каждый раз, когда ему нужно перебрать её элементы.

Преимущества паттерна `Итератор`

- Упрощает классы хранения данных.
- Позволяет реализовать различные способы обхода структуры данных.
- Позволяет одновременно перемещаться по структуре данных в разные стороны.

Недостатки паттерна `Итератор`

- Не оправдан, если можно обойтись простым циклом.

9.4.3 Паттерн Command (команда)

Назначение паттерна `Command`

Используйте паттерн `Command` если

- Система управляется событиями. При появлении такого события (запроса) необходимо выполнить определенную последовательность действий.
- Необходимо параметризировать объекты выполняемым действием, ставить запросы в очередь или поддерживать операции отмены (`undo`) и повтора (`redo`) действий.
- Нужен объектно-ориентированный аналог функции обратного вызова в процедурном программировании.

Пример событийно-управляемой системы – приложение с пользовательским интерфейсом. При выборе некоторого пункта меню пользователем вырабатывается запрос на выполнение определенного действия (например, открытия файла).

Описание паттерна `Command`

Паттерн `Command` преобразовывает запрос на выполнение действия в отдельный объект-команду. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию. Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.

Интерфейс командного объекта определяется абстрактным базовым классом `Command` и в самом простом случае имеет единственный метод `execute()`. Производные классы определяют получателя запроса (указатель на объект-получатель) и необходимую для выполнения операцию (метод этого объекта). Метод `execute()` подклассов `Command` просто вызывает нужную операцию получателя.

В паттерне `Command` может быть до трех участников:

- *Клиент*, создающий экземпляр командного объекта.
- *Инициатор запроса*, использующий командный объект.
- *Получатель запроса*.

UML-диаграмма классов паттерна `Command`

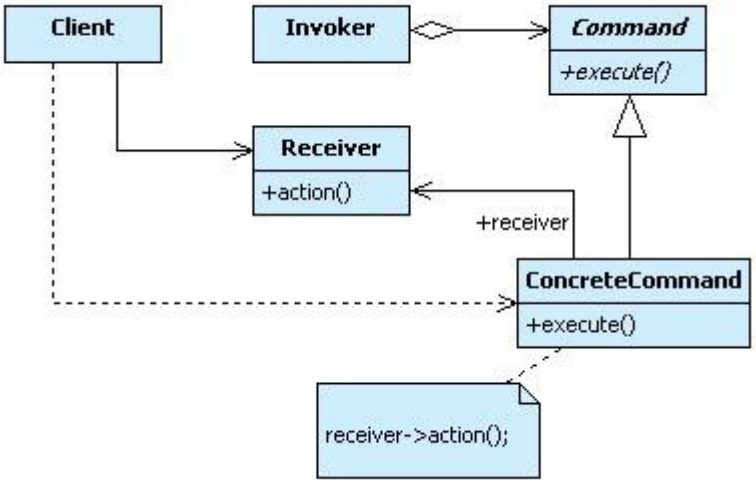


Рисунок 9.31

Сначала клиент создает объект `ConcreteCommand`, конфигурируя его получателем запроса. Этот объект также доступен инициатору. Инициатор использует его при отправке запроса, вызывая метод `execute()`. Этот алгоритм напоминает работу функции обратного вызова в процедурном программировании – функция регистрируется, чтобы быть вызванной позднее.

Паттерн `Command` отделяет объект, инициирующий операцию, от объекта, который знает, как ее выполнить. Единственное, что должен знать инициатор, это как отправить команду. Это придает системе гибкость: позволяет осуществлять динамическую замену команд, использовать сложные составные команды, осуществлять отмену операций.

Достоинства паттерна `Command`

- Придает системе гибкость, отделяя инициатора запроса от его получателя.

9.4.4 Паттерн Mediator (посредник)

Назначение паттерна `Mediator`

- Паттерн `Mediator` определяет объект, инкапсулирующий взаимодействие множества объектов. `Mediator` делает систему слабо связанной, избавляя объекты от необходимости ссылаться друг на друга, что позволяет изменять взаимодействие между ними независимо.
- Паттерн `Mediator` вводит посредника для развязывания множества взаимодействующих объектов.
- Заменяет взаимодействие «все со всеми» взаимодействием «один со всеми».

Решаемая проблема

Мы хотим спроектировать систему с повторно используемыми компонентами, однако существующие связи между этими компонентами можно охарактеризовать феноменом «спагетти-кода».

Спагетти-код – плохо спроектированная, слабо структурированная, запутанная и трудная для понимания программа. Спагетти-код назван так, потому что ход выполнения программы похож на миску спагетти, то есть извилистый и запутанный.

Обсуждение паттерна `Mediator`

В Unix права доступа к системным ресурсам определяются тремя уровнями: *владелец, группа и прочие*. *Группа* представляет собой совокупность пользователей, обладающих некоторой функциональной принадлежностью. Каждый пользователь в системе может быть членом одной или нескольких групп, и каждая группа может иметь 0 или более пользователей, назначенных этой группе. Следующий рисунок показывает трех пользователей, являющихся членами всех трех групп.

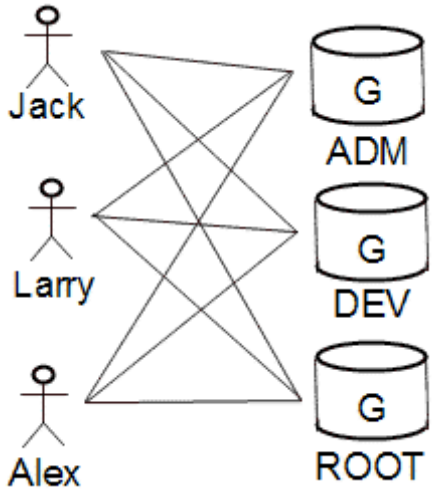


Рисунок 9.32

Если нам нужно было бы построить программную модель такой системы, то мы могли бы связать каждый объект `User` с каждым объектом `Group`, а каждый объект `Group` – с каждым объектом `User`. Однако из-за наличия множества взаимосвязей модифицировать поведение такой системы очень непросто, пришлось бы изменять все существующие классы.

Альтернативный подход – введение «дополнительного уровня косвенности» или построение абстракции из отображения (соответствия) пользователей в группы и групп в пользователей. Такой подход обладает следующими преимуществами: пользователи и группы отделены друг от друга, отображениями легко управлять одновременно и абстракция отображения может быть расширена в будущем путем определения производных классов.

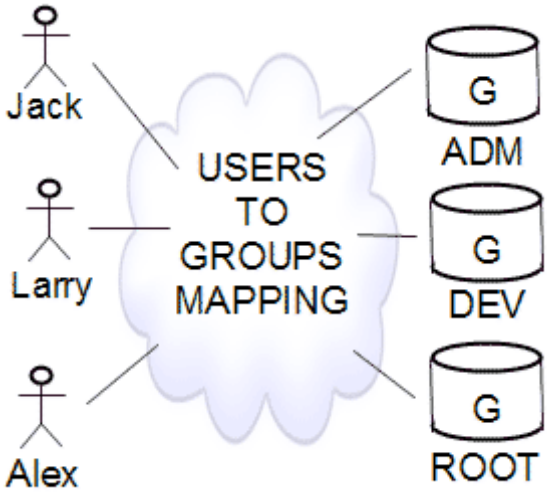


Рисунок 9.33

Разбиение системы на множество объектов в общем случае повышает степень повторного использования, однако множество взаимосвязей между этими объектами, как правило, приводит к обратному эффекту. Чтобы этого не допустить, инкапсулируйте взаимодействия между объектами в объект-посредник. Действуя как центр связи, этот объект-посредник контролирует и координирует взаимодействие группы объектов. При этом объект-посредник делает взаимодействующие объекты слабо связанными, так как им больше не нужно хранить ссылки друг на друга – все взаимодействие идет через этого посредника. Расширить или изменить это взаимодействие можно через его подклассы.

Паттерн `Mediator` заменяет взаимодействие «все со всеми» взаимодействием «один со всеми».

Пример рационального использования паттерна `Mediator` – моделирование отношений между пользователями и группами операционной системы. Группа может иметь `0` или более пользователей, а пользователь может быть членом `0` или более групп. Паттерн `Mediator` предусматривает гибкий способ управления пользователями и группами.

Структура паттерна `Mediator`

UML-диаграмма классов паттерна `Mediator`

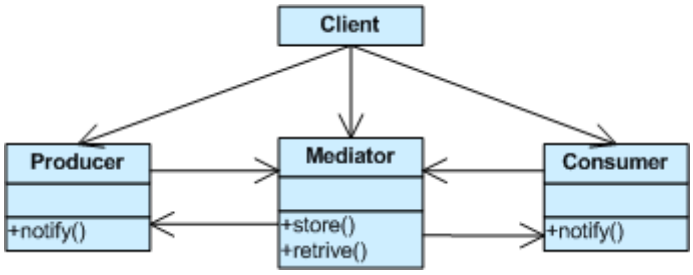


Рисунок 9.34

Коллеги (или взаимодействующие объекты) не связаны друг с другом. Каждый из них общается с посредником, который, в свою очередь, знает об остальных и управляет

ими. Паттерн `Mediator` делает статус взаимодействия «все со всеми» «полностью объектным».

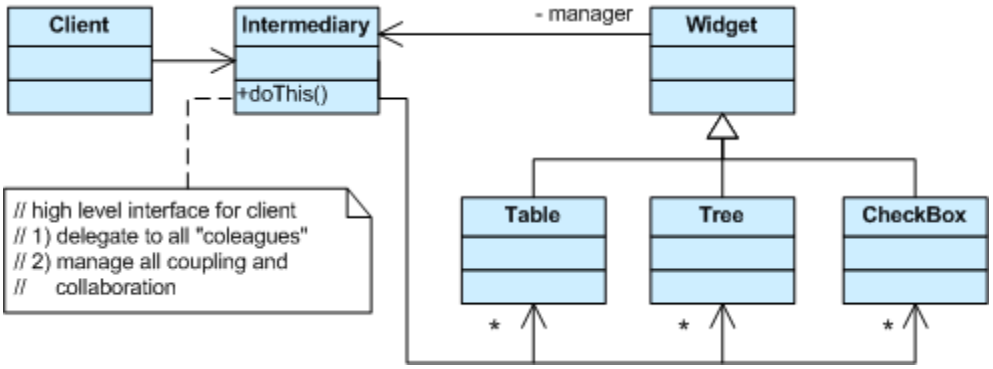


Рисунок 9.35

Пример паттерна `Mediator`

Паттерн `Mediator` определяет объект, управляющий набором взаимодействующих объектов. Слабая связанность достигается благодаря тому, что вместо непосредственного взаимодействия друг с другом коллеги общаются через объект-посредник. Башня управления полетами в аэропорту хорошо демонстрирует этот паттерн. Пилоты взлетающих или идущих на посадку самолетов в районе аэропорта общаются с башней вместо непосредственного общения друг с другом. Башня определяет, кто и в каком порядке будет садиться или взлетать. Важно отметить, что башня контролирует самолеты только в районе аэродрома, а не на протяжении всего полета.

Рисунок 9.36

Использование паттерна `Mediator`

- Определите совокупность взаимодействующих объектов, связанность между которыми нужно уменьшить.
- Инкапсулируйте все взаимодействия в абстракцию нового класса.
- Создайте экземпляр этого нового класса. Объекты-коллеги для взаимодействия друг с другом используют только этот объект.

- Найдите правильный баланс между принципом слабой связанности и принципом распределения ответственности.
- Будьте внимательны и не создавайте объект-«контроллер» вместо объекта-посредника.

Особенности паттерна `Mediator`

- Паттерны `Chain of Responsibility`, `Command`, `Mediator` и `Observer` показывают, как можно разделить отправителей и получателей запросов с учетом их особенностей. `Chain of Responsibility` передает запрос отправителя по цепочке потенциальных получателей. Command номинально определяет связь – «оправитель-получатель» с помощью подкласса. В `Mediator` отправитель и получатель ссылаются друг на друга косвенно, через объект-посредник. В паттерне `Observer` связь между отправителем и получателем слабее, при этом число получателей может конфигурироваться во время выполнения.
- `Mediator` и `Observer` являются конкурирующими паттернами. Если `Observer` распределяет взаимодействие с помощью объектов «наблюдатель» и «субъект», то `Mediator` использует объект-посредник для инкапсуляции взаимодействия между другими объектами. Мы обнаружили, что легче сделать повторно используемыми `Наблюдателей` и `Субъектов`, чем `Посредников`.
- С другой стороны, `Mediator` может использовать `Observer` для динамической регистрации коллег и их взаимодействия с посредником.
- `Mediator` похож `Facade` в том, что он абстрагирует функциональность существующих классов. `Mediator` абстрагирует/централизует взаимодействие между объектами-коллегами, добавляет новую функциональность и известен всем объектам-коллегам (то есть определяет двунаправленный протокол взаимодействия). `Facade`, наоборот, определяет более простой интерфейс к подсистеме, не добавляя новой функциональности, и неизвестен классам подсистемы (то есть имеет однонаправленный протокол взаимодействия, то есть запросы отправляются в подсистему, но не наоборот).

Реализация паттерна `Mediator`

Демонстрация паттерна `Mediator`

Степень повторного использования можно повысить через разбиение системы на множество объектов, однако при этом возникает множество взаимосвязей между этими объектами, которое приводит к обратному эффекту. Для исключения этой проблемы инкапсулируйте взаимодействие между объектами (коллективное поведение) в объект-посредник. Этот посредник будет управлять взаимодействием группы объектов.

В этом примере объект диалогового окна функционирует в качестве посредника. Виджеты диалогового окна ничего не знают о своих соседях. Всякий раз, когда происходит взаимодействие с пользователем виджет в `Widget::changed()` «делегирует» это событие посреднику `mediator->widgetChanged(this)`.

