Министерство образования и науки Российской Федерации

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

---

Д. В. ЛИСИЦИН

# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Утверждено Редакционно-издательским советом университета
в качестве конспекта лекций

2-е издание, переработанное и дополненное

НОВОСИБИРСК
2010

УДК 004.434(075.8)
Л 632

Рецензенты
д-р техн. наук, проф. *В.И. Хабаров*;
канд. техн. наук, доц. *И.Л. Еланцева*

Работа подготовлена на кафедре прикладной математики
для студентов II курса ФМПИ

**Лисицин Д.В.**

Л 632 Объектно-ориентированное программирование : конспект лекций / Д.В. Лисицин. – 2-е изд., перераб. и доп. – Новосибирск : Изд-во НГТУ, 2010. – 88 с.

ISBN 978-5-7782-1454-5

Конспект лекций является введением в объектно-ориентированные методы разработки программного обеспечения. Рассмотрены основы объектно-ориентированного подхода: составляющие объектной модели, характеристики объектов, классов и типы отношений между ними. Особое внимание уделяется выражению рассматриваемых элементов объектно-ориентированного подхода на языке программирования C++ и унифицированном языке моделирования UML.

**УДК 004.434(075.8)**

**ISBN 978-5-7782-1454-5**

# ВВЕДЕНИЕ

За последние 15 – 20 лет объектно-ориентированная технология стала *основным* методом промышленной разработки программного обеспечения.

Начало ее развитию положил язык программирования Simula 67, который был разработан в конце 1960-х гг. в Норвегии. Несмотря на то что язык намного опередил свое время, современники (программисты 60-х гг.) оказались не готовы воспринять ценности языка Simula 67, и он не выдержал конкуренции с другими языками программирования (прежде всего, с языком Fortran).

Но достоинства языка Simula 67 были замечены некоторыми программистами, и в 1970-е гг. было разработано большое число экспериментальных объектно-ориентированных языков программирования. В результате исследования этих языков были разработаны современные объектно-ориентированные языки программирования: C++, Ada, Smalltalk и др.

Одним из наиболее распространенных объектно-ориентированных языков программирования является язык C++. Он возник на базе соединения языков C и Simula. C++ был разработан в начале 1980-х Бьерном Страуструпом, сотрудником компании AT&T. В течение почти двух десятилетий язык C++ интенсивно развивался, и в 1998 г. был принят его международный стандарт.

Разработка новых объектно-ориентированных языков программирования продолжается и в настоящее время. Например, за последние 15 лет были разработаны и получили широкое распространение объектно-ориентированные языки программирования Java и C#, отражающие ряд современных тенденций в области объектно-ориентированной технологии.

Вместе с развитием объектно-ориентированного программирования развивались и объектно-ориентированные методы разработки про-

граммного обеспечения, охватывающие стадии анализа и проектирования. Среди известных объектно-ориентированных подходов к анализу и проектированию следует выделить методы Г. Буча, Д. Рамбо, А. Джекобсона, Шлеера–Меллора и Коуда–Йордона. В результате объединения усилий первых трех авторов появился на свет унифицированный язык моделирования UML, который в 1997 г. был принят в качестве стандарта консорциумом Object Management Group и получил широкое распространение в сфере производства программного обеспечения.

Основные идеи объектно-ориентированного подхода опираются на следующие положения:

- программа представляет собой *модель* некоторого реального процесса, части реального мира; модель содержит не все признаки и свойства представляемой ею части реального мира, а только те, которые существенны для разрабатываемой программной системы;
- модель реального мира или его части может быть описана как совокупность взаимодействующих между собой *объектов*;
- объект описывается набором *атрибутов* (свойств), значения которых определяют состояние объекта, и набором *операций* (действий), которые может выполнять объект;
- взаимодействие объектов осуществляется посылкой специальных *сообщений* от одного объекта к другому; сообщение, полученное объектом, может потребовать выполнения определенных действий, например, изменения состояния объекта;
- объекты, описанные одним и тем же набором атрибутов и способные выполнять один и тот же набор операций, представляют собой *класс* однотипных объектов.

Процесс представления предметной области задачи в виде совокупности объектов, обменивающихся сообщениями, называется **объектной декомпозицией**. Использование объектной декомпозиции – главный признак, отличающий объектно-ориентированный подход от структурного, для которого характерна функциональная (алгоритмически-ориентированная) декомпозиция задачи.

С точки зрения языка программирования класс объектов можно рассматривать как тип данных, а отдельные объекты – как данные этого типа. Определение программистом собственных классов объектов должно позволить описывать конкретную задачу в терминах ее предметной области (при соответствующем выборе имен типов и имен объектов, их атрибутов и выполняемых действий).

Объектно-ориентированный подход дает следующие основные преимущества:

- уменьшает *сложность* программного обеспечения;
- повышает его *надежность*;
- обеспечивает возможность *модификации* отдельных компонентов программ без изменения остальных компонентов;
- обеспечивает возможность *повторно использовать* отдельные компоненты программного обеспечения.

Систематическое применение объектно-ориентированного подхода позволяет разрабатывать хорошо структурированные, надежные в эксплуатации, достаточно просто модифицируемые программные системы. Этим объясняется интерес программистов к объектно-ориентированному подходу и объектно-ориентированным языкам программирования.

Целью данного конспекта лекций является введение в объектно-ориентированный подход к разработке программного обеспечения. В рамках курса рассмотрены концепции и понятия объектно-ориентированного подхода (на основе [2, 5]), включая их выражение на языке программирования C++ и унифицированном языке моделирования UML.

# 1. СЛОЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Объектно-ориентированный подход возник в первую очередь в ответ на растущую **сложность** программного обеспечения. На заре компьютерной эры возможности компьютеров были ограничены и было очень трудно написать большую программу. В 1960–70-е гг. эффективность применения компьютеров резко возросла и стало все больше создаваться прикладных программ повышенной сложности. Наибольшее распространение в это время получило структурное проектирование по методу сверху вниз. Однако через некоторое время оказалось, что структурный подход не работает, если объем программы превышает приблизительно 100 тыс. строк. Как результат – выход проектов за рамки установленных сроков и бюджетов и, более того, их несоответствие начальным требованиям. Для решения этих проблем и стали применять объектно-ориентированный подход.