FileSelectionDialog::widgetChanged() инкапсулирует все коллективное поведение для диалогового окна (служит центром взаимодействия). Пользователь может выбрать «взаимодействие» с полем редактирования Filter, списком Directories, списком Files или полем редактирования Selection.

```
#include <iostream.h>

class FileSelectionDialog;

class Widget
{
  public:
    Widget(FileSelectionDialog *mediator, char *name)
    {
        _mediator = mediator;
        strcpy(_name, name);
    }
    virtual void changed();
    virtual void updateWidget() = 0;
    virtual void queryWidget() = 0;
  protected:
    char _name[20];
  private:
    FileSelectionDialog *_mediator;
};

class List: public Widget
{
  public:
    List(FileSelectionDialog   *dir,   char   *name):
Widget(dir, name){}
    void queryWidget()
    {
        cout << "   " << _name << " list queried" <<
endl;
    }
    void updateWidget()
    {
        cout << "   " << _name << " list updated" <<
endl;
    }
};

class Edit: public Widget
{
  public:
    Edit(FileSelectionDialog   *dir,   char   *name):
Widget(dir, name){}
    void queryWidget()
    {
```

```
            cout << "   " << _name << " edit queried" <<
endl;
        }
        void updateWidget()
        {
            cout << "   " << _name << " edit updated" <<
endl;
        }
    };

    class FileSelectionDialog
    {
      public:
        enum Widgets
        {
            FilterEdit, DirList, FileList, SelectionEdit
        };
        FileSelectionDialog()
        {
            _components[FilterEdit]   =   new   Edit(this,
"filter");
            _components[DirList] = new List(this, "dir");
            _components[FileList]   =   new   List(this,
"file");
            _components[SelectionEdit]  =  new  Edit(this,
"selection");
        }
        virtual ~FileSelectionDialog();
        void handleEvent(int which)
        {
            _components[which]->changed();
        }
        virtual         void         widgetChanged(Widget
*theChangedWidget)
        {
            if            (theChangedWidget            ==
_components[FilterEdit])
            {
                _components[FilterEdit]->queryWidget();
                _components[DirList]->updateWidget();
                _components[FileList]->updateWidget();
                _components[SelectionEdit]-
>updateWidget();
            }
            else       if       (theChangedWidget       ==
_components[DirList])
            {
                _components[DirList]->queryWidget();
                _components[FileList]->updateWidget();
                _components[FilterEdit]->updateWidget();
```

```
                _components[SelectionEdit]-
>updateWidget();
            }
            else       if       (theChangedWidget       ==
_components[FileList])
            {
                _components[FileList]->queryWidget();
                _components[SelectionEdit]-
>updateWidget();
            }
            else       if       (theChangedWidget       ==
_components[SelectionEdit])
            {
                _components[SelectionEdit]-
>queryWidget();
                cout << "   file opened" << endl;
            }
        }
      private:
        Widget *_components[4];
    };

    FileSelectionDialog::~FileSelectionDialog()
    {
      for (int i = 0; i < 3; i++)
        delete _components[i];
    }

    void Widget::changed()
    {
      _mediator->widgetChanged(this);
    }

    int main()
    {
      FileSelectionDialog fileDialog;
      int i;

      cout  <<  "Exit[0],  Filter[1],  Dir[2],  File[3],
Selection[4]: ";
      cin >> i;

      while (i)
      {
        fileDialog.handleEvent(i - 1);
        cout  <<  "Exit[0],  Filter[1],  Dir[2],  File[3],
Selection[4]: ";
        cin >> i;
      }
    }
```

Реализация паттерна `Mediator`: до и после

До

Объекты `Node` взаимодействуют друг с другом напрямую, требуется рекурсия, неудобное удаление узла, при этом первый узел удалить невозможно.

```
class Node
{
  public:
    Node(int v)
    {
        m_val = v;
        m_next = 0;
    }
    void add_node(Node *n)
    {
        if (m_next)
          m_next->add_node(n);
        else
          m_next = n;
    }
    void traverse()
    {
        cout << m_val << "  ";
        if (m_next)
          m_next->traverse();
        else
          cout << '\n';
    }
    void remove_node(int v)
    {
        if (m_next)
          if (m_next->m_val == v)
            m_next = m_next->m_next;
          else
            m_next->remove_node(v);
    }
  private:
    int m_val;
    Node *m_next;
};

int main()
{
  Node lst(11);
  Node two(22), thr(33), fou(44);
  lst.add_node(&two);
  lst.add_node(&thr);
  lst.add_node(&fou);
  lst.traverse();
  lst.remove_node(44);
  lst.traverse();
```

```
  lst.remove_node(22);
  lst.traverse();
  lst.remove_node(11);
  lst.traverse();
}
```

Результаты вывода программы:

```
11  22  33  44
11  22  33
11  33
11  33
```

После

«Посреднический» класс `List` упрощает все административные функции, рекурсия исключена.

```
class Node
{
  public:
    Node(int v)
    {
        m_val = v;
    }
    int get_val()
    {
        return m_val;
    }
  private:
    int m_val;
};

class List
{
  public:
    void add_node(Node *n)
    {
        m_arr.push_back(n);
    }
    void traverse()
    {
        for (int i = 0; i < m_arr.size(); ++i)
          cout << m_arr[i]->get_val() << "  ";
        cout << '\n';
    }
    void remove_node(int v)
    {
        for (vector::iterator it = m_arr.begin();
             it != m_arr.end(); ++it)
        if ((*it)->get_val() == v)
        {
            m_arr.erase(it);
            break;
        }
```

```
    }
  private:
    vector m_arr;
};

int main()
{
  List lst;
  Node one(11), two(22);
  Node thr(33), fou(44);
  lst.add_node(&one);
  lst.add_node(&two);
  lst.add_node(&thr);
  lst.add_node(&fou);
  lst.traverse();
  lst.remove_node(44);
  lst.traverse();
  lst.remove_node(22);
  lst.traverse();
  lst.remove_node(11);
  lst.traverse();
}
```

Результаты вывода программы:

```
11  22  33  44
11  22  33
11  33
33
```

9.4.5 Паттерн Chain of Responsibility (цепочка обязанностей)

Назначение паттерна `Chain of Responsibility`

- Паттерн `Chain of Responsibility` позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом запрос может быть обработан несколькими объектами. Объекты-получатели связываются в цепочку. Запрос передается по этой цепочке, пока не будет обработан.
- Вводит конвейерную обработку для запроса с множеством возможных обработчиков.
- Объектно-ориентированный связанный список с рекурсивным обходом.

Решаемая проблема

Имеется поток запросов и переменное число «обработчиков» этих запросов. Необходимо эффективно обрабатывать запросы без жесткой привязки к их обработчикам, при этом запрос может быть обработан любым обработчиком.

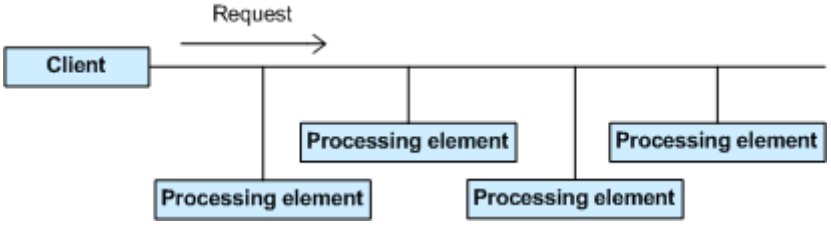


Рисунок 9.37

Обсуждение паттерна `Chain of Responsibility`

Инкапсулирует элементы по обработке запросов внутри абстрактного «конвейера». Клиенты «кидают» свои запросы на вход этого конвейера.

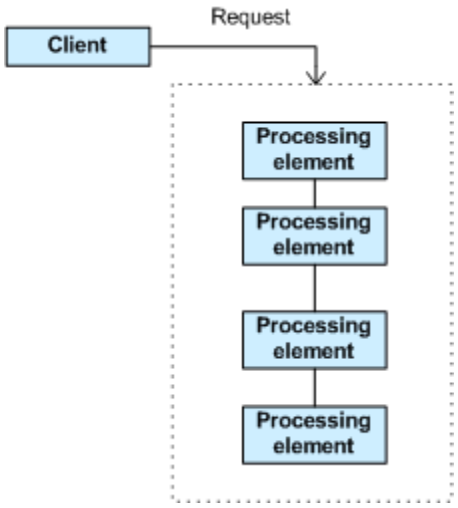


Рисунок 9.38

Паттерн `Chain of Responsibility` связывает в цепочку объекты-получатели, а затем передает запрос-сообщение от одного объекта к другому до тех пор, пока не достигнет объекта, способного его обработать. Число и типы объектов-обработчиков заранее неизвестны, они могут настраиваться динамически. Механизм связывания в цепочку использует рекурсивную композицию, что позволяет использовать неограниченное число обработчиков.

Паттерн `Chain of Responsibility` упрощает взаимосвязи между объектами. Вместо хранения ссылок на всех кандидатов-получателей запроса, каждый отправитель хранит единственную ссылку на начало цепочки, а каждый получатель имеет единственную ссылку на своего преемника - последующий элемент в цепочке.

Убедитесь, что система корректно «отлавливает» случаи необработанных запросов.

Не используйте паттерн `Chain of Responsibility`, когда каждый запрос обрабатывается только одним обработчиком, или когда клиент знает, какой именно объект должен обработать его запрос.

Структура паттерна `Chain of Responsibility`

Производные классы знают, как обрабатывать запросы клиентов. Если «текущий» объект не может обработать запрос, то он делегирует его базовому классу, который делегирует «следующему» объекту и так далее.

UML-диаграмма классов паттерна `Chain of Responsibility`

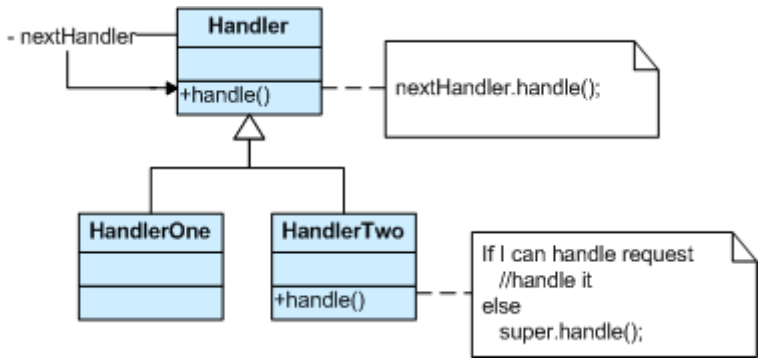


Рисунок 9.39

Обработчики могут вносить свой вклад в обработку каждого запроса. Запрос может быть передан по всей длине цепочки до самого последнего звена.

Пример паттерна `Chain of Responsibility`

Паттерн `Chain of Responsibility` позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким получателям. Банкомат использует `Chain of Responsibility` в механизме выдачи денег.

Рисунок 9.40

Использование паттерна `Chain of Responsibility`

- Базовый класс имеет указатель на «следующий обработчик».
- Каждый производный класс реализует свой вклад в обработку запроса.
- Если запрос должен быть «передан дальше», то производный класс «вызывает» базовый класс, который с помощью указателя делегирует запрос далее.
- Клиент (или третья сторона) создает цепочку получателей (которая может иметь ссылку с последнего узла на корневой узел).
- Клиент передает каждый запрос в начало цепочки.

- Рекурсивное делегирование создает иллюзию волшебства.

Особенности паттерна `Chain of Responsibility`

- Паттерны `Chain of Responsibility`, `Command`, `Mediator` и `Observer` показывают, как можно разделить отправителей и получателей с учетом их особенностей. `Chain of Responsibility` передает запрос отправителя по цепочке потенциальных получателей.
- `Chain of Responsibility` может использовать `Command` для представления запросов в виде объектов.
- `Chain of Responsibility` часто применяется вместе с паттерном `Composite`. Родитель компонента может выступать в качестве его преемника.

Реализация паттерна `Chain of Responsibility`

Реализация паттерна `Chain of Responsibility` по шагам

1. Создайте указатель на следующий обработчик `next` в базовом классе.
2. Метод `handle()` базового класса всегда делегирует запрос следующему объекту.
3. Если производные классы не могут обработать запрос, они делегируют его базовому классу.

```
#include <iostream>
#include <vector>
#include <ctime>
using namespace std;

class Base
{
    // 1. Указатель "next" в базовом классе
    Base *next;
  public:
    Base()
    {
        next = 0;
    }
    void setNext(Base *n)
    {
        next = n;
    }
    void add(Base *n)
    {
        if (next)
          next->add(n);
        else
          next = n;
    }
    // 2. Метод базового класса, делегирующий запрос next-объекту
    virtual void handle(int i)
    {
        next->handle(i);
    }
};
```

```
class Handler1: public Base
{
  public:
     /*virtual*/void handle(int i)
    {
        if (rand() % 3)
        {
            // 3. 3 из 4 запросов не обрабатываем
            cout << "H1 passsed " << i << "  ";
            // 3. и делегируем базовому классу
            Base::handle(i);        }
        else
          cout << "H1 handled " << i << "  ";
    }
};

class Handler2: public Base
{
  public:
     /*virtual*/void handle(int i)
    {
        if (rand() % 3)
        {
            cout << "H2 passsed " << i << "  ";
            Base::handle(i);
        }
        else
          cout << "H2 handled " << i << "  ";
    }
};

class Handler3: public Base
{
  public:
     /*virtual*/void handle(int i)
    {
        if (rand() % 3)
        {
            cout << "H3 passsed " << i << "  ";
            Base::handle(i);
       }
        else
          cout << "H3 handled " << i << "  ";
    }
};

int main()
{
  srand(time(0));
  Handler1 root;
```

```
 Handler2 two;
 Handler3 thr;
 root.add(&two);
 root.add(&thr);
thr.setNext(&root);
 for (int i = 1; i < 10; i++)
 {
   root.handle(i);
  cout << '\n';
 }
```

9.4.6 Паттерн Strategy (стратегия)

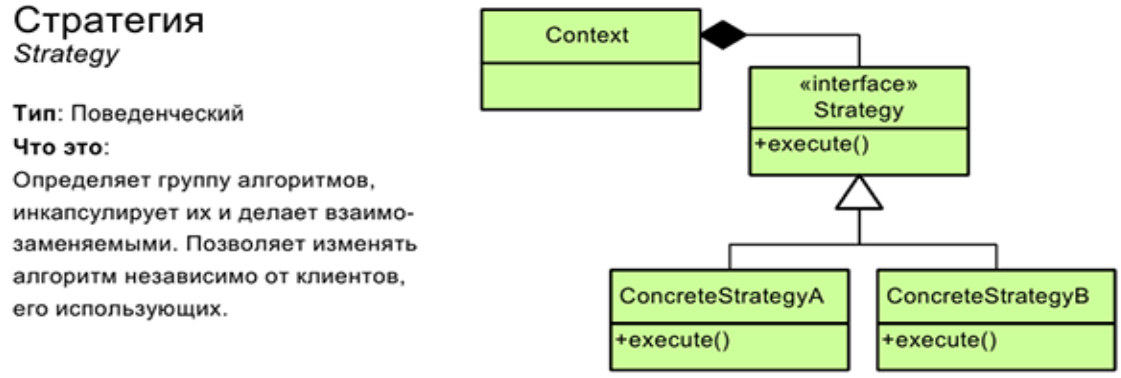


Рисунок 9.41

Назначение паттерна `Strategy`

Существуют системы, поведение которых может определяться согласно одному алгоритму из некоторого семейства. Все алгоритмы этого семейства являются родственными: предназначены для решения общих задач, имеют одинаковый интерфейс для использования и отличаются только реализацией (поведением). Пользователь, предварительно настроив программу на нужный алгоритм (выбрав стратегию), получает ожидаемый результат. Как пример, – приложение, предназначенное для компрессии файлов использует один из доступных алгоритмов: `zip`, `arj` или `rar`.

Объектно-ориентированный дизайн такой программы может быть построен на идее использования полиморфизма. В результате получаем набор родственных классов с общим интерфейсом и различными реализациями алгоритмов.

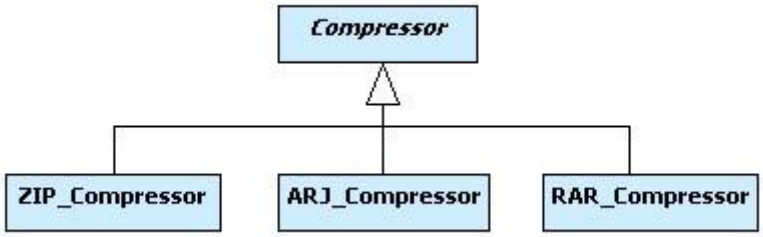


Рисунок 9.42

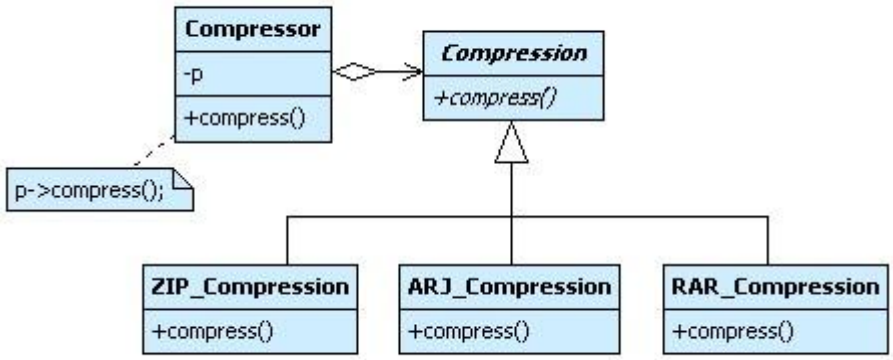


Рисунок 9.43

Представленному подходу свойственны следующие недостатки:

1. Реализация алгоритма жестко привязана к его подклассу, что затрудняет поддержку и расширение такой системы.
2. Система, построенная на основе наследования, является статичной. Заменить один алгоритм на другой в ходе выполнения программы уже невозможно.

Применение паттерна `Strategy` позволяет устранить указанные недостатки.

Описание паттерна `Strategy`

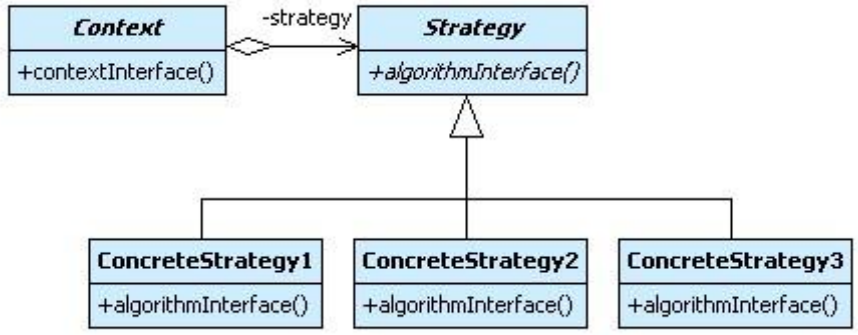


Рисунок 9.44

Паттерн `Strategy` переносит в отдельную иерархию классов все детали, связанные с реализацией алгоритмов. Для случая программы сжатия файлов абстрактный базовый класс `Compression` этой иерархии объявляет интерфейс, общий для всех алгоритмов и используемый классом `Compressor`. Подклассы `ZIP_Compression`, `ARJ_Compression` и `RAR_Compression` его реализуют в соответствии с тем или иным алгоритмом. Класс `Compressor` содержит указатель на объект абстрактного типа `Compression` и предназначен для переадресации пользовательских запросов конкретному алгоритму. Для замены одного алгоритма другим достаточно перенастроить этот указатель на объект нужного типа.

Результаты применения паттерна `Strategy`

Достоинства паттерна `Strategy`

- Систему проще поддерживать и модифицировать, так как семейство алгоритмов перенесено в отдельную иерархию классов.
- Паттерн `Strategy` предоставляет возможность замены одного алгоритма другим в процессе выполнения программы.
- Паттерн `Strategy` позволяет скрыть детали реализации алгоритмов от клиента.

Недостатки паттерна `Strategy`

- Для правильной настройки системы пользователь должен знать об особенностях всех алгоритмов.
- Число классов в системе, построенной с применением паттерна `Strategy`, возрастает.

Реализация паттерна `STRATEGY`

```
#include <iostream>
#include <string>

// Иерархия классов, определяющая алгоритмы сжатия файлов
class Compression
{
  public:
    virtual ~Compression() {}
    virtual void compress( const string & file ) = 0;
};

class ZIP_Compression : public Compression
{
  public:
    void compress( const string & file ) {
        cout << "ZIP compression" << endl;
    }
};

class ARJ_Compression : public Compression
{
  public:
    void compress( const string & file ) {
        cout << "ARJ compression" << endl;
    }
};

class RAR_Compression : public Compression
{
  public:
    void compress( const string & file ) {
        cout << "RAR compression" << endl;
    }
};
```

```
// Класс для использования
class Compressor
{
  public:
    Compressor( Compression* comp): p(comp) {}
   ~Compressor() { delete p; }
    void compress( const string & file ) {
      p->compress( file);
    }
  private:
    Compression* p;
};


int main()
{
  Compressor* p = new Compressor( new ZIP_Compression);
  p->compress( "file.txt");
  delete p;
  return 0;
}
```

10 ТЕСТИРОВАНИЕ

Контроль качества, надежности и безопасности создаваемых и модифицируемых программ должен сопровождать весь жизненный цикл ПО. Это достигается путем применения специальной, достаточно эффективной технологической системы тестирования программ и обеспечения их качества.

Для обнаружения и устранения ошибок проектирования все этапы разработки и сопровождения ПО должны быть поддержаны методами и средствами систематического ***автоматизированного тестирования и поэтапных испытаний***.

Для проверки достигнутого качества, надежности и безопасности применения сложных, критических программных продуктов служит их обязательная сертификация в аттестованных проблемно-ориентированных сертификационных лабораториях.

Непредсказуемость вида, места и времени проявления дефектов ПО в процессе эксплуатации приводит к необходимости создания специальных, дополнительных ***систем автоматической оперативной защиты*** от непредумышленных, случайных искажений вычислительного процесса, программ и данных. Оперативная защита за счет временной, программной и информационной избыточности позволяет выявить и блокировать распространение негативных последствий проявления дефектов и уменьшить их влияние на качество функционирования ПО до устранения первопричины дефекта.

Первоочередными мерами по обеспечению качества ПО является ***воспитание и обучение*** следующих двух **категорий специалистов**:

1. **Специалисты, управляющие качеством ПО**. Они *должны владеть*:

- методиками и стандартами фирмы, поддерживающими тестирование, контроль, документирование; управляющими воздействиями, на показатели качества на всех этапах жизненного цикла программы.

Должны выявлять:

- все отклонения от заданных параметров качества объектов и процессов;
- отклонения от предписанной технологии тестирования на промежуточных и заключительных этапах разработки.

Должны анализировать возможные последствия выявленных отклонений от требований технического задания или спецификаций ПО. Результатом анализа могут являться либо меры по устранению отклонений либо пересмотр ТЗ или спецификаций, в случае если вариант с их полным выполнением неприемлем по срокам или затратам ресурсов.

Таким образом, **основные задачи данной группы** специалистов **сосредоточены на контроле качества процессов** и результатов выполнения работ. Кроме того, на них лежит ответственность за принятие организационных и технологических мер для достижения необходимого качества, обеспечивающего выполнение требований технического задания и спецификаций.

2. **Непосредственные разработчики компонентов** и программного обеспечения в целом (с заданными показателями качества). В процессе управления качеством задача данной группы состоит в тщательном **соблюдении** принятой в фирме **технологии разработки**, **тестирования и испытаний** и в формировании всех предписанных руководствами исходных и отчетных документов. При этом предполагается, что выбранная технология тестирования способна обеспечить необходимые значения показателей качества, а достижение заданных функциональных характеристик гарантируется тематической квалификацией соответствующих специалистов и регулярным контролем данных характеристик в процессе разработки.

Система стандартизированного документирования частных работ должна обеспечить объективное отражение достигнутого качества компонентов и процессов их создания на всех этапах разработки.

Организационное разделение специалистов, осуществляющих разработку и тестирование ПО, и специалистов, контролирующих и управляющих его качеством в процессе разработки, обеспечивает независимый достоверный контроль качества результатов разработки и эффективное достижение заданных характеристик.

Дестабилизирующие факторы и методы обеспечения высокого качества функционирования ПО

В общем случае под ошибкой подразумевается дефект, погрешность или неумышленное искажение объекта или процесса. При этом предполагается, что известно правильное, эталонное состояние объекта, по отношению к которому **может быть определено** наличие отклонения – ***дефекта или ошибки***. Для систематической, скоординированной борьбы с ними **необходимы исследования факторов**, влияющих

на качество ПО со стороны случайных, существующих (и потенциально возможных) дефектов в конкретных программах.

При строго фиксированных исходных данных программы исполняются по **заданным маршрутам** и выдают **строго предопределенные результаты**. Многочисленные варианты исполнения программ при разнообразных исходных данных представляются для внешнего наблюдателя как случайные. В связи с этим **дефекты** функционирования программных средств, **не вызванные злоумышленными действиями**, **проявляются** внешне **как случайные**, имеют разную природу и последствия. В частности, они могут приводить к последствиям, соответствующим нарушениям работоспособности, и к отказам при использовании ПО.

Степень влияния всех внутренних дестабилизирующих факторов, а также некоторых внешних угроз на качество и надежность ПО определяется в наибольшей степени ***качеством технологий анализа требований, проектирования, кодирования, тестирования, сопровождения и документирования ПО*** и его основных компонентов. При ограниченных ресурсах на разработку ПО для достижения заданных требований **необходимо управление обеспечением качества** в течение всего цикла создания программ и данных. **Такое управление подразумевает высокую дисциплину и проектировочную культуру всего коллектива специалистов**, использование им методик, типовых нормативных документов и средств автоматизации разработки. Кроме того, обеспечение качества ПО предполагает формализацию и сертификацию технологий разработки, а также выделение в специальный процесс, поэтапное измерение и анализ качества создаваемых и применяемых компонент.

Предотвращение ошибок и улучшение технико-экономических показателей создания ПО обеспечивается применением современных технологий и систем автоматизированного проектирования, объединенных понятиями CASE и 4GL(графические языки четвертого поколения).

10.1 Использование CASE для повышения качества ПО

CASE-технологии (Computer Aided Software/System Engineering) представляют собой **высокопроизводительные**, **ресурсосберегающие технологии** создания комплексов программ высокого качества и надежность. Основная цель применения CASE- сокращение общих затрат на проектирование, реализацию, сопровождение и развитие ПО. **Применение CASE** приводит к **исключению** или **значительному уменьшению** количества **системных**, **алгоритмических** и **программных ошибок** в ПО, передаваемом в эксплуатацию. Кроме того, CASE-технологии эффективны при модификации и сопровождении ПО, а также при адаптации ПО к изменениям конфигурации внешней среды. Повышение уровня автоматизации проектирования, применение методов и средств CASE-технологий – один из самых эффективных путей повышения качества разрабатываемого ПО.

При создании ПО **важная пробл**ема заключается **в** системотехническом и **информационно-технологическом проекте**, обеспечивающем высокие потребительские качества и надежность ПО. CASE-средства предназначены для реализации проектов коллективами разработчиков и базируются на конкретных

методологиях коллективной разработки и сопровождения ПО. CASE-средства используются для изъятия и формализации знаний заказчика на этапе проведения обследования, анализа и подготовки технического задания. Затем CASE-средства могут быть использованы для проектирования концептуальной и логической структур разрабатываемого ПО, используемых в нем баз данных. При этом в CASE-средствах активно используется тестирование корректности реализованных системных решений. Одновременно благодаря высокому качеству проработки и документирования такого проекта создается основа для снижения трудоемкости отладки, тестирования, испытаний, сопровождения и развития разрабатываемого ПО.

Совместное применение CASE и 4GLспособно снизить трудоемкость разработки сложных программных средств в несколько раз и сократить продолжительность разработки до нескольких месяцев или даже недель.

Базовым принципом современных методов и технологий создания прикладного ПО является многократное использование отработанных технических решений на различных платформах. В настоящее время основная часть программных средств не создается вновь, а переносится с других платформ или комплексируется и собирается из готовых, испытанных и повторно используемых компонент гарантированно высокого качества.

Результатом внедрения CASE-средств в разработку ПО является значительное сокращение затрат на разработку, высокое качество проекта и надежность полученного ПО.

IBM Rational Functional Tester – мощное средство функционального тестирования для приложений Java, Web, VS.NET и WinForm, **автоматизирующее процессы функционального и регрессионного тестирования**. Основные возможности IBM Rational Functional Tester:

• обеспечивает возможность автоматизации для тестирования, управляемого данными (*datadriven testing*), и выбор языка разработки скриптов вместе с мощным редактором для создания и настройки скриптов;

• облегчает начинающим тестировщикам автоматизацию тестирования благодаря таким возможностям, как тестирование, управляемое данными;

• предоставляет выбор языка разработки скриптов для опытных тестировщиков: `Java` для `Eclipse` или `Microsoft Visual Basic .NET` для `Visual Studio .NET`;

• дополнительно имеются инструменты функционального и регрессионного тестирования для работы с графическим интерфейсом пользователя (GUI);

• включает технологию ScriptAssure и возможности использования шаблонов для улучшения устойчивости скриптов к внесению частых незначительных изменений в пользовательский интерфейс;

• поддерживает контроль версий и параллельную разработку использование скриптов, в том числе для географически распределенных команд;

• поддерживает тестирование приложений на базе 3270 (zSeries) и 5250 (iSeries) при использовании дополнительно IBM Rational Functional Tester Extension for Terminal-based Applications;

• при использовании расширений (Extention) обеспечивает автоматизированное функциональное и регрессионное тестирование для приложений на основе Siebel 7.7, 7.8 и SAP;

• поддерживает специализированные средства управления через промежуточную среду разработки (`Java/.Net`);

• интегрируется с IBM Rational Quality Manager;

• есть возможность использования ключевых слов для частичной автоматизации ручного тестирования. IBM **Rational Performance Tester** – **многопользовательский инструмент тестирования**, предназначенный для **проверки масштабируемости приложений** перед их развертыванием. Rational Performance Tester помогает точно определять узкие места системы до ее развертывания при помощи эмуляции одновременной работы заданного числа пользователей и генерирования отчетов. Функции высокого уровня включают в себя подробное планирование тестирования на уровне пользовательской активности и модели использования каждой из их групп. Rational Performance Tester также предоставляет **возможность автоматической организации пула данных**, что позволяет **изменять набор тестовых данных**, используемый каждым **смоделированным пользователем**.

10.2 Влияние стандартов открытых систем на качество ПО

Предотвращению дефектов в сложном, распределенном ПО способствует развитие и применение концепции и стандартов открытых систем.

Однако их использование сопряжено с некоторыми тенденциями в составе и величинах **рисков**, отражающихся **на качестве ПО.** Стандартизация интерфейсов между внешними и внутренними компонентами ПО и возможность массового переноса данных на различные платформы **способствует** распространению **невыявленных дефектов** и ошибок, остающихся в переносимых компонентах. Усложнение взаимодействия программ и данных в распределенном ПО также приводит к возрастанию появления рисков и снижению качества и надежности.

Однако переносимые компоненты, как правило, **тщательнее тестируются** и испытываются, поэтому имеют более **высокое качество** чем те, которые созданы без ориентации на переносимость. Стандартизация и глубокий формализованный контроль интерфейсов и протоколов взаимодействия компонентов ПО позволяют создавать сложные, распределенные комплексы программ с высоким качеством.

Строгое соблюдение и контроль соответствия стандартам открытых систем является высокоэффективным методом предотвращения ряда классов дефектов и ошибок и повышения качества ПО.

10.2.1 Повышение качества ПО путем тестирования

Для обнаружения и устранения ошибок в ПО все этапы его разработки и сопровождения должны быть поддержаны методами систематического, автоматизированного тестирования корректности реализованных решений.

Качество функционирования ПО непосредственно зависит от **полноты применяющихся комплексов** тестов и адекватности генераторов тестов реальным объектам внешней среды и условиям будущей эксплуатации.

Тестирование– основной метод измерения качества, определения корректности и реальной надежности функционирования программ на любом этапе разработки. Результаты тестирования и измерения показателей качества должны сравниваться с требованиями технического задания или спецификаций для определения соответствия разрабатываемого ПО требованиям, полученным разработчиком от заказчика.

Важная особенность тестирования сложного ПО – необходимость достаточно **полной проверки** программ при ограниченной длительности испытаний. Эта особенность определяет целесообразность тщательного планирования тестирования с учетом всех результатов, полученных на предыдущих этапах разработки. Основная задача такого планирования – достижение максимальной достоверности испытаний, наиболее полного определения качества и надежности ПО при ограничении ресурсов на проведение испытаний.

Для достижения высокого качества **целесообразно проводить испытания** не только **завершенного** ПО, но и результатов **ряда промежуточных этапов**. Для этого в процессе формирования технического задания следует **формулировать план** и основные положения методики обеспечения качества, поэтапных испытаний компонент и определения характеристик, допустимых для продолжения работ на следующих этапах. Одновременно следует проводить уточнение технического задания и методик испытания ПО. Применение такого подхода позволяет испытателям и представителям заказчика глубоко ознакомиться с создаваемым ПО, провести поэтапный контроль качества и достаточно полный учет результатов тестирования при последующих испытаниях.

10.2.2 Основные особенности процесса тестирования ПО

- Отсутствие полностью определенного достоверного эталона – программы, которому должны точно соответствовать все результаты тестирования проверяемого ПО.
- Высокая сложность комплексов программ и принципиальная невозможность построения полных комплектов тестовых наборов, достаточных для их исчерпывающей проверки, в том числе и на надежность функционирования.
- Относительно невысокая степень формализации критериев качества процесса тестирования и достигаемых при этом корректности и надежности функционирования объектов испытаний.

10.2.3 Организационные особенности тестирования

На **начальных этапах** процессы **разработки** и **тестирования** часто практически **смыкаются**, однако целесообразно, по возможности, разделять их методически, инструментально и организационно (по исполняющим специалистам). Для сложного ПО систематическому тестированию программ приходится уделять столько же времени и сил, сколько непосредственно разработке. Недооценка необходимости планомерного тестирования в процессе разработки приводит к резкому возрастанию затрат на выявление и исправление ошибок в процессе завершающих испытаний и эксплуатации, а также к снижению надежности использования такого ПО.