Справедливости ради отметим, что объектно-ориентированный подход – это достаточно универсальный инструмент. Он может применяться и для разработки программ малой и средней сложности. Однако именно для сложных систем использование объектно-ориентированного подхода является критичным. Таким образом, основная область использования объектно-ориентированного подхода – это **сложные промышленные программные продукты.**

Подобные системы могут применяться для решения самых разных задач. В качестве примеров можно привести:

- системы с обратной связью (интеллектуальные, самообучающиеся системы), которые активно взаимодействуют с физическим миром или управляются его событиями и для которых ресурсы времени и памяти ограничены;
- задачи поддержания целостности информации объемом в сотни тысяч записей при параллельном доступе к ней с обновлениями и запросами;
- системы управления реальными процессами и контроля над ними (например, диспетчеризация воздушного и железнодорожного транспорта).

Системы подобного типа обычно имеют большое время жизни, и большое количество пользователей оказывается в зависимости от их нормального функционирования. Глобальные системы национального

или даже мирового масштаба – это яркие примеры таких систем (например, сеть Интернет).

Сложность является существенной чертой промышленной программы: один разработчик практически не в состоянии охватить все аспекты такой системы. Фактически сложность промышленных программ превышает возможности интеллекта одного человека.

Сложность, присущая программному обеспечению, определяется следующими **основными причинами**: сложностью реального мира, сложностью управления процессом разработки, гибкостью программного обеспечения, сложностью описания поведения систем.

**Сложность реального мира**

Проблемы, которые мы пытаемся решить с помощью разрабатываемого программного обеспечения, часто содержат сложные элементы, к которым предъявляется множество различных и нередко противоположных требований.

Но даже в простых проблемах сложность может возникнуть из-за языковой и понятийной «нестыковки» между заказчиком системы и ее разработчиком: пользователи обычно с трудом могут внятно объяснить разработчикам, *что* на самом деле нужно сделать. Часто пользователь лишь смутно представляет, *что* ему нужно от будущей программной системы. С другой стороны, разработчик, являясь экспертом лишь в своей области знаний, недостаточно квалифицирован в предметной области.

Дополнительные сложности возникают в результате изменения требований к программной системе уже в процессе разработки. В основном требования корректируются из-за того, что само осуществление программного проекта часто изменяет проблему. Использование системы, после того как она разработана и установлена, заставляет пользователей лучше понять и отчетливей сформулировать то, что им действительно нужно. В то же время этот процесс повышает квалификацию разработчиков в предметной области и позволяет им задавать более осмысленные вопросы, которые проясняют темные места в проектируемой системе.

**Сложность управления процессом разработки**

Большое число требований к системе неизбежно приводит либо к созданию нового программного продукта значительных размеров, либо к модификации существующего, что также не делает его проще.

Сегодня обычными стали системы размером в десятки тысяч и даже миллионы строк на языках высокого уровня. Ни один человек не в состоянии понять и разработать полностью такую систему. Поэтому возникает необходимость разбиения системы на подсистемы и модули и коллективной разработки систем.

В идеале для успеха разработки команда разработчиков должна быть как можно меньше. Но какой бы она ни была, всегда возникнут трудности, связанные с координацией работ над проектом, и проблема взаимопонимания требований и спецификаций системы.

**Гибкость программного обеспечения**

Программирование обладает максимальной гибкостью среди технических наук. Программист, как и писатель, работает со словом, и всеми базовыми элементами, необходимыми для создания программ, он может обеспечить себя сам, зачастую пренебрегая уже существующими разработками. Такая гибкость – чрезвычайно привлекательное, но опасное качество: пользователь, осознав эту возможность, постоянно изменяет свои требования; разработчик увлекается украшательством своей системы во вред основному ее назначению. Поэтому программные разработки остаются очень кропотливым и «бесконечным» делом, а программные системы потенциально не завершенными.

**Сложность описания поведения системы**

Сложные программные системы содержат сотни и тысячи переменных, текущие значения которых в каждый момент времени описывают состояние программы. Кроме того, они имеют большое количество точек ветвления, которые определяют множество зависящих от ситуации путей решения задачи. Все это разработчик должен продумать, зафиксировать в программах, протестировать и отладить.

Любая сложная система, в том числе и сложная программная система, обладает следующими **общими признаками.**

1. Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы и т. д.

Сложная система состоит не просто из отдельных компонентов, между ними имеются определенные иерархические отношения.

Например, большинство персональных компьютеров состоит из одних и тех же основных элементов: системного блока, монитора, клавиатуры и манипулятора «мышь». Мы можем взять любую из этих

частей и разложить ее, в свою очередь, на составляющие. Системный блок, например, содержит материнскую плату, платы оперативной памяти, центральный процессор, жесткий диск и т. д.

Продолжая, мы можем разложить на составляющие центральный процессор. Он состоит из регистров и схем управления, которые сами состоят из еще более простых деталей: диодов, транзисторов и т. д. Возникает вопрос: что же считать простейшим элементом системы? Ответ дает второй признак.

2. Выбор того, какие компоненты в данной системе считаются элементарными, относительно произволен и в большой степени оставляется на усмотрение исследователя.

Низший уровень для одного наблюдателя может оказаться достаточно высоким для другого. Если пользователю достаточно выделить системный блок, монитор и клавиатуру, то для разработчика компьютера этого явно недостаточно.

3. Внутрикомпонентная связь обычно сильнее, чем связь между компонентами.

Это обстоятельство позволяет отделять интенсивные (высокочастотные) взаимодействия внутри компонентов от менее интенсивных (низкочастотных) взаимодействий между компонентами и дает возможность относительно изолированно изучать каждый компонент.

4. Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных.

Иными словами, разные сложные системы содержат одинаковые структурные части. Эти части в свою очередь могут использовать общие более мелкие компоненты. Например, и у растений, и у животных имеются крупные подсистемы типа сосудистых систем и клетки как более мелкие компоненты.

5. Любая работающая сложная система является результатом развития более простой системы.

В качестве примера назовем теорию эволюции живой природы.

Сложная система, спроектированная с нуля, вряд ли заработает. Следует начать с работающей простой системы.

В процессе развития системы объекты, первоначально рассматривавшиеся как сложные, становятся элементарными, и из них (как устойчивых промежуточных форм) строятся более сложные системы.

# 2. ОБЪЕКТНАЯ МОДЕЛЬ

Объектно-ориентированный подход основывается на совокупности ряда принципов, называемой **объектной моделью**.

Главными принципами являются: абстрагирование, инкапсуляция, модульность, иерархичность. Главные они в том смысле, что без них модель не будет объектно-ориентированной.

Кроме главных, назовем еще три дополнительных принципа: типизация, параллелизм, сохраняемость. Называя их дополнительными, мы имеем в виду, что они полезны в объектной модели, но не обязательны.