Модели внешней среды и совокупные наборы тестов по своей сложности сопоставимы с самими тестируемыми объектами и также не гарантированы от ошибок. В результате в программах и данных всегда остаются ошибки, часть из которых проявляется позже, в процессе эксплуатации ПО в реальных условиях.

Это приводит к необходимости сопровождения всего тиража ПО, выявления и устранения дефектов, а также последовательного выпуска очередных, исправленных версий ПО для замены эксплуатируемых.

Тестирование — это проверка соответствия ПО требованиям, осуществляемая с помощью наблюдения за его работой в специальных, искусственно построенных ситуациях. Такого рода ситуации называют тестовыми или просто **тестами**.

Тестирование — конечная процедура. Набор ситуаций, в которых будет проверяться тестируемое ПО, всегда конечен. Более того, он должен быть настолько мал, чтобы *тестирование* можно было провести в рамках проекта разработки ПО, не слишком увеличивая его бюджет и сроки. Это означает, что при *тестировании* всегда проверяется очень небольшая доля всех возможных ситуаций. По этому поводу *Дейкстра (Dijkstra)* заметил, что *тестирование* позволяет точно определить, что в программе есть ошибка, но не позволяет утверждать, что там нет ошибок.

Тем не менее, *тестирование* может использоваться для достаточно уверенного вынесения оценок о *качестве ПО*. Для этого необходимо иметь **критерии полноты** *тестирования*, описывающие важность различных ситуаций для оценки качества, а также эквивалентности и зависимости между ними. Этот критерий может утверждать, что все равно в какой из ситуаций, A или B, проверять правильность работы ПО, или, если программа правильно работает в ситуации A, то, скорее всего, в ситуации B все тоже будет правильно. Часто критерий полноты *тестирования* задается при помощи определения эквивалентности ситуаций, дающей конечный набор классов ситуаций. В этом случае считают, что все равно, какую из ситуаций одного класса использовать в качестве теста. Такой критерий называют **критерием тестового покрытия**, а процент классов эквивалентности ситуаций, случившихся во время *тестирования*, — достигнутым **тестовым покрытием**.

Таким образом, **основные задачи *тестирования***: построить такой набор ситуаций, который был бы достаточно представителен и позволял бы завершить *тестирование* с достаточной степенью уверенности в правильности ПО вообще, и убедиться, что в конкретной ситуации ПО работает правильно, в соответствии с требованиями.

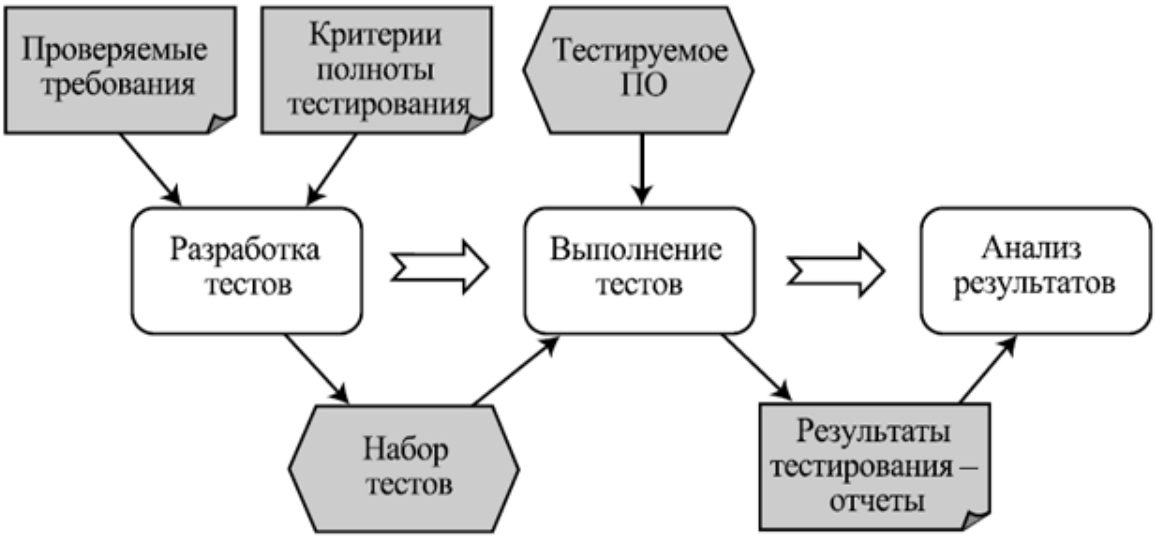


Рисунок 10.1 Схема процесса тестирования

Тестирование — наиболее широко применяемый *метод контроля качества*. Для оценки многих атрибутов качества не существует других эффективных способов, кроме *тестирования*.

Организация *тестирования* ПО регулируется следующими стандартами:

- IEEE 829-1998 Standard for Software Test Documentation. Описывает виды документов, служащих для подготовки тестов.
- IEEE 1008-1987 (R1993, R2002) Standard for Software *Unit Testing*. Описывает организацию модульного *тестирования* (см. ниже).
- ISO/IEC 12119:1994 (аналог AS/NZS 4366:1996 и ГОСТ Р-2000, также принят IEEE под номером IEEE 1465-1998) Information Technology. *Software packages* — Quality requirements and testing.

  Описывает требования к процедурам *тестирования* программных систем.

Тестировать можно соблюдение любых требований, соответствие которым выявляется во время работы ПО. Из характеристик качества по ISO 9126 этим свойством не обладают только атрибуты *удобства сопровождения*. Поэтому выделяют виды *тестирования*, связанные с проверкой определенных характеристик и атрибутов качества — *тестирование функциональности*, *надежности*, *удобства использования*, *переносимости* и *производительности*, а также *тестирование* защищенности, функциональной пригодности и пр. Кроме того, особо выделяют **нагрузочное** или **стрессовое** *тестирование*, проверяющее работоспособность ПО и показатели его *производительности* в условиях повышенных нагрузок — при большом количестве пользователей, интенсивном обмене данными с другими системами, большом объеме передаваемых или используемых данных и пр.

На основе исходных данных, используемых для построения тестов, *тестирование* делят на следующие виды:

- *Тестирование* **черного ящика**, нацеленное на проверку требований. Тесты для него и критерий полноты *тестирования* строятся на основе требований и ограничений, четко зафиксированных в спецификациях, стандартах, внутренних

нормативных документах. Часто такое *тестирование* называется *тестированием* **на соответствие (conformance testing)**. Частным случаем его является **функциональное** *тестирование* — тесты для него, а также используемые критерии полноты проведенного *тестирования* определяют на основе требований к *функциональности*.

Еще одним примером *тестирования* на соответствие является **аттестационное** или **квалификационное** *тестирование*, по результатам которого программная система получает (или не получает) официальный документ, подтверждающий ее соответствие определенным требованиям и стандартам.

- *Тестирование* **белого ящика**, оно же **структурное** *тестирование* — тесты создаются на основе знаний о структуре самой системы и о том, как она работает. Критерии полноты основаны на проценте элементов кода, которые отработали в ходе выполнения тестов. Для оценки степени соответствия требованиям могут привлекаться дополнительные знания о связи требований с определенными ограничениями на значения внутренних данных системы (например, на значения параметров вызовов, результатов и локальных переменных).
- *Тестирование*, нацеленное на определенные ошибки. Тесты для такого *тестирования* строятся так, чтобы гарантированно выявлять определенные виды ошибок. Полнота *тестирования* определяется на основе количества проверенных ситуаций по отношению к общему числу ситуаций, которые мы пытались проверить. К этому виду относится, например, *тестирование* **на отказ (smoke testing)**, в ходе которого просто пытаются вывести систему из строя, давая ей на вход как обычные данные, так и некорректные, с нарочно внесенными ошибками.

Другим примером служит метод оценки полноты *тестирования* при помощи набора **мутантов** — программ, совпадающих с тестируемой всюду, кроме нескольких мест, где специально внесены некоторые ошибки. Чем больше *мутантов* не проходит успешно через данный набор тестов, тем полнее и качественнее проводимое с его помощью *тестирование*.

Еще одна классификация видов *тестирования* основана на том уровне, на который оно нацелено. Эти же разновидности *тестирования*можно связать с фазой жизненного цикла, на которой они выполняются.

- **Модульное** *тестирование* **(unit testing)** предназначено для проверки правильности отдельных модулей, вне зависимости от их окружения. При этом проверяется, что если модуль получает на вход данные, удовлетворяющие определенным критериям корректности, то и результаты его корректны. Для описания критериев корректности входных и выходных данных часто используют **программные контракты — предусловия**, описывающие для каждой операции, на каких входных данных она предназначена работать, **постусловия**, описывающие для каждой операции, как должны соотноситься входные данные с возвращаемыми ею результатами, и **инварианты**, определяющие критерии целостности внутренних данных модуля.

Модульное *тестирование* является важной составной частью **отладочного** *тестирования*, выполняемого разработчиками для отладки написанного ими кода.

- **Интеграционное** *тестирование* **(integration testing)** предназначено для проверки правильности взаимодействия модулей некоторого набора друг с другом. При этом проверяется, что в ходе совместной работы модули обмениваются данными и вызовами операций, не нарушая взаимных ограничений на такое взаимодействие, например, предусловий вызываемых операций. Интеграционное *тестирование* также используется при отладке, но на более позднем этапе разработки.
- **Системное** *тестирование* **(system testing)** предназначено для проверки правильности работы системы в целом, ее способности правильно решать поставленные пользователями задачи в различных ситуациях.

Системное *тестирование* выполняется через внешние интерфейсы ПО и тесно связано с *тестированием* **пользовательского интерфейса** (или через пользовательский интерфейс), проводимым при помощи имитации действий пользователей над элементами этого интерфейса. Частными случаями этого вида *тестирования* являются *тестирование* **графического пользовательского интерфейса** (*Graphical User Interface*, GUI) и **пользовательского интерфейса Web-приложений (WebUI)**.

Если интеграционное и модульное *тестирование* чаще всего проводят, воздействуя на компоненты системы при помощи операций предоставляемого ими программного интерфейса (Application Programming Interface, API), то на системном уровне без использования пользовательского интерфейса не обойтись, хотя *тестирование* через API в этом случае также вполне возможно.

Основной недостаток *тестирования* состоит в том, что проводить его можно, только когда проверяемый элемент программы уже разработан. Снизить влияние этого ограничения можно, подготавливая тесты (а это — наиболее трудоемкая часть *тестирования*) на основе требований заранее, когда исходного кода еще нет. Подход опережающей разработки тестов с успехом используется, например, в рамках XP.

Проверка на моделях

Проверка свойств на моделях (model checking) — проверка соответствия ПО требованиям при помощи формализации проверяемых свойств, построения формальных моделей проверяемого ПО (чаще всего в виде автоматов различных видов) и автоматической проверки выполнения этих свойств на построенных моделях. *Проверка свойств на моделях* позволяет проверять достаточно сложные свойства автоматически, при минимальном участии человека. Однако она оставляет открытым вопрос о том, насколько выявленные свойства модели можно переносить на само ПО.

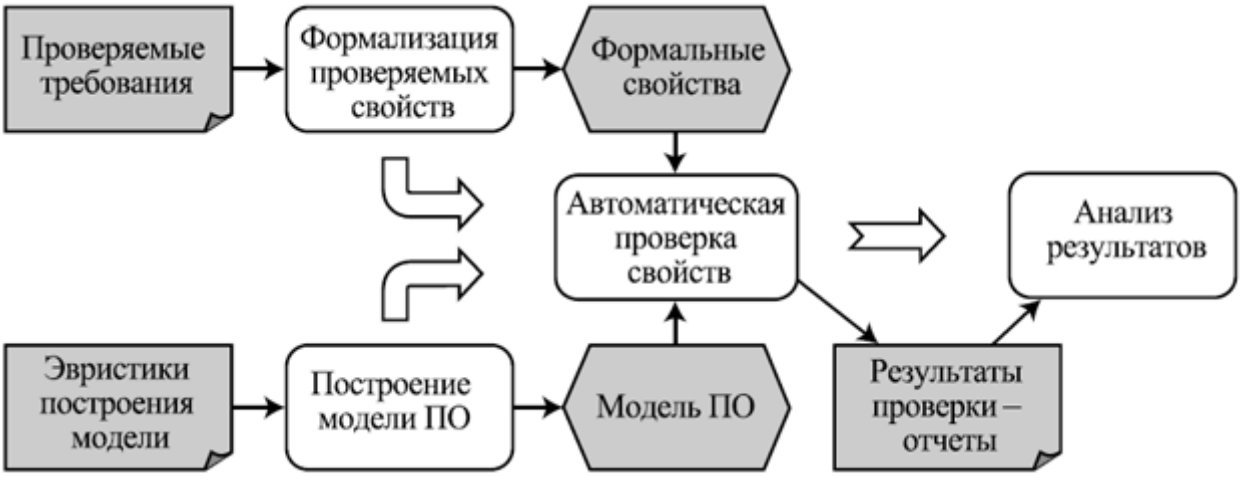


Рисунок 10.2 Схема процесса проверки свойств ПО на моделях

Обычно при помощи *проверки свойств на моделях* анализируют два вида свойств алгоритмов, использованных при построении ПО. **Свойства безопасности (safety properties)** утверждают, что нечто нежелательное никогда не случится в ходе работы ПО. **Свойства живучести (liveness properties)** утверждают, наоборот, что нечто желательное при любом развитии событий произойдет в ходе его работы.

Примером свойства первого типа служит отсутствие **взаимных блокировок (deadlocks)**. Взаимная блокировка возникает, если каждый из группы параллельно работающих в проверяемом ПО процессов или потоков ожидает прибытия данных или *снятия блокировки* ресурса от одного из других, а тот не может продолжить выполнение, ожидая того же от первого или от третьего процесса, и т.д.

Примером свойства живости служит гарантированная доставка сообщения, обеспечиваемая некоторыми протоколами — как бы ни развивались события, если сетевое соединение между машинами будет работать, посланное с одной стороны (процессом на первой машине) сообщение будет доставлено другой стороне (процессу на второй машине).

В классическом подходе к проверке на моделях проверяемые свойства формализуются в виде формул так называемых **временных логик**. Их общей чертой является наличие операторов "всегда в будущем" и "когда-то в будущем". Заметим, что второй оператор может быть выражен с помощью первого и отрицания — то, что некоторое свойство когда-то будет выполнено, эквивалентно тому, что отрицание этого свойства не будет выполнено всегда. Свойства безопасности легко записываются в виде "всегда будет выполнено отрицание нежелательного свойства", а свойства живости — в виде "когда-то обязательно будет выполнено желаемое".

Проверяемая программа в классическом подходе моделируется при помощи конечного автомата. Проверка, выполняемая автоматически, состоит в том, что для всех достижимых при работе системы состояний этого автомата проверяется нужное свойство. Если оно оказывается выполненным, выдается сообщение об успешности проверки, если нет — выдается трасса, последовательность выполнения отдельных шагов программы, моделируемых переходами автомата, приводящая из начального состояния в такое, в котором нужное свойство нарушается. Эта трасса используется для

анализа происходящего и исправления либо программы, либо модели, если ошибка находится в ней.

Основная проблема этого подхода — огромное, а часто и бесконечное, количество состояний в моделях, достаточно хорошо отражающих поведение реальных программ. Для борьбы с комбинаторным взрывом состояний применяются различные методы оптимизации представления автомата, выделения и поиска состояний, существенных для выполнения проверяемого свойства.

2 ПРАКТИЧЕСКАЯ РАЗДЕЛ

ОГЛАВЛЕНИЕ

Лабораторная работа №1

Тема
КЛАССЫ.

Цель: *сформировать умения, необходимые для создания классов и использования: ограничений доступа, переменных и констант, конструкторов, методов, перегрузки, логических блоков, классов-шаблонов и методов-шаблонов.*

Основное содержание работы: *написать программу, в которой реализовано решение поставленной задачи с использованием целевой установки.*

Порядок работы

1. Изучить материалы.
2. Выполнить задания, соответствующие собственному варианту.

Задания лабораторной работы

Задание 1

1. Определить класс **`Vector`** размерности `n`. Реализовать методы сложения, вычитания, умножения, инкремента, декремента, индексирования. Определить массив из m объектов. Каждую из пар векторов передать в методы, возвращающие их скалярное произведение и длины. Вычислить и вывести углы между векторами.

2. Определить класс **`Vector`** размерности `n`. Определить несколько конструкторов. Реализовать методы для вычисления модуля вектора, скалярного произведения, сложения, вычитания, умножения на константу. Объявить массив объектов. Написать метод, который для заданной пары векторов будет определять, являются ли они коллинеарными или ортогональными.

3. Определить класс **`Vector`** в `R3`. Реализовать методы для проверки векторов на ортогональность, проверки пересечения неортогональных векторов, сравнения векторов. Создать массив из m объектов. Определить, какие из векторов компланарны.

4. Определить класс **`Matrix`** размерности (`n x n`). Класс должен содержать несколько конструкторов. Реализовать методы для сложения, вычитания, умножения матриц. Объявить массив объектов. Создать методы, вычисляющие первую и вторую нормы матрицы

$$\| a \|_1 = \max_{1 \le i \le n} \sum_{j=1}^{n} (a_{ij}), \|a\|_2 = \max_{1 \le j \le n} \sum_{i=1}^{n} (a_{ij}).$$

Определить, какая из матриц имеет наименьшую первую и вторую нормы.

5. Определить класс **`Matrix`** размерности (`m x n`). Класс должен содержать несколько конструкторов. Объявить массив объектов. Передать объекты в метод, меняющий местами строки с максимальным и минимальным элементами k-го столбца. Создать метод, который изменяет i-ю матрицу путем возведения ее в квадрат.

6. Определить класс **`Цепная дробь`**

$$A = a_0 + \cfrac{x}{a_1 + \cfrac{x}{a_2 + \cfrac{x}{a_3 + \ldots}}}$$

Определить методы сложения, вычитания, умножения, деления.

Вычислить значение для заданного `n, x, a[n]`.

7. Определить класс **Дробь** в виде пары `(m,n)`. Класс должен содержать несколько конструкторов. Реализовать методы для сложения, вычитания, умножения и деления дробей. Объявить массив из `k` дробей, ввести/вывести значения для массива дробей. Создать массив объектов и передать его в метод, который изменяет каждый элемент массива с четным индексом путем добавления следующего за ним элемента массива.

8. Определить класс **Complex**. Класс должен содержать несколько конструкторов. Реализовать методы для сложения, вычитания, умножения, деления, присваивания комплексных чисел. Создать два вектора размерности n из комплексных координат. Передать их в метод, который выполнит их сложение.

9. Определить класс **Квадратное уравнение**. Класс должен содержать несколько конструкторов. Реализовать методы для поиска корней, экстремумов, а также интервалов убывания/возрастания. Создать массив объектов и определить наибольшие и наименьшие по значению корни.

10. Определить класс **Булева матрица** (**BoolMatrix**) размерности (`n x m`). Класс должен содержать несколько конструкторов. Реализовать методы для логического сложения (дизъюнкции), умножения и инверсии матриц. Реализовать методы для подсчета числа единиц в матрице и упорядочения строк в лексикографическом порядке.

11. Построить класс **Булев вектор** (**BoolVector**) размерности `n`. Определить несколько конструкторов. Реализовать методы для выполнения поразрядных конъюнкции, дизъюнкции и отрицания векторов, а также подсчета числа единиц и нулей в векторе.

12. Определить класс **Множество символов** мощности n. Написать несколько конструкторов. Реализовать методы для определения принадлежности заданного элемента множеству; пересечения, объединения, разности двух множеств. Создать методы сложения, вычитания, умножения (пересечения), индексирования, присваивания. Создать массив объектов и передавать пары объектов в метод другого класса, который строит множество, состоящее из элементов, входящих только в одно из заданных множеств.

13. Определить класс **Polynom** степени `n.` Создать методы для сложения и умножения объектов. Объявить массив из m полиномов и передать его в метод, вычисляющий сумму полиномов массива. Определить класс **Rational Polynom** с полем типа **Polynom**. Определить метод для сложения:

$$R = \frac{p_1(x)}{Q_1(x)} + \frac{p_2(x)}{Q_2(x)}$$

и методы для ввода/вывода.

14. Определить класс **Нелинейное уравнение** для двух переменных. Написать несколько конструкторов. Создать методы для сложения и умножения объектов. Реализовать метод определения корней методом бисекции.

Задание 2

Создать классы, спецификации которых приведены ниже. Определить конструктор и методы **set**Тип**(), get**Тип**(), show**Информ**().** Реализовать в классе, создающем массив объектов. Задать критерий выбора данных и вывести эти данные на консоль.

1. **Student**: id, Фамилия, Имя, Отчество, Дата рождения, Адрес, Телефон, Факультет, Курс, Группа.

Создать массив объектов. Вывести:

a) список студентов заданного факультета;

b) списки студентов для каждого факультета и курса;

c) список студентов, родившихся после заданного года;

d) список учебной группы.

2. **Customer**: id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Номер банковского счета.

Создать массив объектов. Вывести:

a) список покупателей в алфавитном порядке;

b) список покупателей, у которых номер кредитной карточки находится в заданном интервале.

3. **Patient**: id, Фамилия, Имя, Отчество, Адрес, Телефон, Номер медицинской карты, диагноз.

Создать массив объектов. Вывести:

a) список пациентов, имеющих данный диагноз;

b) список пациентов, номер медицинской карты у которых находится в заданном интервале.

4. **Abiturient**: id, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки.

Создать массив объектов. Вывести:

a) список абитуриентов, имеющих неудовлетворительные оценки;

b) список абитуриентов, средний балл у которых выше заданного;

c) выбрать заданное число n абитуриентов, имеющих самый высокий средний балл (вывести также полный список абитуриентов, имеющих полупроходной балл).

5. **Book**: id, Название, Автор(ы), Издательство, Год издания, Количество страниц, Цена, Переплет.

Создать массив объектов. Вывести:

a) список книг заданного автора;

b) список книг, выпущенных заданным издательством;

c) список книг, выпущенных после заданного года.

6. **`House`**: id, Номер квартиры, Площадь, Этаж, Количество комнат, Улица, Тип здания, Срок эксплуатации.

Создать массив объектов. Вывести:

a) список квартир, имеющих заданное число комнат;

b) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;

c) список квартир, имеющих площадь, превосходящую заданную.

7. **`Phone`**: id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Дебет, Кредит, Время городских и междугородных разговоров.

Создать массив объектов. Вывести:

a) сведения об абонентах, у которых время внутригородских
разговоров превышает заданное;

b) сведения об абонентах, которые пользовались междугородной
связью;

c) сведения об абонентах в алфавитном порядке.

8. **`Car`**: id, Марка, Модель, Год выпуска, Цвет, Цена, Регистрационный номер.

Создать массив объектов. Вывести:

a) список автомобилей заданной марки;

b) список автомобилей заданной модели, которые эксплуатируются больше n лет;

c) список автомобилей заданного года выпуска, цена которых
больше указанной.

9. **`Product`**: id, Наименование, UPC, Производитель, Цена, Срок хранения, Количество.

Создать массив объектов. Вывести:

a) список товаров для заданного наименования;

b) список товаров для заданного наименования, цена которых
не превосходит заданную;

c) список товаров, срок хранения которых больше заданного.

10. **`Train`**: Пункт назначения, Номер поезда, Время отправления, Число мест (общих, купе, плацкарт, люкс).

Создать массив объектов. Вывести:

a) список поездов, следующих до заданного пункта назначения;

b) список поездов, следующих до заданного пункта назначения
и отправляющихся после заданного часа;

c) список поездов, отправляющихся до заданного пункта назначения и имеющих общие места.

11. **`Bus`**: Фамилия и инициалы водителя, Номер автобуса, Номер маршрута, Марка, Год начала эксплуатации, Пробег.

Создать массив объектов. Вывести:

a) список автобусов для заданного номера маршрута;

b) список автобусов, которые эксплуатируются больше 10 лет;

c) список автобусов, пробег у которых больше 100000 км.

12. **Aeroflot**: Пункт назначения, Номер рейса, Тип самолета, Время вылета, Дни недели.

Создать массив объектов. Вывести:

a) список рейсов для заданного пункта назначения;

b) список рейсов для заданного дня недели;

c) список рейсов для заданного дня недели, время вылета для которых больше заданного.

Порядок подготовки отчета и сдачи лабораторной работы.

Подготовьте отчет, включающий

1. Тему лабораторной работы № 1.

2. Содержание отчета

2а. Условие задачи (с отражением всех параметров графических задач)

2б. Текст программы.

2в. Выводы.

Для сдачи работы необходимо:

1. предоставить отчет

2. ответить на вопросы преподавателя по теме работы. Для ответа на вопросы необходимо знать теорию.

Лабораторная работа №2

Тема

ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ И ОПРЕДЕЛЕНИЕ ОПЕРАЦИЙ НАД ОБЪЕКТАМИ КЛАССОВ

Цель: *изучить принципы и механизмы установления связей между объектами в объектно-ориентированном программировании, включая ассоциацию, агрегацию и композицию; научиться проектировать и реализовывать классы с учетом этих типов связей, а также понимать различия между ними; сформировать умения, необходимые для создания классов с реализацией идей наследования, ключевых слов super и this, полиморфизма, переопределения и перегрузки методов.*

Основное содержание работы: *написать программу, в которой реализовано решение поставленной задачи с использованием целевой установки.*

Порядок работы

1. Изучить материалы.

2. Выполнить задания, соответствующие собственному варианту.

Задания лабораторной работы

Задание 1

Реализовать агрегирование. При создании класса агрегируемый класс объявляется как атрибут (локальная переменная, параметр метода).

1. Создать объект класса **Строка**, используя классы **Слово**, **Символ**.

2. Создать объект класса **Абзац**, используя класс **Строка**.

3. Создать объект класса **Страница**, используя класс **Абзац**.

4. Создать объект класса **Текст**, используя классы **Страница**, **Слово**.

5. Создать объект класса **Абзац**, используя класс **Слово**.

6. Создать объект класса **Страница**, используя класс **Слово**.

7. Создать объект класса **Страница**, используя классы **Строка**, **Слово**.

8. Создать объект класса **Текст**, используя класс **Абзац**.

9. Создать объект класса **Автомобиль**, используя класс **Колесо**.

10. Создать объект класса **Самолет**, используя класс **Крыло**.

11. Создать объект класса **Беларусь**, используя класс **Область**.

12. Создать объект класса **Планета**, используя класс **Материк**.

13. Создать объект класса **Звездная система**, используя классы **Планета**, **Звезда**, **Луна**.

14. Создать объект класса **Компьютер**, используя классы **Винчестер**, **Дисковод**, **ОЗУ**.

Задание 2

Построить модель программной системы с применением отношений (обобщения, ассоциации, использования, реализации) между классами. Задать атрибуты и методы классов. Ввести (если необходимо) дополнительные классы.

1. Система **Факультатив**. **Преподаватель** объявляет запись на **Курс**. **Студент** записывается на **Курс**, обучается и по окончании **Преподаватель** выставляет **Оценку**, которая сохраняется в **Архиве**. **Студентов**, **Преподавателей** и **Курсов** при обучении может быть несколько.

2. Система **Платежи**. **Клиент** имеет **Счет** в банке и **Кредитную Карту (КК)**. **Клиент** может оплатить **Заказ**, сделать платеж на другой **Счет**, заблокировать **КК** и аннулировать **Счет**. **Администратор** может заблокировать **КК** за превышение кредита.

3. Система **Больница**. **Пациенту** назначается лечащий **Врач**. **Врач** может сделать назначение **Пациенту** (процедуры, лекарства, операции). **Медсестра** или другой **Врач** выполняют назначение. **Пациент** может быть выписан из **Больницы** по окончании лечения, при нарушении режима или иных обстоятельствах.

4. Система **Вступительные экзамены**. **Абитуриент** регистрируется на **Факультет**, сдает **Экзамены**. **Преподаватель** выставляет **Оценку**. Система подсчитывает средний балл и определяет **Абитуриентов**, зачисленных в учебное заведение.

5. Система **Библиотека**. **Читатель** оформляет **Заказ** на **Книгу**. Система осуществляет поиск в **Каталоге**. **Библиотекарь** выдает **Читателю Книгу** на абонемент или в читальный зал. При невозвращении **Книги Читателем** он может быть занесен **Администратором** в “черный список”.

6. Система **Конструкторское бюро**. **Заказчик** представляет **Техническое Задание (ТЗ)** на проектирование многоэтажного **Дома**. **Конструктор** регистрирует **ТЗ**, определяет стоимость проектирования и строительства, выставляет **Заказчику**

Счет за проектирование и создает **Бригаду Конструкторов** для выполнения Проекта.

7. Система **Телефонная станция**. **Абонент** оплачивает **Счет** за разговоры и **Услуги**, может попросить **Администратора** сменить номер и отказаться от услуг. **Администратор** изменяет номер, **Услуги** и временно отключает **Абонента** за неуплату.

8. Система **Автобаза**. **Диспетчер** распределяет заявки на **Рейсы** между **Водителями** и назначает для этого **Автомобиль**. **Водитель** может сделать заявку на ремонт. **Диспетчер** может отстранить **Водителя** от работы. **Водитель** делает отметку о выполнении **Рейса** и состоянии **Автомобиля**.

9. Система **Интернет-магазин**. **Администратор** добавляет информацию о **Товаре**. **Клиент** делает и оплачивает **Заказ** на **Товары**. **Администратор** регистрирует **Продажу** и может занести неплательщиков в “черный список”.

10. Система **Железнодорожная касса**. **Пассажир** делает **Заявку**

на станцию назначения, время и дату поездки. Система регистрирует **Заявку** и осуществляет поиск подходящего **Поезда**. **Пассажир** делает выбор **Поезда** и получает **Счет** на оплату. **Администратор** вводит номера **Поездов**, промежуточные и конечные станции, цены.

11. Система **Городской транспорт**. На **Маршрут** назначаются **Автобус**, **Троллейбус** или **Трамвай**. Транспортные средства должны двигаться с определенным для каждого **Маршрута** интервалом. При поломке на **Маршрут** должен выходить резервный транспорт или увеличиваться интервал движения.

12. Система **Аэрофлот**. **Администратор** формирует летную **Бригаду** (пилоты, штурман, радист, стюардессы) на **Рейс**. Каждый **Рейс** выполняется **Самолетом** с определенной вместимостью и дальностью полета. **Рейс** может быть отменен из-за погодных условий в **Аэропорту** отлета или назначения. **Аэропорт** назначения может быть изменен в полете из-за технических неисправностей, о которых сообщил командир.

Задание 3

1. Реализуйте класс **Vector2D**, представляющий вектор на плоскости. Перегрузите оператор **+** для сложения двух векторов.

2. Создайте класс **ComplexNumber** и перегрузите оператор **==** для сравнения двух комплексных чисел.

3. Напишите класс **Matrix**, представляющий матрицу. Перегрузите оператор ***** для умножения двух матриц.

4. Создайте класс **String**, который управляет динамическим массивом символов. Перегрузите оператор **[]** для доступа к отдельным символам строки.

5. Создайте класс **Time**, представляющий время (часы, минуты, секунды). Перегрузите оператор **-** для вычитания одного времени из другого.

6. Напишите класс **Counter**, представляющий счётчик. Перегрузите операторы **++** и **--** для увеличения и уменьшения значения счётчика.

7. Создайте класс **Date**, представляющий дату (день, месяц, год). Перегрузите операторы **<** и **>** для сравнения двух дат.

8. Напишите класс **Polynomial**, представляющий полином. Перегрузите оператор ***** для умножения двух полиномов.

9. Создайте класс **Money**, представляющий денежную сумму. Перегрузите оператор **/** для деления суммы на число.

10. Создайте класс **Function**, представляющий математическую функцию. Перегрузите оператор **()** для вызова функции с одним аргументом.

Порядок подготовки отчета и сдачи лабораторной работы.

Подготовьте отчет, включающий

1. Тему лабораторной работы № 2.
2. Содержание отчета

2а. Условие задачи (с отражением всех параметров графических задач)

2б. Текст программы.

2в. Выводы.