## 2.1. АБСТРАГИРОВАНИЕ

Люди развили чрезвычайно эффективную технологию преодоления сложности. Мы абстрагируемся от нее. Если мы не в состоянии полностью воссоздать сложный объект, то приходится игнорировать не слишком важные детали. В результате мы имеем дело с обобщенной, идеализированной моделью объекта.

Например, изучая процесс фотосинтеза у растений, мы концентрируем внимание на химических реакциях в определенных клетках листа и не обращаем внимания на остальные части – черенки, жилки и т. д.

**Абстракция** – совокупность существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа.

**Абстрагирование** – процесс выделения абстракций в предметной области задачи.

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных. Такое разделение смысла и реализации называют **барьером абстракции**. Установление того или иного барьера абстракции порождает множество различных абстракций для одного и того же предмета или явления реального мира. Абстрагируясь в большей или меньшей степени от различных аспектов проявления реальности, мы находимся на разных **уровнях абстракции**.

Для примера рассмотрим системный блок компьютера. Пользователю, использующему компьютер для набора текста, не важно, из ка-

ких частей состоит этот блок. Для него это – коробка с кнопками и возможностью подсоединения внешних запоминающих устройств. Он абстрагируется от таких понятий, как «процессор» или «оперативная память». С другой стороны, у программиста, пишущего программы на языках низкого уровня, барьер абстракции лежит намного ниже. Ему необходимо знать устройство процессора и команды, понимаемые им.

Является полезным еще один дополнительный принцип, называемый **принципом наименьшего удивления.** Согласно ему абстракция должна охватывать все поведение объекта, но не больше и не меньше, и не привносить сюрпризов или побочных эффектов, лежащих вне ее сферы применимости.

Все абстракции обладают как статическими, так и динамическими свойствами. Например, файл как объект требует определенного объема памяти на конкретном устройстве, имеет имя и содержимое. Эти атрибуты являются статическими свойствами. Конкретные же значения каждого из перечисленных свойств динамичны и изменяются в процессе использования объекта: файл можно увеличить или уменьшить, изменить его имя и содержимое.

Будем называть **клиентом** любой объект, использующий ресурсы другого объекта, называемого **сервером.** Мы будем характеризовать поведение объекта услугами, которые он оказывает другим объектам, и операциями, которые он выполняет над другими объектами. Этот подход концентрирует внимание на внешних проявлениях объекта и реализует так называемую **контрактную модель программирования.** Эта модель заключается в следующем: внешнее проявление объекта рассматривается с точки зрения его контракта с другими объектами, в соответствии с этим должно быть выполнено и его внутреннее устройство (часто – во взаимодействии с другими объектами). Контракт фиксирует все обязательства, которые объект-сервер имеет перед объектом-клиентом. Другими словами, этот контракт определяет **ответственность** объекта – то поведение, за которое он отвечает.

Каждая операция, предусмотренная контрактом, однозначно определяется ее **сигнатурой** – списком типов формальных параметров и типом возвращаемого значения (в языке C++ тип возвращаемого значения не является частью сигнатуры). Полный набор операций, которые клиент может осуществлять над другим объектом, вместе с правильным порядком, в котором эти операции вызываются, называется **протоколом.** Протокол отражает все возможные способы, кото-

рыми объект может действовать или подвергаться воздействию. Тем самым протокол полностью определяет внешнее поведение абстракции.

**Пример.** В тепличном хозяйстве, использующем гидропонику, растения выращиваются на питательном растворе без песка, гравия и другой почвы. Управление режимом работы парниковой установки – очень ответственное дело. Оно зависит как от вида выращиваемых культур, так и от стадии выращивания. Нужно контролировать целый ряд факторов: температуру, влажность, освещение, кислотность и концентрацию питательных веществ. В больших хозяйствах для решения этой задачи часто используют автоматические системы, которые контролируют и регулируют указанные факторы. Цель автоматизации состоит здесь в том, чтобы при минимальном вмешательстве человека добиться соблюдения режима выращивания.

Одна из ключевых абстракций в данной задаче – **датчик.** Известно несколько разновидностей датчиков. Все, что влияет на урожай, должно быть измерено. Таким образом, нужны датчики температуры воды, температуры воздуха, влажности, кислотности, освещения и концентрации питательных веществ.

С внешней точки зрения **датчик температуры** – это объект, который способен измерять температуру там, где он расположен. Температура – это числовой параметр, имеющий ограниченный диапазон значений и определенную точность и означающий число градусов по Цельсию.

Местоположение датчика – это некоторое однозначно определенное место в теплице, температуру в котором необходимо знать. Таких мест, вероятно, немного. Для датчика температуры при этом существенно не само местоположение, а только то, что данный датчик расположен именно в данном месте.

Рассмотрим элементы реализации нашей абстракции на языке C++.

```
typedef  float  Temperature; // Температура по Цельсию
typedef  unsigned  int  Location; // Число, однозначно определяющее
                                  // положение датчика
```

Здесь два оператора определения типов Temperature и Location вводят удобные псевдонимы для простейших типов, и это позволяет нам выражать свои абстракции на языке предметной области.

Temperature – это числовой тип данных в формате с плавающей точкой для записи температур. Значения типа Location нумеруют места, где могут располагаться температурные датчики.

Рассмотрим обязанности датчика температуры. Датчик должен знать значение температуры в своем местонахождении и сообщать ее по запросу. Клиент по отношению к датчику может выполнить такие действия: калибровать датчик и получать от него значение текущей температуры. Таким образом, объект «Датчик температуры» имеет две операции: «Калибровать» и «Текущая температура».

```
struct  TemperatureSensor { // Датчик температуры
        Temperature  curTemperature; // текущая температура в
                                     // местонахождении датчика
        Location  loc; // местонахождение датчика
        void calibrate(Temperature actualTemperature); // калибровать
        Temperature  currentTemperature( ); // текущая температура
};
```

Данным описанием вводится новый тип TemperatureSensor. Важным здесь является то, что, во-первых, данные и функции, изменяющие их, объединены вместе в одном описании, и, во-вторых, мы не работаем непосредственно с данными, а только посредством соответствующих функций. В частности, здесь мы использовали так называемые **set-** и **get-функции**, соответственно устанавливающие и возвращающие значения переменных (calibrate – set-функция, currentTemperature – get-функция).