Для сдачи работы необходимо:

1. предоставить отчет
2. ответить на вопросы преподавателя по теме работы. Для ответа на вопросы необходимо знать теорию.

Лабораторная работа №3

Тема

ПРОИЗВОДНЫЕ КЛАССЫ

Цель: *сформировать умения, необходимые для создания абстрактных классов и методов, реализации интерфейсов и работы с пакетами.*

Основное содержание работы: *написать программу, в которой реализовано решение поставленной задачи с использованием целевой установки.*

Порядок работы

1. Изучить материалы
2. Выполнить задания, соответствующие собственному варианту.

Задания лабораторной работы

Задание 1

Реализовать абстрактные классы или интерфейсы, а также наследование и полиморфизм для следующих классов:

1. Абстрактный класс **Книга** (Шифр, Автор, Название, Год, Издательство). Подклассы **Справочник** и **Энциклопедия**.

2. **interface Abiturient ← abstract class Student ←class Student Of Faculty**.

3. **interface Сотрудник ← class Инженер ← class Руководитель**.

4. **interface Учебное Заведение ← class Колледж ←class Университет**.

5. **interface Здание ← abstract class Общественное Здание ← class Театр**.

6. **interface Mobile ← abstract class Siemens Mobile← class Model**.

7. **interface Корабль ← abstract class Военный Корабль ← class Авианосец**.

8. **interface Врач ← class Хирург ← class Нейрохирург**.

9. **interface Корабль ← class Грузовой Корабль ←class Танкер**.

10. **interface Диск ← abstract class Директория ←class Файл**.

Задание 2

В следующих заданиях требуется создать *суперкласс (абстрактный класс, интерфейс)* и определить общие методы для данного класса. Создать подклассы, в которых добавить специфические свойства и методы. Часть методов переопределить. Создать массив объектов суперкласса и заполнить объектами подклассов. Объекты подклассов идентифицировать конструктором по имени или идентификационному номеру. Использовать объекты подклассов для моделирования реальных ситуаций и объектов.

1. Создать суперкласс **Транспортное средство** и подклассы **Автомобиль**, **Велосипед**, **Повозка**. Подсчитать время и стоимость перевозки пассажиров и грузов каждым транспортным средством.

2. Создать суперкласс **Грузоперевозчик** и подклассы **Самолет**, **Поезд**, **Автомобиль**. Определить время и стоимость перевозки для указанных городов и расстояний.

3. Создать суперкласс **Пассажироперевозчик** и подклассы **Самолет**, **Поезд**, **Автомобиль**. Определить время и стоимость передвижения.

4. Создать суперкласс **Учащийся** и подклассы **Школьник** и **Студент**. Создать массив объектов суперкласса и заполнить этот массив объектами. Показать отдельно студентов и школьников.

5. Создать суперкласс **Музыкальный инструмент** и классы **Ударный, Струнный**, **Духовой**. Создать массив объектов **Оркестр**. Выдать состав оркестра.

6. Определить суперкласс **Множество** и подкласс **Кольцо** (операции сложения и умножения, обе коммутативные и ассоциативные, связанные законом дистрибутивности). Ввести кольца целых чисел многочленов, систему классов целых чисел, сравнимых по модулю. Кольцо является полем, если в нем определена операция деления, кроме деления на ноль.

7. Создать абстрактный класс **Работник фирмы** и подклассы **Менеджер**, **Аналитик**, **Программист**, **Тестировщик**, **Дизайнер**.

8. Создать суперкласс **Домашнее животное** и подклассы **Собака**, **Кошка**, **Попугай**. С помощью конструктора установить имя каждого животного и его характеристики.

9. Создать базовый класс **`Садовое дерево`** и производные классы **`Яблоня`**, **`Вишня`**, **`Груша`** и другие. С помощью конструктора автоматически становить номер каждого дерева. Принять решение о пересадке каждого дерева в зависимости от возраста и плодоношения.

Порядок подготовки отчета и сдачи лабораторной работы.

Подготовьте отчет, включающий

1. Тему лабораторной работы № 3.
2. Содержание отчета

2а. Условие задачи (с отражением всех параметров графических задач)

2б. Текст программы.

2в. Выводы.

Для сдачи работы необходимо:

1. предоставить отчет
2. ответить на вопросы преподавателя по теме работы. Для ответа на вопросы необходимо знать теорию.

Лабораторная работа №4

Тема

ФАЙЛЫ. ПОТОКИ ВВОДА/ВЫВОДА

Цель: *сформировать умения, необходимые для работы с файлами, потоками ввода-вывода, предопределенными потоками, механизмом сериализации объектов класса.*

Основное содержание работы: *написать программу, в которой реализовано решение поставленной задачи с использованием целевой установки.*

Порядок работы

1. Изучить материалы.
2. Выполнить задания, соответствующие собственному варианту.

Задания лабораторной работы

Задание 1

В следующих заданиях требуется ввести последовательность строк из текстового потока и выполнить указанные действия. При этом могут рассматриваться два варианта:

- каждая строка состоит из одного слова;
- каждая строка состоит из нескольких слов.

Имена входного и выходного файлов, а также абсолютный путь к ним могут быть введены как параметры командной строки или храниться в файле.

1. В каждой строке найти и удалить заданную подстроку.
2. В каждой строке стихотворения А. Блока найти и заменить заданную подстроку на подстроку иной длины.
3. В каждой строке найти слова, начинающиеся с гласной буквы.
4. Найти и вывести слова текста, для которых последняя буква одного слова совпадает с первой буквой следующего слова.
5. Найти в строке наибольшее число цифр, идущих подряд.

6. В каждой строке стихотворения С. Есенина подсчитать частоту повторяемости каждого слова из заданного списка и вывести эти слова в порядке возрастания частоты повторяемости.

7. В каждом слове сонета В. Шекспира заменить первую букву слова на прописную.

8. Определить частоту повторяемости букв и слов в стихотворении А. Пушкина.

Задание 2

Выполнить задания из варианта В лабораторной работы №1, сохраняя объекты приложения в одном или нескольких файлах с применением механизма сериализации. Объекты могут содержать поля, помеченные как `static`, а также `transient`. Для изменения информации и извлечения информации в файле создать специальный класс-коннектор с необходимыми для выполнения этих задач методами.

Задание 3

1. Реализуйте программу, которая записывает данные в файл и использует функции `seekg` и `seekp` для чтения и записи данных на произвольных позициях.

2. Напишите программу, которая создает несколько потоков, каждый из которых записывает данные в общий файл. Используйте мьютексы для синхронизации доступа к файлу.

3. Создайте программу, которая записывает объекты класса `Rectangle` (с полями `width` и `height`) в бинарный файл и затем считывает их обратно, демонстрируя работу с бинарным форматом.

4. Создайте программу, которая использует несколько потоков для записи данных в один файл, используя `synchronized` блоки для обеспечения потокобезопасности.

5. Реализуйте программу, которая форматирует вывод объектов класса `Employee` (с полями `name`, `position`, `salary`) с использованием `String.format()` и считывает их из файла.

6. Напишите программу, которая использует `BufferedReader` и `BufferedWriter` для чтения и записи данных в `CSV` файл, управляя позициями для обработки различных строк и колонок.

7. Напишите программу, которая считывает данные о студенте (имя, возраст, средний балл) из текстового файла и выводит их на экран в форматированном виде.

8. Напишите программу, которая считывает данные из файла, где каждая строка имеет разный формат (например, сначала целое число, затем строка, затем вещественное число) и выводит данные в форматированном виде.

9. Создайте программу, которая записывает таблицу чисел в файл, форматируя её в виде таблицы, и затем считывает таблицу и выводит её на экран.

10. Реализуйте программу, которая записывает данные в файл в формате `XML`, используя форматированный вывод для обеспечения правильного форматирования `XML`.

Порядок подготовки отчета и сдачи лабораторной работы.

Подготовьте отчет, включающий

1. Тему лабораторной работы № 4.

2. Содержание отчета

2а. Условие задачи (с отражением всех параметров графических задач)

2б. Текст программы.

2в. Выводы.

Для сдачи работы необходимо:

1. предоставить отчет

2. ответить на вопросы преподавателя по теме работы. Для ответа на вопросы необходимо знать теорию.

Лабораторная работа №5

Тема

ШАБЛОНЫ ФУНКЦИЙ И КЛАССОВ

Цель: *сформировать умения, необходимые для работы с шаблонами функций и классов.*

Основное содержание работы: *написать программу, в которой реализовано решение поставленной задачи с использованием целевой установки.*

Порядок работы

1. Изучить материалы.

2. Выполнить задания, соответствующие собственному варианту.

Задания лабораторной работы

Задание 1

1. Создайте шаблон функции `compareValues`, которая возвращает `true`, если два значения одинаковы. Переопределите эту функцию для строк, чтобы сравнение было нечувствительно к регистру.

2. Реализуйте шаблон функции `sumValues`, который складывает два значения числового типа. Переопределите функцию для строк, чтобы она объединяла строки.

3. Напишите шаблон функции `averageValue`, которая принимает массив чисел и возвращает их среднее значение. Переопределите функцию для массива строк, чтобы возвращать среднюю длину строки.

4. Реализуйте шаблон функции `maxValue`, который находит максимальное из двух чисел. Переопределите функцию для строк, чтобы находить строку с наибольшей длиной.

5. Напишите шаблон функции `sortArray`, которая сортирует массив чисел по возрастанию. Переопределите функцию для строк, чтобы сортировка проводилась по длине строки.

6. Реализуйте шаблон функции `minValue`, который находит минимальное из двух значений числового типа. Переопределите функцию для строк, чтобы находить строку с наименьшей длиной.

7. Напишите шаблон функции `copyArray`, который копирует массив любого типа. Переопределите функцию для массивов строк, чтобы результат копирования был в верхнем регистре.

8. Реализуйте шаблон функции `findFirst`, которая находит первое вхождение элемента в массиве. Переопределите функцию для строк, чтобы находить первую строку, которая начинается с определенного символа.

9. Реализуйте шаблон функции `` `subtractValues` ``, который вычисляет разность двух чисел. Переопределите функцию для строк, чтобы возвращать разницу в длине строк.

10. Реализуйте шаблон функции `` `transformArray` ``, которая применяет заданную функцию ко всем элементам массива. Переопределите для строки, чтобы функция удаляла пробелы.

Задание 2

1. Создайте шаблон класса `SortableList`, который хранит список элементов любого типа. Реализуйте метод сортировки списка. Переопределите этот шаблон класса для сортировки строк в алфавитном порядке.

2. Создайте шаблон класса `Filter`, который фильтрует элементы списка любого типа по определенному критерию. Переопределите этот шаблон для фильтрации строк по длине.

3. Реализуйте шаблонный класс `Printer`, который выводит элементы типа `T`. Переопределите этот класс для типа `int`, чтобы метод вывода форматировал числа с двумя знаками после запятой.

4. Создайте шаблонный класс `Transformer`, который выполняет преобразование значения типа T. Переопределите этот класс для типа `std::string`, чтобы преобразование заменяло пробелы на подчеркивания.

5. Создайте шаблонный класс `Validator`, который проверяет данные на соответствие некоторым критериям. Переопределите этот класс для типа `std::string`, чтобы он проверял строки на соответствие формату электронной почты.

6. Создайте шаблонный класс `ValueHolder`, который хранит одно значение типа `T` и имеет методы для установки и получения этого значения. Переопределите этот класс для типа `std::string`, чтобы он также поддерживал метод для преобразования строки в верхний регистр.

7. Создайте шаблонный класс `Comparer`, который сравнивает два значения типа `T`. Переопределите этот класс для типа `std::string`, чтобы метод сравнения игнорировал регистр символов.

8. Создайте шаблонный класс `SequenceGenerator`, который генерирует последовательность значений типа `T`. Переопределите этот класс для типа `int`, чтобы генерация происходила по арифметической прогрессии.

9. Реализуйте шаблонный класс `RangeChecker`, который проверяет, попадает ли значение типа `T` в заданный диапазон. Переопределите этот класс для типа `std::string`, чтобы диапазон был задан по длине строки.

10. Реализуйте шаблонный класс `Formatter`, который форматирует данные типа `T` в строку. Переопределите этот класс для типа `int`, чтобы метод форматирования выводил число в шестнадцатеричном формате.

Задание 3

1. Напишите программу, которая принимает на вход массив чисел, сохраняет его в `std::vector`, сортирует числа по возрастанию и выводит их.

2. Реализуйте программу, которая принимает текст, разбивает его на слова и подсчитывает частоту каждого слова с использованием `std::map`.

3. Напишите программу, которая принимает массив чисел, сохраняет его в `std::set` и выводит уникальные элементы массива.

4. Создайте `std::list` из строковых значений, удалите первый и последний элементы, затем выведите оставшиеся элементы на экран.

5. Напишите программу, которая принимает строку, сохраняет каждый символ в `std::stack`, а затем выводит символы в обратном порядке.

6. Напишите программу, которая принимает вектор целых чисел, сортирует его по возрастанию с использованием `std::sort` и выводит результат.

7. Реализуйте программу, которая находит первое вхождение заданного числа в `std::vector` и выводит его индекс.

8. Напишите программу, которая использует `std::transform` для преобразования всех символов строки в верхний регистр.

9. Реализуйте программу, которая находит сумму всех элементов вектора целых чисел с использованием `std::accumulate`.

10. Реализуйте программу, которая удаляет подряд идущие дублирующиеся элементы вектора целых чисел.

Порядок подготовки отчета и сдачи лабораторной работы.

Подготовьте отчет, включающий

1. Тему лабораторной работы № 5.

2. Содержание отчета

2а. Условие задачи (с отражением всех параметров графических задач)

2б. Текст программы.

2в. Выводы.

Для сдачи работы необходимо:

1. предоставить отчет

2. ответить на вопросы преподавателя по теме работы. Для ответа на вопросы необходимо знать теорию.

Лабораторная работа №6

Тема

ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ

Цель: *сформировать умения, необходимые для работы с иерархией исключительных ситуаций и их обработки, оператором* `throw` *для генерации исключений, ключевым словом* `finally`.

Основное содержание работы: *написать программу, в которой реализовано решение поставленной задачи с использованием целевой установки.*

Порядок работы

1. Изучить материалы
2. Выполнить задания, соответствующие собственному варианту.

Задания лабораторной работы

Задание 1

Выполнить задания на основе задания 1 лабораторной работы №1, контролируя состояние потоков ввода/вывода. При возникновении ошибок, связанных с корректностью выполнения математических операций, генерировать и обрабатывать исключительные ситуации. Предусмотреть обработку исключений, возникающих при: нехватке памяти, отсутствии требуемой записи (объекта) в файле, недопустимом значении поля и т.д.

Задание 1 лабораторной работы №1.

1. Определить класс **`Vector`** размерности `n`. Реализовать методы сложения, вычитания, умножения, инкремента, декремента, индексирования. Определить массив из m объектов. Каждую из пар векторов передать в методы, возвращающие их скалярное произведение и длины. Вычислить и вывести углы между векторами.

2. Определить класс **`Vector`** размерности `n`. Определить несколько конструкторов. Реализовать методы для вычисления модуля вектора, скалярного произведения, сложения, вычитания, умножения на константу. Объявить массив объектов. Написать метод, который для заданной пары векторов будет определять, являются ли они коллинеарными или ортогональными.

3. Определить класс **`Vector`** в `R3`. Реализовать методы для проверки векторов на ортогональность, проверки пересечения неортогональных векторов, сравнения векторов. Создать массив из m объектов. Определить, какие из векторов компланарны.

4. Определить класс **`Matrix`** размерности (`n x n`) **.** Класс должен содержать несколько конструкторов. Реализовать методы для сложения, вычитания, умножения матриц. Объявить массив объектов. Создать методы, вычисляющие первую и вторую нормы матрицы

$$\| a \|_1 = \max_{1 \le i \le n} \sum_{j=1}^{n} (a_{ij}), \|a\|_2 = \max_{1 \le j \le n} \sum_{i=1}^{n} (a_{ij}).$$

Определить, какая из матриц имеет наименьшую первую и вторую нормы.

5. Определить класс **Matrix** размерности (m x n). Класс должен содержать несколько конструкторов. Объявить массив объектов. Передать объекты в метод, меняющий местами строки с максимальным и минимальным элементами k-го столбца. Создать метод, который изменяет i-ю матрицу путем возведения ее в квадрат.