Объекты данного типа вводятся так же, как и переменные стандартных типов:

```
TemperatureSensor  TSensors[100]; // массив из ста объектов типа
                                  // TemperatureSensor
```

Функции, объявленные внутри описания, называются **функциями-членами.** Их можно вызывать только для переменной соответствующего типа. Например, калибровать датчик можно так:

```
TSensors[3].calibrate(20.); // калибруется датчик номер 3
```

Поскольку имя объекта, для которого вызывается функция-член, неявно ей передается, в списках аргументов функций отсутствует аргумент типа TemperatureSensor, задающий конкретный датчик, над которым производятся действия. К этому объекту внутри функции можно явно обратиться по указателю this. Например, в теле функции calibrate можно написать один из двух эквивалентных операторов

```
curTemperature = actualTemperature;
this -> curTemperature = actualTemperature;
```

Центральной идеей абстракции является понятие инварианта. **Инвариант** – это некоторое логическое условие, значение которого (истина или ложь) должно сохраняться. Для каждой операции объекта можно задать предусловия (т. е. инварианты, предполагаемые операцией) и постусловия (т. е. инварианты, которым удовлетворяет операция).

Рассмотрим инварианты, связанные с операцией currentTemperature. Предусловие включает предположение, что датчик установлен в правильном месте в теплице, а постусловие – что датчик возвращает значение температуры в градусах Цельсия.

Изменение инварианта нарушает контракт, связанный с абстракцией. Если нарушено предусловие, то клиент не соблюдает свои обязательства и сервер не может выполнить задачу правильно. Если нарушено постусловие, то свои обязательства нарушил сервер и клиент не может ему больше доверять.

Для проверки условий язык C++ предоставляет ряд специальных средств.

В случае нарушения какого-либо условия следует сгенерировать **исключительную ситуацию (исключение).** Объекты могут генерировать исключения, чтобы запретить дальнейшее выполнение операции и предупредить о проблеме другие объекты, которые в свою очередь могут принять на себя перехват исключения и справиться с проблемой. Причиной такого разделения является то, что объект-сервер, обнаруживший ошибку, может не знать, что предпринимать для ее исправления, а объект-клиент может знать, что делать, но не уметь определить место возникновения.

C++ имеет специальный механизм обработки исключений, чувствительный к контексту. Контекстом для генерации исключения явля-

ется блок try (пробный блок). Если при выполнении операторов, находящихся внутри блока try, происходит исключительная ситуация, то управление передается обработчикам исключений, которые задаются ключевым словом catch и находятся ниже блока try. Синтаксически обработчик catch выглядит подобно функции с одним аргументом без указания типа возвращаемого значения. Для одного блока try может быть задано несколько обработчиков, отличающихся типом аргумента.

```
try{ // пробный блок
. . .
}
catch(char * error){. . .} // имя аргумента используется в обработчике
catch(int){. . .} // имя аргумента не используется в обработчике
catch(…){. . .} // обрабатываются все исключения
```

Исключение генерируется посредством указания ключевого слова throw с не обязательным аргументом-выражением.

```
throw 1;
```

Исключение будет обработано посредством вызова того обработчика catch, тип параметра которого будет соответствовать типу аргумента throw. При поиске подходящего обработчика все обработчики просматриваются в порядке их записи.

При наличии вложенных блоков try (например, из-за вложенности вызовов функций) будет использован обработчик самого глубокого блока. Если обработчика, соответствующего типу аргумента throw, на данном уровне не будет найдено, будет осуществлен выход из текущей функции (с уничтожением всех локальных объектов) и поиск в блоке try с меньшей глубиной вложенности и т. д. После обработки исключения управление передается на оператор, следующий за описаниями обработчиков catch.

**Пример.** Рассмотрим стек, реализованный с использованием массива фиксированной длины.

```
int stack [100]; // не более ста элементов в стеке
int top=0; // номер доступного места для помещения элемента
void push(int  el) {
```

```
    if(top = = 100) throw 1; // проверить на переполнение
                                    // (предусловие top < 100)
    else stack[top ++] = el; // поместить элемент в стек
}
int pop( ) {
    if(top = = 0) throw 0; // проверить на пустоту
                                  // (предусловие top > 0)
    else return stack[--top]; // извлечь элемент из стека
}
void main( ) {
    int i = 0, k;
    try{ // пробный блок
          push(i);
          k = pop( );
          if(i!=k) throw 2; // нарушено постусловие
    }
    catch(int error){. . .} // если error = 0, то стек пуст;
// если error = 1, то стек полон; если error = 2, то стек неработоспособен
}
```

В примере аргументом throw является целое число – «номер исключения». В сложных программах разрабатываются специальные типы для исключений, позволяющие передать в обработчик исключения больше информации.

## 2.2. ИНКАПСУЛЯЦИЯ

**Инкапсуляция** – это процесс разделения элементов абстракции, определяющих ее структуру и поведение; инкапсуляция предназначена для изоляции контрактных обязательств абстракции от их реализации.

На самом деле клиента не интересует и не должно интересовать то, как реализовано выполнение контрактных обязательств. По крайней мере, пока сервер соблюдает свои обязательства.

**Пример.** Продолжим пример со стеком. Стек позволяет осуществлять операции pop (извлечь из стека) и push (поместить в стек). Для программиста, *использующего стек*, важно только то, что он может помещать и извлекать нужные ему объекты с помощью вызова данных

операций. Как реализован стек, он может не знать, и детали реализации для него не всегда важны. Стек может быть реализован с использованием массива, имеющего фиксированное количество элементов, или посредством списковой структуры. Однако все эти детали скрыты от пользователя.

Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция сосредоточена на внутреннем устройстве, обеспечивающем заданное поведение. Инкапсуляция выполняется посредством скрытия информации, т. е. маскировкой всех внутренних деталей, не влияющих на внешнее поведение. Обычно скрываются и внутренняя структура объекта, и реализация его операций. Для скрытия информации многие объектно-ориентированные языки программирования имеют соответствующие механизмы.

В результате всего сказанного мы можем ввести понятия интерфейса и реализации. **Интерфейс** – это набор операций, используемый для специфицирования услуг, предоставляемых объектом. Интерфейс отражает внешнее поведение объекта. Внутренняя **реализация** описывает представление этой абстракции и механизмы достижения желаемого поведения объекта.

Интерфейс стека – это его операции pop и push, а реализация – это конкретное представление стека.

**Пример.** Перепишем реализацию стека, рассмотренную в предыдущем пункте, с использованием структуры.

```
struct Stack {
        int s[100];
        int top;
        void push(int el);
        int pop( );
};
```

Функции pop и push изменяют значения переменных-членов структуры. Однако изменить их значения могут и другие функции. При этом такие изменения могут быть внесены и по ошибке. Следовательно, имеет смысл ограничить доступ к данным объектов типа Stack.

Объявление Stack предоставляет набор функций для работы с объектами типа Stack. Однако оно не указывает, что только эти функции

могут непосредственно осуществлять доступ к элементам объекта типа Stack. Эти ограничения можно отразить следующим образом:

```
class Stack {
private:
        int s[100];
        int top;
public:
        void push(const int el);
        int pop( );
        bool isFull( ) const;
        bool isEmpty( ) const;
};
```

Описание класса Stack разделено на **закрытую** и **открытую** части, помеченные как private и public. Открытая часть (public) образует открытый интерфейс объектов класса. Имена закрытой части (private) могут использоваться только функциями-членами, а также друзьями класса.

**Друзьями класса** называются классы или операции, имеющие доступ к закрытым операциям или данным некоторого класса. При описании класса его друзья указываются с ключевым словом friend.

Мы описали Stack как класс, а не как структуру. Принципиального отличия здесь нет, поскольку структура в C++ является классом, члены которого, однако, по умолчанию открыты. Члены класса, описанного ключевым словом class, по умолчанию являются закрытыми.

В описание стека добавлены две функции, определяющие, является ли стек пустым или переполненным. Их введение обусловлено тем, что переменная top, отражающая ту же информацию, уже недоступна пользователю. При описании данных функций используется модификатор const. Он явно указывает, что функция не изменит значений никаких членов класса. Модификатор const аргумента функции push указывает, что данный аргумент в функции не будет изменен.

Таким образом, введение ограничения доступа к элементам класса на практике реализует понятие инкапсуляции.

Инкапсуляция локализует те особенности проекта, которые могут подвергнуться изменениям. По мере развития системы разработчики могут решить, что какие-то операции выполняются несколько дольше,

чем допустимо, а какие-то объекты занимают больше памяти, чем приемлемо. В таких ситуациях часто изменяют внутреннее представление объекта. В результате становится возможным реализовать более эффективные алгоритмы либо оптимизировать алгоритм по критерию памяти, заменяя хранение данных их вычислением. Важным преимуществом ограничения доступа является возможность внесения изменений в объект без изменения других объектов.

Сокрытие информации – понятие относительное: то, что спрятано на одном уровне абстракции, обнаруживается на другом уровне. Кроме того, на практике иногда необходимо ознакомиться с реализацией класса, чтобы понять его назначение. Это особенно важно, если нет внешней документации. С другой стороны язык C++ предоставляет средства, позволяющие нарушить инкапсуляцию. Одним из таких средств является использование друзей класса.

## 2.3. МОДУЛЬНОСТЬ

При традиционном структурном подходе **модулем** называют набор связанных процедур вместе с данными, которые они обрабатывают. Необходимо разложить подпрограммы по модулям так, чтобы в один модуль попадали подпрограммы, использующие друг друга или изменяемые вместе.

В объектно-ориентированном программировании по модулям необходимо распределить классы и объекты.

В большинстве языков, поддерживающих принцип модульности, интерфейс модуля отделен от его реализации. Таким образом, принципы модульности и инкапсуляции являются взаимосвязанными.

В языке C++ модулями являются файлы, которые компилируются отдельно один от другого и затем объединяются в один исполняемый файл при помощи редактора связей.

**Пример.** В качестве примера рассмотрим модульную структуру программы, использующей стек.

Реализация стека и код пользователя будут находиться в раздельно компилируемых частях программы.

Как правило, объявления, описывающие интерфейс модуля, помещаются в так называемый **заголовочный файл**, имеющий характерное имя, которое отражает его использование. Заголовочный файл обычно включается и в файл с пользовательским кодом, и в файл с

реализацией модуля. Это достаточно простой и эффективный способ обеспечить идентичность представления интерфейса в обоих файлах. Включение информации об интерфейсе в файл с пользовательским кодом обусловлено необходимостью проверки типов используемых в нем интерфейсных функций во время компиляции.

Итак, интерфейс стека будет помещен в файл stack.h.

Интерфейс модуля стека, представленного в виде набора данных и функций (без использования понятия класс), включает в себя объявления (прототипы) функций, доступных пользователю стека. Таким образом, файл stack.h имеет следующее содержание

```
void push(int el);
int pop( );
```

Код пользователя будет находиться, например, в файле user.cpp:

```
#include "stack.h"        // включить интерфейс
main(void)
{
        push(1);
        if (pop( ) != 1) . . .; // ???
        . . .
}
```

Файл, содержащий реализацию модуля Stack, может называться, например, stack.cpp:

```
#include "stack.h"        // включить интерфейс
int stack [100]; //реализация
int top;
void push(int el){. . .}
int pop( ){. . .}
```

Тексты user.cpp и stack.cpp совместно используют информацию об интерфейсе, содержащуюся в stack.h. Во всем другом эти два файла независимы и могут быть раздельно откомпилированы. Графическое изображение упомянутых фрагментов программы представлено на рис. 2.1.

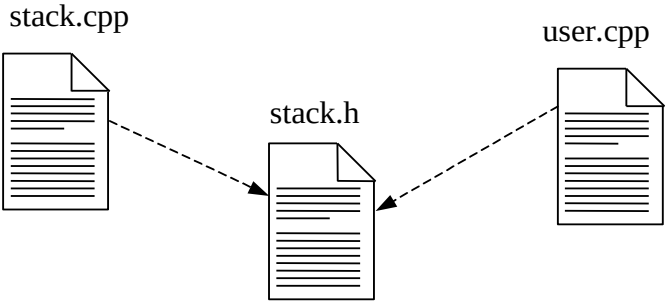


*Рис. 2.1.* Структура модулей программы, использующей стек

Если стек представлен в виде объекта типа Stack, введенного с использованием понятия класс, то информация о данных, агрегированных в этот новый тип, а не только о предоставляемом им интерфейсе, также должна быть доступна при компиляции пользовательского кода на языке C++.

Предположим, что компилятор встречает объявление объекта

```
Stack My_stack;
```

Компилятор должен знать, сколько отвести под него памяти. Если бы эта информация содержалась только в реализации класса, нам пришлось бы написать ее полностью, прежде чем мы смогли бы задействовать клиентов класса, т. е. весь смысл отделения интерфейса от реализации был бы потерян.

Таким образом, представление объекта в языке C++ определяется в интерфейсной части класса, а не в его реализации. В связи с этим вместо разделения интерфейс-реализация говорят о разделении описание-реализация. При этом описания всех используемых классов помещаются в заголовочный файл.