6. Определить класс **Цепная дробь**

$$A = a_0 + \cfrac{x}{a_1 + \cfrac{x}{a_2 + \cfrac{x}{a_3 + \ldots}}}$$

Определить методы сложения, вычитания, умножения, деления.

Вычислить значение для заданного n, x, a[n].

7. Определить класс **Дробь** в виде пары (m, n). Класс должен содержать несколько конструкторов. Реализовать методы для сложения, вычитания, умножения и деления дробей. Объявить массив из k дробей, ввести/вывести значения для массива дробей. Создать массив объектов и передать его в метод, который изменяет каждый элемент массива с четным индексом путем добавления следующего за ним элемента массива.

8. Определить класс **Complex**. Класс должен содержать несколько конструкторов. Реализовать методы для сложения, вычитания, умножения, деления, присваивания комплексных чисел. Создать два вектора размерности n из комплексных координат. Передать их в метод, который выполнит их сложение.

9. Определить класс **Квадратное уравнение**. Класс должен содержать несколько конструкторов. Реализовать методы для поиска корней, экстремумов, а также интервалов убывания/возрастания. Создать массив объектов и определить наибольшие и наименьшие по значению корни.

10. Определить класс **Булева матрица** (**BoolMatrix**) размерности (n x m). Класс должен содержать несколько конструкторов. Реализовать методы для логического сложения (дизъюнкции), умножения и инверсии матриц. Реализовать методы для подсчета числа единиц в матрице и упорядочения строк в лексикографическом порядке.

11. Построить класс **Булев вектор** (**BoolVector**) размерности n. Определить несколько конструкторов. Реализовать методы для выполнения поразрядных конъюнкции, дизъюнкции и отрицания векторов, а также подсчета числа единиц и нулей в векторе.

12. Определить класс **Множество символов** мощности n. Написать несколько конструкторов. Реализовать методы для определения принадлежности заданного элемента множеству; пересечения, объединения, разности двух множеств. Создать методы сложения, вычитания, умножения (пересечения), индексирования, присваивания. Создать массив объектов и передавать пары объектов в метод другого класса, который строит множество, состоящее из элементов, входящих только в одно из заданных множеств.

13. Определить класс **Polynom** степени n. Создать методы для сложения и умножения объектов. Объявить массив из m полиномов и передать его в метод,

вычисляющий сумму полиномов массива. Определить класс **Rational Polynom** с полем типа **Polynom**. Определить метод для сложения:

$$R = \frac{p_1(x)}{Q_1(x)} + \frac{p_2(x)}{Q_2(x)}$$

и методы для ввода/вывода.

14. Определить класс **Нелинейное уравнение** для двух переменных. Написать несколько конструкторов. Создать методы для сложения и умножения объектов. Реализовать метод определения корней методом бисекции.

Задание 2

Выполнить задания из задания 2 лабораторной работы №2, реализуя собственные обработчики исключений и исключения ввода/вывода. Построить модель программной системы с применением отношений (обобщения, ассоциации, использования, реализации) между классами. Задать атрибуты и методы классов. Ввести (если необходимо) дополнительные классы.

1. Система **Факультатив**. **Преподаватель** объявляет запись на **Курс**. **Студент** записывается на **Курс**, обучается и по окончании **Преподаватель** выставляет **Оценку**, которая сохраняется в **Архиве**. **Студентов**, **Преподавателей** и **Курсов** при обучении может быть несколько.

2. Система **Платежи**. **Клиент** имеет **Счет** в банке и **Кредитную Карту (КК)**. **Клиент** может оплатить **Заказ**, сделать платеж на другой **Счет**, заблокировать **КК** и аннулировать **Счет**. **Администратор** может заблокировать **КК** за превышение кредита.

3. Система **Больница**. **Пациенту** назначается лечащий **Врач**. **Врач** может сделать назначение **Пациенту** (процедуры, лекарства, операции). **Медсестра** или другой **Врач** выполняют назначение. **Пациент** может быть выписан из **Больницы** по окончании лечения, при нарушении режима или иных обстоятельствах.

4. Система **Вступительные экзамены**. **Абитуриент** регистрируется на **Факультет**, сдает **Экзамены**. **Преподаватель** выставляет **Оценку**. Система подсчитывает средний балл и определяет **Абитуриентов**, зачисленных в учебное заведение.

5. Система **Библиотека**. **Читатель** оформляет **Заказ** на **Книгу**. Система осуществляет поиск в **Каталоге**. **Библиотекарь** выдает **Читателю Книгу** на абонемент или в читальный зал. При невозвращении **Книги Читателем** он может быть занесен **Администратором** в “черный список”.

6. Система **Конструкторское бюро**. **Заказчик** представляет **Техническое Задание (ТЗ)** на проектирование многоэтажного **Дома**. **Конструктор** регистрирует **ТЗ**, определяет стоимость проектирования и строительства, выставляет **Заказчику Счет** за проектирование и создает **Бригаду Конструкторов** для выполнения Проекта.

7. Система **Телефонная станция**. **Абонент** оплачивает **Счет** за разговоры и **Услуги**, может попросить **Администратора** сменить номер и отказаться от услуг. **Администратор** изменяет номер, **Услуги** и временно отключает **Абонента** за неуплату.

8. Система **Автобаза**. **Диспетчер** распределяет заявки на **Рейсы** между **Водителями** и назначает для этого **Автомобиль**. **Водитель** может сделать заявку на ремонт. **Диспетчер** может отстранить **Водителя** от работы. **Водитель** делает отметку о выполнении **Рейса** и состоянии **Автомобиля**.

9. Система **Интернет-магазин**. **Администратор** добавляет информацию о **Товаре**. **Клиент** делает и оплачивает **Заказ** на **Товары**. **Администратор** регистрирует **Продажу** и может занести неплательщиков в «черный список».

10. Система **Железнодорожная касса**. **Пассажир** делает **Заявку** на станцию назначения, время и дату поездки. Система регистрирует **Заявку** и осуществляет поиск подходящего **Поезда**. **Пассажир** делает выбор **Поезда** и получает **Счет** на оплату. **Администратор** вводит номера **Поездов**, промежуточные и конечные станции, цены.

11. Система **Городской транспорт**. На **Маршрут** назначаются **Автобус**, **Троллейбус** или **Трамвай**. Транспортные средства должны двигаться с определенным для каждого **Маршрута** интервалом. При поломке на **Маршрут** должен выходить резервный транспорт или увеличиваться интервал движения.

12. Система **Аэрофлот**. **Администратор** формирует летную **Бригаду** (пилоты, штурман, радист, стюардессы) на **Рейс**. Каждый **Рейс** выполняется **Самолетом** с определенной вместимостью и дальностью полета. **Рейс** может быть отменен из-за погодных условий в **Аэропорту** отлета или назначения. **Аэропорт** назначения может быть изменен в полете из-за технических неисправностей, о которых сообщил командир.

Задание 3

1. Создайте базовый класс исключения `BaseException`, и два производных класса: `FileException` и `DatabaseException`. Реализуйте методы для вывода подробных сообщений об ошибках, используя различные уровни иерархии.

2. Реализуйте класс `InputValidator`, который содержит методы для проверки ввода пользователя (например, проверка возраста, `Email`). В случае некорректного ввода выбрасывайте соответствующие исключения и обрабатывайте их.

3. Реализуйте программу, которая читает данные из файла, обрабатывает их и записывает результат в другой файл. Обрабатывайте исключения, связанные с отсутствием файлов, ошибками чтения/записи и обеспечьте корректное завершение работы программы.

4. Создайте класс исключения `CustomException` с дополнительным параметром (например, код ошибки). Продемонстрируйте использование этого класса для обработки различных ошибок в программе.

5. Реализуйте программу, которая парсит `JSON` данные. В случае обнаружения ошибок в данных выбрасывайте и обрабатывайте пользовательские исключения.

6. Реализуйте класс `SafeArray`, который позволяет работать с массивами. Реализуйте проверку и выброс исключений при попытке доступа за пределы массива.

7. Напишите программу, которая читает данные в различных форматах (например, `CSV`, `XML`) и обрабатывает исключения, возникающие при некорректных данных.

Порядок подготовки отчета и сдачи лабораторной работы.

Подготовьте отчет, включающий

1. Тему лабораторной работы № 6.
2. Содержание отчета

2а. Условие задачи (с отражением всех параметров графических задач)

2б. Текст программы.

2в. Выводы.

Для сдачи работы необходимо:

1. предоставить отчет
2. ответить на вопросы преподавателя по теме работы. Для ответа на вопросы необходимо знать теорию.

Лабораторная работа №7

Порядок работы

1. Изучить материалы
2. Выполнить задания, соответствующие собственному варианту.

Задания лабораторной работы

На выбор решить по две задачи каждой сложности (всего должно быть решено 6 задач) из источников

1. https://www.codewars.com/dashboard (уровни сложности 8, 6, 4)
2. https://www.hackerrank.com/dashboard

Задачи должны быть категории ООП.

Порядок подготовки отчета и сдачи лабораторной работы.

Подготовьте отчет, включающий

1. Тему лабораторной работы №7.
2. Содержание отчета

2а. Условие задачи (с отражением всех параметров графических задач)

2б. Текст программы.

2в. Выводы.

Для сдачи работы необходимо:

1. предоставить отчет
2. ответить на вопросы преподавателя по теме работы. Для ответа на вопросы необходимо знать теорию.

Лабораторная работа №8

Цель: *познакомиться с теорией `UML`-диаграмм; научиться проектировать классы и отношения между ними посредством диаграмм классов.*

Задание лабораторной работы

1) Для иерархии классов, построенной в лабораторных работах №2 и №3, выполнить построение диаграммы классов в программе `Dia` с учетов всех специфичных особенностей (наличие абстрактных классов и интерфейсов, наследование и т.д.).

2) Согласно своего номера варианта, выполнить индивидуальные задания, построив диаграмму класса для гипотетического проекта. Предусмотреть наличие различных отношений между классами (композиция, агрегация и т.д.), а также различные типы классов и интерфейсы. При необходимости разрешается использовать дополнительно к построенной диаграмме классов диаграммы других типов (последовательности- `sequence diagrams`, вариантов использования `use case diagrams` и т.д.).

Задания по вариантам

1. Карточная игра «Косынка».
2. Игра «Сокобан».
3. Игра «Тетрис».
4. Игра «Змейка».
5. Игра «Морской бой».
6. Игра «Зума».
7. Игра «Tower Defense».
8. Игра «Pac-Man»
9. Градостроительный симулятор.
10. Игра «Пинболл».

Порядок подготовки отчета и сдачи лабораторной работы.

Подготовьте отчет, включающий

1. Тему лабораторной работы №8.

2. Содержание отчета

2а. Условие задачи (с отражением всех параметров графических задач)

2б. Текст программы.

2в. Выводы.

Для сдачи работы необходимо:

1. предоставить отчет

2. ответить на вопросы преподавателя по теме работы. Для ответа на вопросы необходимо знать теорию.

Отчеты по всем работам хранятся студентом до получения зачета по дисциплине

3 РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

ВОПРОСЫ ДЛЯ ЭКЗАМЕНА

1. Основные концепции ООП: объекты, классы, наследование, инкапсуляция и полиморфизм.

2. Отличия ООП от процедурного и функционального программирования

3. Создание классов и объектов в `Python`. Конструктор `__init__`, атрибуты экземпляра и класса, методы. Примеры использования классов для моделирования реальных сущностей.

4. Создание классов и объектов в `Java`. Конструктор и деструктор, атрибуты и методы. Примеры использования классов для моделирования реальных сущностей.

5. Принцип инкапсуляции, модификаторы доступа (`public`, `protected`, `private`). Практика скрытия внутреннего состояния объектов и предоставления доступа через методы.

6. Основы наследования, создание подклассов и расширение функциональности базового класса.

7. Использование функций классов родителей и работа с многоуровневой иерархией классов.

8. Понятие полиморфизма, динамическое определение типов и переопределение методов в подклассах.

9. Использование абстрактных классов и интерфейсов. Декораторы в `Python`.

10. Статические и классовые методы, использование декораторов `@staticmethod` и `@classmethod`.

11. Работа с атрибутами класса и их отличия от атрибутов экземпляра.

12. Переопределение специальных методов (магических методов) для изменения поведения объектов: `__str__`, `__repr__`, `__len__`, `__getitem__`, `__call__` и другие. Работа с перегрузкой операторов.

13. Принципы композиции и агрегации для создания сложных объектов из простых.

14. Различия между наследованием и композицией, примеры их использования в `Python/Java`.

15. Создание пользовательских итераторов и генераторов в классах. Реализация методов `__iter__` и `__next__`. Работа с ленивыми вычислениями и генераторами.

16. Управление исключениями в ООП, создание пользовательских классов исключений.

17. Паттерны проектирования для устойчивых к ошибкам объектов.

18. Метаклассы и их роль в `Python`.

19. Создание и использование метаклассов для динамического изменения поведения классов.

20. Сохранение объектов в файлы и их восстановление (сериализация и десериализация) с использованием модулей `pickle`, `json`.

21. Основные паттерны проектирования: `Singleton`, `Factory`, `Observer` и др.

22. Методики тестирования и отладки ООП-кода.

23. Использование модулей `unittest` и `pytest` для написания тестов.

24. Логирование и анализ выполнения программ.

25. Принципы SOLID и их применение в `Python`.

26. Практика рефакторинга ООП-кода для улучшения его читаемости, масштабируемости и поддержки.

4 ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ

4.1 УЧЕБНАЯ ПРОГРАММА

Р-1 2024

Учреждение образования
«Брестский государственный технический университет»

УТВЕРЖДАЮ
Проректор по учебной работе
_____ М.В.Нерода
28.06 2024

Регистрационный № УД-24-1-258/уч.

Объектно-ориентированное программирование

Учебная программа учреждения высшего образования по учебной дисциплине для специальности:
6-05-0612-03 Системы управления информацией

2024

Учебная программа составлена на основе образовательного стандарта ОСВО 6-05-0612-03-2022, учебных планов по специальности высшего образования первой ступени специальности 6-05-0612-03 Системы управления информацией.

СОСТАВИТЕЛИ:
Михняев А.Л., старший преподаватель кафедры ИИТ
Соловчук А.М., старший преподаватель кафедры ИИТ
Яцук Т.А., преподаватель-стажер кафедры ИИТ

РЕЦЕНЗЕНТЫ:
Л.П. Махнист, доцент кафедры математики и информатики БрГТУ, кандидат технических наук
Д.В. Грицук, доцент заведующий кафедрой «Прикладной математики» БрГУ имени А.С. Пушкина, кандидат физико-математических наук.

РЕКОМЕНДОВАНА К УТВЕРЖДЕНИЮ:
Кафедрой интеллектуальных информационных технологий
Заведующий кафедрой _____ В.А. Головко
(протокол № 9 от 17.06.2024);

Методической комиссией факультета электронных информационных систем
Председатель методической комиссии _____ С.С. Дереченник
(протокол № 9 от 22.06.2024);

Научно-методическим советом БрГТУ (протокол № 5 от 28.06.2024)
Методист УМО _____ В.И. Серов

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Дисциплина «Объектно-ориентированное программирование» относится к государственному компоненту модуля «Программирование».

Подготовка современного специалиста требует уверенного владения возможностями, предоставляемыми компьютерными технологиями. Изучение настоящей дисциплины обеспечивает подготовку специалиста, владеющего фундаментальными знаниями и практическими навыками в области объектно-ориентированного анализа, программирования и элементов проектирования при решении практических задач.

Целью изучения курса является: углубленное обучение студентов технологическим основам и практическим навыкам проектирования, реализации и сопровождения больших программных систем современных ЭВМ на основе технологии объектно-ориентированного программирования.

Задачи изучения дисциплины:

- формирование представления об объектной технологии как примере использования системного подхода и реализации результатов системного анализа;
- приобретение знаний о возможностях, методах, моделях и средствах поддержки современных промышленных информационных технологий;
- приобретение навыков практической работы со средствами обеспечения жизненного цикла создания и эволюционного развития сложных программных систем.