В результате файл stack.h должен содержать описание структуры или класса Stack, приведенное в разд. 2.2.

Файл stack.cpp, содержащий реализацию стека, имеет вид

```
#include "stack.h"
void Stack :: push(const int el) { …}
int Stack :: pop( ) { …}
bool Stack :: isFull( ) const { …}
bool Stack :: isEmpty( ) const { …}
```

Формат записи функций здесь включает двойное двоеточие (::) – оператор разрешения области видимости. С его помощью формируется **квалифицированное имя** члена класса

имя_класса :: имя_члена_класса

В данном случае использование квалифицированного имени функции позволяет указать ее принадлежность классу Stack и отличить от иных (возможно, одноименных) функций, принадлежащих или не принадлежащих каким-либо классам.

Заметим, что язык C++ позволяет включить реализацию функций-членов в описание класса

```
class String {
int length;
public:
        int getLength(void) const { return length; };
        …
};
```

Функции, описанные таким образом, обладают свойством подстановки и называются **подставляемыми (инлайн-) функциями**. Когда в тексте программы встречается вызов инлайн-функции, то вместо действительно ее вызова компилятор *может* подставить его текст.

Например, фрагмент

```
String s;
int x = s.getLength( );
```

в случае подстановки функции выполняется, как будто он был записан как

```
String s;
int x = s.length;
```

Подстановка осуществляется только для относительно простых коротких функций. Имеется ряд ограничений для применения подстановки.

При записи реализации функции вне описания класса можно указать свойство подстановки, используя ключевое слово inline.

```
inline int String :: getLength(void) const { return length; };
```

Правильное разделение программы на модули является сложной проблемой. Для небольших задач допустимо наличие одного модуля. Однако для большинства программ лучшим решением будет сгруппировать логически связанные элементы в отдельный модуль. При этом следует оставить открытыми только те элементы, которые совершенно необходимо видеть другим модулям. Заметим, что деление программы на модули бессистемным образом иногда гораздо хуже, чем отсутствие модульности вообще.

Рассмотрим приемы и правила, которые позволяют составлять модули наиболее эффективным образом:

- конечной целью разбиения программы на модули является снижение затрат на программирование за счет независимой разработки и тестирования;
- структура модуля должна быть достаточно простой для восприятия;
- реализация каждого модуля не должна зависеть от реализации других модулей;
- должны быть приняты меры для облегчения процесса внесения изменений там, где они наиболее вероятны.

Программист должен находить баланс между двумя противоположными тенденциями: стремлением скрыть информацию и необходимостью обеспечения видимости тех или иных абстракций в нескольких модулях. Для этого используют следующие правила:

- особенности системы, подверженные изменениям, следует скрывать в отдельных модулях;
- в качестве межмодульных можно использовать только те элементы, вероятность изменения которых мала;
- все структуры данных должны быть обособлены в модуле; доступ к ним будет возможен для всех процедур этого модуля и закрыт для всех других;
- доступ к данным из модуля должен осуществляться только через процедуры данного модуля.

Следует стремиться построить модули так, чтобы объединить логически связанные абстракции и минимизировать взаимные связи между модулями.

На выбор разбиения на модули могут влиять и некоторые внешние обстоятельства. При коллективной разработке программ распределение работы осуществляется, как правило, по модульному принципу и правильное разделение проекта минимизирует связи между участниками. Абстракции можно распределить так, чтобы быстро установить интерфейсы модулей по соглашению между группами, участвующими в работе. Внесение изменений в интерфейс одной подсистемы приводит к необходимости модификации других подсистем и изменений в их документации, все эти факторы требуют от интерфейса консерватизма.

Могут сказываться и требования секретности: одна часть кода может быть несекретной, а другая – секретной, тогда последняя выполняется в виде отдельного модуля (модулей).

В результате всего сказанного сформулируем следующее определение модульности.

**Модульность** – это свойство системы, разложенной на цельные, но слабо связанные между собой модули.

Большие системы могут быть разложены на несколько сотен, если не тысяч, модулей. Пытаться разобраться в физической архитектуре такой системы без ее дополнительного структурирования почти невозможно. По этой причине удобно ввести понятие подсистемы. Подсистемы представляют собой совокупности логически связанных модулей.

**Подсистема** – это агрегат, содержащий другие модули и другие подсистемы. Каждый модуль в системе должен располагаться в одной подсистеме или находиться на самом верхнем уровне.

Некоторые модули подсистемы могут быть общедоступны, т. е. экспортированы из системы и видимы снаружи. Другие модули могут быть частью реализации подсистемы и не использоваться внешними модулями.

## 2.4. ИЕРАРХИЧНОСТЬ

Абстракция является полезным инструментом. Однако всегда, кроме самых простых ситуаций, число абстракций в системе намного превышает наши умственные возможности. Инкапсуляция позволяет в

какой-то степени устранить это препятствие, убрав из поля зрения внутреннее содержание абстракций. Модульность также упрощает задачу, объединяя логически связанные абстракции в группы. Но этого оказывается недостаточно.

Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций иерархической структуры.

**Иерархия** – ранжированная или упорядоченная система абстракций.

Принцип **иерархичности** предполагает использование иерархических структур при разработке программных систем.

Основными видами иерархических структур применительно к сложным системам являются иерархии типа «является» и иерархии типа «имеет».

**Иерархия «является»** подразумевает, что элемент, стоящий на нижнем уровне абстракции, является разновидностью (частным случаем) элемента, стоящего на верхнем уровне.

Например, лазерный принтер *является* разновидностью принтеров: лазерный принтер *является* принтером; принтер HP LaserJet 1020 *является* разновидностью лазерных принтеров: принтер HP LaserJet 1020 *является* лазерным принтером (рис. 2.2). Понятие «принтер» обобщает свойства, присущие всем принтерам, а лазерный принтер – это просто особый тип принтера со свойствами, которые отличают его, например, от матричного или струйного принтера.

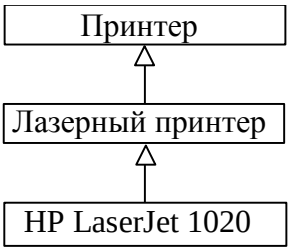


*Рис. 2.2.* Иерархия «является» для принтеров

Важный элемент объектно-ориентированных систем и основной вид иерархии «является» – иерархия наследования (отношение родитель–потомок).

Наследование означает такое отношение между абстракциями, когда абстракция-потомок заимствует структурную и/или функциональную часть одной или нескольких абстракций-родителей. Если абстракция-потомок заимствует часть одной абстракции-родителя, то говорят об **одиночном наследовании.** Если же потомок заимствует части нескольких родителей, то говорят о **множественном наследовании**. Часто потомок достраивает или переписывает компоненты родителя.

«Лакмусовой бумажкой» наследования является обратная проверка. Если B не есть A, то B не стоит производить от A.

В наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общей абстракции. Потомок представляет собой специализированный частный случай своего предка. По этой причине говорят о наследовании как об иерархии *обобщение–специализация.* Таким образом, абстракция, стоящая на верхнем уровне, является обобщением для нижестоящей, а нижестоящая – специализацией вышестоящей.

Принцип наследования позволяет упростить выражение абстракций, делает проект менее громоздким и более выразительным. В отсутствие наследования каждая часть сложной системы становится самостоятельным блоком и должна разрабатываться «с нуля». Абстракции лишаются общности, поскольку каждый программист реализует их по-своему. Стройность системы достигается тогда только за счет дисциплинированности программистов.

С другой стороны, принципы абстрагирования, инкапсуляции и иерархичности находятся между собой в некоем здоровом конфликте. Абстрагирование и инкапсуляция создают непрозрачный барьер, скрывающий состояние и функции объекта; принцип наследования требует открыть доступ и к состоянию, и к функциям объекта для производных объектов.

**Пример.** Одиночное наследование. Вернемся к иерархии «принтер – лазерный принтер» (лазерный принтер является разновидностью принтеров).

Абстракция «лазерный принтер» строится на основе родительской абстракции «принтер». «Лазерный принтер» наследует от «принтера» свойства, определяющие все принтеры. Кроме того, лазерный принтер имеет структурные и функциональные части, реализующие свойства, которые характерны именно для лазерных принтеров.

**Пример.** Множественное наследование. Введем абстракции «временный работник» и «секретарь». Секретарь может быть постоянным работником или временным. В последнем случае абстракция «временно работающий секретарь» наследует компоненты обеих абстракций. Временно работающий секретарь выполняет обязанности секретаря и имеет правовой статус временного работника.

Множественным наследованием часто злоупотребляют. Например, сладкая вата – это частный случай сладости, но никак не ваты. Следует

применять ту же «лакмусовую бумажку»: если B не есть A, то ему не стоит наследовать от A.

**Иерархия «имеет»** вводит отношение агрегации (*целое/часть*). В иерархии «имеет» некоторая абстракция находится на более высоком уровне, чем любая из использовавшихся при ее реализации.

Агрегация есть во всех языках, использующих структуры или записи, состоящие из разнотипных данных. Но в объектно-ориентированном программировании она обретает новую мощь: агрегация позволяет физически сгруппировать логически связанные структуры, а наследование с легкостью копирует эти общие группы в различные абстракции.

**Пример.** Компьютер (рис. 2.3) *имеет* системный блок (системный блок является частью компьютера). Системный блок компьютера одновременно имеет (агрегирует) материнскую плату, платы оперативной памяти, центральный процессор и множество других компонентов. Заметим, что от замены процессора на более мощный, от добавления нескольких плат оперативной памяти или второго жесткого диска системный блок не становится другим системным блоком. Если же мы разбираем системный блок, мы уничтожаем его как объект, однако его компоненты остаются и могут быть использованы в других системных блоках. Другими словами, системный блок и его компоненты имеют свои отдельные и независимые сроки жизни.

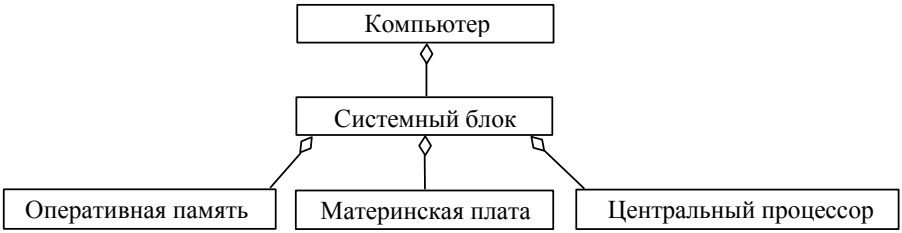


*Рис. 2.3.* Компьютер: иерархия «имеет»

## 2.5. ТИПИЗАЦИЯ

**Типизация** – это способ защититься от использования объектов одного класса (типа) вместо другого или, по крайней мере, управлять таким использованием.

Идея согласования типов занимает в понятии типизации центральное место. Возьмем, к примеру, физические единицы измерения. Разделив расстояние на время, мы ожидаем получить скорость, а не вес. В умножении температуры на силу смысла нет, а в умножении расстояния на силу есть. Все это примеры сильной типизации, когда прикладная область диктует правила и ограничения на использование и сочетание абстракций.

Рассмотрим следующий фрагмент:

```
typedef  char* Pchar;
Pchar p1, p2;
char *p3 = p1;
```

Поскольку объявление, начинающееся с ключевого слова typedef, вводит новое имя для типа, эти имена можно свободно смешивать в вычислениях. В этом смысле C++ имеет слабую типизацию.

При проверке типов у классов C++ типизирован гораздо строже. Выражения, содержащие вызовы операций, проверяются на согласование типов во время компиляции.

Важным понятием объектно-ориентированного подхода в целом и языка C++ в частности является полиморфизм.

**Полиморфизм** – это способ присваивать различные значения (смыслы) одному и тому же сообщению (функции, оператору). Смысл зависит от типа обрабатываемых данных.

Имеется несколько типов полиморфизма.

**Принудительное приведение.** Функция или оператор работает с несколькими различными типами, преобразуя их значения к требуемому типу. Например,

```
int i = 1;
double a, b = 4.5;
a = b + i;
```

В данном примере значение переменной i будет преобразовано к типу double и результат сложения также будет иметь тип double. Заметим, что значение i в памяти останется неизменным, преобразуется только временная копия i, используемая при вычислении значения выражения.

**Перегрузка.** Функция или оператор вызывается на основе сигнатуры. Например,

```
double a;
a = 1/2; // целочисленное деление, a = 0
a = 1./2.; // деление вещественных чисел, a = 0.5
```

Если в описание класса ввести определение функции-члена с именем типа «operator *оператор*», то это означает, что данный *оператор* может быть применен к объектам или объекту данного класса, так же как и к переменным стандартных типов. При этом тело данной функции определяет смысл оператора. Например:

```
class complex {
    double re, im;
public:
    . . .
    complex operator+(complex);
    complex operator*(complex);
 };
```

Мы определили простую реализацию понятия комплексного числа: число представляется парой чисел с плавающей точкой двойной точности, вычисления осуществляются посредством операций + и *. Теперь, определив переменные b и c типа complex, можно записать b+c, что означает (по определению) b.operator+(c). В результате появляется возможность записывать комплексные выражения в форме, близкой к общепринятой.