В результате изучения учебной дисциплины «Объектно-ориентированное программирование» формируются следующие компетенции:

БПК-10. Использовать принципы объектно-ориентированного программирования для компьютерного моделирования реальных и концептуальных систем.

В результате изучения учебной дисциплины студент должен:

знать:

- принципы объектно-ориентированного программирования;
- способы реализации отношений между классами;
- использование свойств полиморфизма, наследования и инкапсуляции;
- возможности и ограничения абстрактных классов, интерфейсов и шаблонов;

уметь:

- создавать структурированные программы на основе объектных технологий в среде современных систем объектно-ориентированного проектирования;
- переходить из одной объектно-ориентированной платформы на другую;
- использовать возможности языка UML для представления проектных решений;

владеть:

- методами и инструментальными средствами и системами разработки объектно-ориентированных программ;

– техникой создания объектно-ориентированных программных компонент и организацией их взаимодействия в программных проектах.

Дисциплина базируется на знаниях, полученных студентами при изучении курсов «Информационные системы и технологии», «Архитектура ЭВМ», «Построение и анализ алгоритмов», «Основы алгоритмизации и программирования».

План учебной дисциплины для дневной формы получения высшего образования

Код специальности (направления специальности)	Наименование специальности (направления специальности)	Курс	Семестр	Всего учебных часов	Количество зачетных единиц	Аудиторных часов (в соответствии с учебным планом УВО)					Академических часов на курсовой проект (работу)	Форма текущей аттестации
						Всего	Лекции	Лабораторные занятия	Практические занятия	Семинары		
6-05-0612-03	Автоматизированные системы обработки информации	2	3	216	6	80	48	32			30	Экзамен

План учебной дисциплины для заочной формы получения высшего образования

Код специальности (направления специальности)	Наименование специальности (направления специальности)	Курс	Семестр	Всего учебных часов	Количество зачетных единиц	Аудиторных часов (в соответствии с учебным планом УВО)					Академических часов на курсовой проект (работу)	Форма текущей аттестации
						Всего	Лекции	Лабораторные занятия	Практические занятия	Семинары		
1-53 01 02	Автоматизированные системы обработки информации	2	3	260	6	18	10	8			30	Экзамен

1. СОДЕРЖАНИЕ УЧЕБНОГО МАТЕРИАЛА

1.1. ЛЕКЦИОННЫЕ ЗАНЯТИЯ, ИХ СОДЕРЖАНИЕ

Тема 1. ОБЗОР ЯЗЫКОВ ПРОГРАММИРОВАНИЯ C++, JAVA

Характеристика языков программирования C++, Java. Комментарии в языках C++, Java. Идентификаторы объектов. Область действия (видимости) объекта. Типы объектов. Атрибуты полей. Обращение по адресу и значению. Набор и приоритет операций в языках C++, Java. Операция преобразования(приведения) типа. Управление размещением объектов в памяти. Объявление и переопределение функций. Установка значений параметров функций по умолчанию. Функции с переменным числом параметров. Функции с подстановкой тела. Пространства имен.

Тема 2. КЛАССЫ КАК ТИПЫ ОБЪЕКТОВ

Понятие класса объектов. Виды классов. Синтаксис описания и определения класса. Схема взаимосвязи элементов класса и внешних объектов. Атрибуты доступа к элементам класса. Функции-элементы и элементы данных класса. Статические элементы класса. Дружественные функции класса. Дружественные классы. Конструкторы и деструкторы объектов. Особенности связи с конструкторами и деструкторами. Правила надежного определения класса. Примеры определения классов. Особенности образования структурированных объектов произвольных классов. Конструкторы вложенных классов. Массивы объектов классов. Статические объекты произвольных классов.

Тема 3. ОПРЕДЕЛЕНИЕ ОПЕРАЦИЙ НАД ОБЪЕКТАМИ КЛАССОВ

Схема определения операций над объектами. Особенности определения операций. Способы согласования типов объектов. Особенности использования ссылочных типов. Примеры определения операций над классами. Особенности операций присваивания и инициализации. Переопределение операции индексации. Переопределение операции вызова функции. Переопределение операций управления памятью.

Тема 4. ПРОИЗВОДНЫЕ КЛАССЫ

Понятие производного класса. Базовый класс и атрибуты его доступа. Иерархия производных классов. Множественное наследование. Виртуальные базовые классы. Конструкторы производных классов. Указатели объектов производного и базовых классов. Виртуальные функции. Виртуальные деструкторы. Абстрактные классы.

Тема 5. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ВВОД-ВЫВОД

Классы и потоки ввода-вывода. Стандартные потоки ввода-вывода. Схема иерархии классов ввода-вывода. Ввод-вывод объектов базовых типов. Примеры программ ввода-вывода с использованием стандартных потоков. Ввод-вывод объектов

определенных пользователем классов. Контроль исключительных ситуаций ввода-вывода. Форматный ввод-вывод. Бесформатный ввод-вывод. Функции-манипуляторы. Управление позиционированием потока. Связанные потоки. Создание и организация взаимодействия потоков. Буферы потоков. Файловый ввод-вывод. Строчный ввод-вывод.

Тема 6. ОСОБЕННОСТИ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Обзор операционных особенностей объектов класса. Особенности динамического управления памятью. Использование статических элементов класса. Статическое и динамическое связывание. Динамические особенности операторов присваивания. Правила программирования на языке C++.

Тема 7. ШАБЛОНЫ ФУНКЦИЙ И КЛАССОВ

Назначение и виды шаблонов в языке C++. Шаблоны функций. Определение и использование шаблона функции. Переопределение шаблонов функций и их специализация. Шаблоны классов. Определение и использование шаблона класса. Переопределение шаблонов классов и их специализация. Параметры шаблонов. Особенности использования шаблонов в многофайловых проектах. Стандартные библиотеки шаблонов STL и ATL. Характеристика шаблонов определений функций и классов.

Тема 8. ОБРАБОТКА ИСКЛЮЧЕНИЙ

Понятие исключения. Виды и спецификация исключений. Порождение и перехват исключений. Контролируемые блоки и ловушки исключений. Структурное управление исключениями. Кадрированное и завершающее управление исключениями. Примеры предопределенных классов исключений. Иерархическое управление исключениями.

Тема 9. ДИНАМИЧЕСКАЯ ИДЕНТИФИКАЦИЯ И ПРИВЕДЕНИЕ ТИПА

Динамическая идентификация типа. Недостатки традиционных операций приведения типа. Обзор новых возможностей приведения типа Динамическое приведение типа. Полиморфизм и приведение типа. Нисходящее приведение типа. Перекрестное приведение типа. Статическое приведение типа. Преобразования типа с сохранением значений.

Тема 10. СИСТЕМЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Обзор систем и языков объектно-ориентированного программирования. Критический анализ средств поддержки ключевых понятий объектно-ориентированного программирования. Область применимости технологии объектно-ориентированного программирования. Системы быстрой разработки приложений. Системы визуального программирования. Библиотечные классы абстрактных структур данных. Иерархия классов абстрактных структура данных. Обзор предопределённых

возможностей классов абстрактных структур данных. Классы паттернов ассоциаций объектов. Компонентные классы. Обзор библиотеки QT.

1.2. ПЕРЕЧЕНЬ ТЕМ ЛАБОРАТОРНЫХ ЗАНЯТИЙ, ИХ НАЗВАНИЕ

ЛАБОРАТОРНАЯ РАБОТА №1 «Классы».
ЛАБОРАТОРНАЯ РАБОТА №2 «Отношения между классами и определение операций над объектами классов».
ЛАБОРАТОРНАЯ РАБОТА №3 «Производные классы».
ЛАБОРАТОРНАЯ РАБОТА №4 «Файлы. Потоки ввода/вывода».
ЛАБОРАТОРНАЯ РАБОТА №5 «Шаблоны функций и классов».
ЛАБОРАТОРНАЯ РАБОТА №6 «Исключительные ситуации».
ЛАБОРАТОРНАЯ РАБОТА №7 «Системы ООП».
ЛАБОРАТОРНАЯ РАБОТА №8 «UML-диаграммы».

2. ТРЕБОВАНИЯ К КУРСОВОМУ ПРОЕКТУ

Цель выполнения курсовой работы – получить практические навыки разработки иерархий классов, приложений на базе объектной парадигмы с использованием моделей UML Задания на курсовое проектирование предполагают выполнение следующих работ:

1. Анализ и описание предметной области, включая описание вариантов использования системы.
2. Проектирование, включая разработку иерархий классов (в том числе с учетом используемых парадигм, библиотек, шаблонов и т.д.), диаграмм взаимодействия объектов, диаграмм видов деятельности и т.д. Разработка структуры системы, диаграмм компонентов, развертывания.
3. Реализация прототипа программной системы. Разработка программной документации.

Примерный перечень тем курсовых проектов (работ)

1. Разработка библиотек классов для решения вычислительных задач.
2. Разработка библиотек классов для решения задач обработки текстов.
3. Разработка библиотек классов для решения задач моделирования.
4. Разработка библиотек классов для решения задач обработки структур данных.
5. Разработка средств, компонентов поддержки интеллектуальных систем
6. Разработка информационно-справочных, обучающих систем, бизнес-приложений, моделей систем и т.д.

3.1. УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ
для дневной формы получения высшего образования

Номер раздела, темы	Название раздела, темы	Количество аудиторных часов				Количество часов самост. работы	Форма контроля знаний
		Лекции	Лабораторные	Практические	Семинарские занятия		
1.1	Обзор языков программирования C++, Java	6				6	устный опрос
1.2	Классы как типы объектов	6				8	устный опрос
	Лабораторная работа №1. «Классы»		4			8	защита отчета
1.3	Определение операций над объектами классов.	2				8	устный опрос
	Лабораторная работа №2 «Отношения между классами и определение операций над объектами классов»		4			8	защита отчета
1.4	Производные классов	4				6	устный опрос
	Лабораторная работа №3 «Производные классы»		4			8	защита отчета
1.5	Объектно-ориентированный ввод-вывод,	2				8	устный опрос
1.6	Особенности объектно-ориентированного программирования	2				8	устный опрос
	Лабораторная работа №4 «Файлы. Потоки ввода/вывода»		4			8	защита отчета
1.7	Шаблоны функций и классов	6				6	устный опрос
	Лабораторная работа №5 «Шаблоны функций и классов»		4			8	защита отчета
1.8	Обработка исключений	6				6	устный опрос
	Лабораторная работа №6 «Исключительные ситуации».		4			8	защита отчета
1.9	Динамическая идентификация и приведение типа	6				6	устный опрос
1.10	Системы объектно-ориентированного программирования.	8				8	устный опрос
	Лабораторная работа №7 «Системы ООП»		4			9	защита отчета
	Лабораторная работа №8 «UML-диаграммы»		4			9	защита отчета
	Итого за семестр	48	32			136	экзамен

3.2. УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ
для заочной формы получения высшего образования

Номер раздела, темы	Название раздела, темы	Количество аудиторных часов				Количество часов самост. работы	Форма контроля знаний
		Лекции	Лабораторные занятия	Практические занятия	Семинарские занятия		
1.1	Обзор языков программирования C++, Java	1				10	устный опрос
1.2	Классы как типы объектов	1				10	устный опрос
	Лабораторная работа №1. «Классы»		1			12	защита отчета
1.3	Определение операций над объектами классов.	1				10	устный опрос
	Лабораторная работа №2 «Отношения между классами и определение операций над объектами классов»		1			12	защита отчета
1.4	Производные классов	1				12	устный опрос
	Лабораторная работа №3 «Производные классы»		1			12	защита отчета
1.5	Объектно-ориентированный ввод-вывод,	1				10	устный опрос
1.6	Особенности объектно-ориентированного программирования	1				10	устный опрос
	Лабораторная работа №4 «Файлы. Потоки ввода/вывода»		1			12	защита отчета
1.7	Шаблоны функций и классов	1				10	устный опрос
	Лабораторная работа №5 «Шаблоны функций и классов»		1			12	защита отчета
1.8	Обработка исключений	1				10	устный опрос
	Лабораторная работа №6 «Исключительные ситуации».		1			12	защита отчета
1.9	Динамическая идентификация и приведение типа	1				10	устный опрос
1.10	Системы объектно-ориентированного программирования.	1				10	устный опрос
	Лабораторная работа №7 «Системы ООП»		1			12	защита отчета
	Лабораторная работа №8 «UML-диаграммы»		1			12	защита отчета
	Итого за семестр	10	8			198	экзамен

4. ИНФОРМАЦИОННО-МЕТОДИЧЕСКАЯ ЧАСТЬ

4.1. Перечень литературы

Основная:

1. Буч Г. Объектно-ориентированное проектирование с примерами применения: 3_е издание. – Пер с англ. – СПб.: "Невский Диалект", 2015. – 720 с.

2. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно–ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2018. – 368 с.

3. Ларман К. Применение UML 2.0 и шаблонов проектирования: Пер с англ. – М.: Вильямс, 2017. – 736 с.

4. Страуструп Б. Язык программирования C++. Специальное издание: Пер с англ.– М.: Бином, 2017. – 1136 с.

5. Фаулер М. UML. Основы, 3_е издание. – Пер. с англ. – СПб: Символ_Плюс, 2018. – 192 с.

6. Фаулер, М. Архитектура корпоративных программных приложений.: Пер. с англ. – М.: Вильяме, 2006. – 544 с.

7. Шилдт Г. C++. Базовый курс: Пер с англ. – М.: Вильямс, 2018. – 624 с.

Дополнительная

1. Голуб А.И. С и C++. Правила программирования: Пер с англ. – М.: Бином, 1997. – 272 с.

2. Лафоре Р. Объектно-ориентированное программирование в C++: Пер с англ. – Спб.: Питер, 2007. – 928 с

3. Калверт Ч. Borland C++ Builder. Энциклопедия пользователя: Пер с англ. – К.: ДИАСОФТ, 1998. – 830 с.

4. Мюллер Дж. Visual C++ 5/Пер с англ. – Спб.: BHV – Санкт-Петербург, 1998. – 720 с.

5. Корнелл Г., Хорстманн К. Java 2. Том I. Основы. Библиотека профессионала: Пер с англ. – М.: Вильямс, 2008. – 816 с.

6. Саттер Г. Решение сложных задач на C++: Пер с англ. – М.: Вильямс, 2015. – 400 с.

7. Седжвик Р. Фундаментальные алгоритмы на C++. Части 1-4. Анализ. Структуры данных. Сортировка. Поиск: Пер с англ. – К.: ДиаСофт, 2002. – 687 с.

8. К. Ш. Тан, В.-Х. Стиб, Й. Харди, Символьный C++. Введение в компьютерную алгебру с использованием объектно-ориентированного программирования. – М.: Мир, 2001. – 624 с.

4.2. Для диагностики результатов учебной деятельности используются:

- контрольные опросы;
- письменные отчеты по лабораторным работам;
- письменный экзамен;
- текущая аттестация.

4.3. Методические рекомендации по организации и выполнению самостоятельной работы обучающихся по учебной дисциплине

Самостоятельная работа предполагает:

- ведение и систематическую проработку конспекта лекций и учебной литературы;
- подготовку к практическим занятиям – включает проработку соответствующих разделов рекомендованной литературы и конспекта лекций по тематике лабораторных работ;
- подготовку письменных отчетов по результатам выполнения индивидуальных заданий лабораторных работ;
- подготовку к письменному экзамену по дисциплине.

Ссылки на рекомендуемые источники, учебную литературу (в разрезе тем изучаемой дисциплины) представлены в таблице ниже.

Тема учебной дисциплины	Литература
1	2
Тема 1. Обзор языков программирования C++, Java	[4], [7]
Тема 2. Классы как типы объектов	[1], [4]
Тема 3. Определение операций над объектами классов	[2], [4], [7]
Тема 4. Производные классы	[2], [4]
Тема 5. Объектно-ориентированный ввод-вывод	[2], [7]
Тема 6. Особенности объектно-ориентированного программирования	[4, [6]
Тема 7. Шаблоны функций и классов	[3], [5], [7]
Тема 8. Обработка исключений	[4], [7]
Тема 9. Динамическая идентификация и приведение типа	[4]
Тема 10. Системы объектно-ориентированного программирования	[6], [7]