Другие типы полиморфизма – **включение и параметрический полиморфизм** – мы рассмотрим в разд. 4.4 и 4.5 соответственно.

Для осуществления явных преобразований переменных одного типа к другому типу в C++ имеются специальные **операторы приведения**.

Оператор static_cast используется для преобразования родственных типов и позволяет провести преобразование типа корректно, переносимо и обратимо. Например,

```
int i;
double a, b;
. . .
a = static_cast < double > (i);
a = static_cast < double > (static_cast < int > (b) + 1);
```

Оператор reinterpret_cast позволяет провести явное преобразование между несвязанными (неродственными) типами. Например,

```
i = reinterpret_cast < int > (&x);// системно-зависимое
```

Использование модификатора const приводит к тому, что значение переменной нельзя изменить. Если данное ограничение необходимо обойти, используется оператор const_cast. Например, он используется при передаче в функцию константного указателя на место формального параметра, не имеющего модификатора const.

Преобразование static_cast предполагает, что типы, участвующие в преобразовании, известны во время компиляции. В случаях когда это не так, используется оператор приведения dynamic_cast. Данные ситуации мы рассмотрим в разд. 4.4.

## 2.6. ПАРАЛЛЕЛИЗМ

Есть задачи, в которых автоматические системы должны обрабатывать много событий одновременно. В других случаях потребность в вычислительной мощности превышает ресурсы одного процессора. В каждой из таких ситуаций естественно использовать несколько компьютеров для решения задачи или задействовать многозадачность на многопроцессорном (многоядерном) компьютере.

**Процесс (поток управления)** – это фундаментальная единица действия в системе. Каждая программа имеет, по крайней мере, один поток управления, в параллельной системе таких потоков много. Век одних потоков недолог, а другие живут в течение всего сеанса работы системы.

Параллелизм главное внимание уделяет абстрагированию и синхронизации процессов.

Объект, полученный из абстракции реального мира, может представлять собой отдельный поток управления (т. е. абстракцию процесса). Такой объект называется **активным.**

Для систем, построенных на основе объектно-ориентированного проектирования, мир может быть представлен как совокупность взаимодействующих объектов, часть из которых является активной и выступает в роли независимых вычислительных центров.

**Параллелизм** – свойство нескольких объектов одновременно находиться в активном состоянии.

Например, управление в теплице может быть централизованным, когда центральный компьютер регулярно обращается к объектам типа TemperatureSensor, заставляя их измерять и сообщать текущую температуру. В децентрализованном варианте управления может быть использован датчик ActiveTemperatureSensor, который сам периодически измеряет температуру и сообщает, что она отклонилась от установленного диапазона. Во втором случае, очевидно, требуется параллелизм.

## 2.7. СОХРАНЯЕМОСТЬ

Любой программный объект существует в памяти и живет во времени.

Существуют объекты, которые присутствуют лишь во время вычисления выражения. Но есть и такие (например, базы данных), которые существуют независимо от программы. Временной спектр сохраняемости объектов охватывает следующее:

- промежуточные результаты вычисления выражений;
- локальные переменные в вызове процедур;
- глобальные переменные и динамически создаваемые данные;
- данные, сохраняющиеся между сеансами выполнения программы;
- данные, сохраняемые при переходе на новую версию программы;
- данные, которые *вообще* переживают программу.

По традиции, первыми тремя уровнями занимаются языки программирования, а последними – базы данных. Языки программирования, как правило, не поддерживают понятия сохраняемости. Можно записывать объекты в неструктурированные файлы, но этот подход пригоден только для небольших систем. Как правило, сохраняемость достигается применением специальных объектно-ориентированных баз данных.

До сих пор мы говорили о сохранении объектов во времени. В большинстве систем объектам при их создании отводится место в памяти, которое не изменяется и в котором объект находится всю свою жизнь. Однако иногда необходимо обеспечивать возможность перемещения объектов в пространстве так, чтобы их можно было переносить с машины на машину и изменять форму представления объекта в памяти. Это касается систем, распределенных в пространстве.

**Сохраняемость** – свойство объекта существовать во времени независимо от процесса, породившего данный программный объект, и/или в пространстве, перемещаясь из адресного пространства, в котором он был создан.

# 3. ОБЪЕКТЫ

**Объект** – отдельно реализуемая часть предметной области задачи. Объект существует во времени и пространстве. Объект обладает состоянием, поведением и идентичностью; структура и поведение схожих объектов определяют общий для них класс; термины «экземпляр класса» и «объект» взаимозаменяемы.

## 3.1. СОСТОЯНИЕ

**Пример.** Рассмотрим торговый автомат, продающий напитки. Поведение такого объекта состоит в том, что после опускания в него монеты и нажатия кнопки автомат выдает выбранный напиток. Предположим, что сначала нажата кнопка выбора напитка, а потом уже опущена монета. Большинство автоматов при этом просто ничего не сделают, так как пользователь нарушил их основные правила. Таким образом автомат играл роль (ожидание монеты), которую пользователь игнорировал, нажав сначала кнопку. Предположим далее, что пользователь автомата не обратил внимания на предупреждающий сигнал «Бросьте столько мелочи, сколько стоит напиток» и опустил в автомат лишнюю монету. В большинстве случаев автоматы не дружественны к пользователю и радостно заглатывают все деньги.

В каждой из таких ситуаций поведение объекта определяется его историей: важна последовательность совершаемых над объектом действий. Такая зависимость поведения от событий и от времени объясня-

ется тем, что у объекта есть внутреннее состояние. Для торгового автомата, например, состояние определяется суммой денег, опущенных до нажатия кнопки выбора. Другая важная информация – это набор воспринимаемых монет и запас напитков. На основе этого примера дадим следующее определение.

**Состояние** объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств. В число свойств входят атрибуты объекта и атрибуты всех его агрегированных частей.

Одним из свойств торгового автомата является способность принимать монеты. Это статическое (фиксированное) свойство, в том смысле, что оно – существенная характеристика торгового автомата. С другой стороны, этому свойству соответствует динамическое значение, характеризующее количество принятых монет. Сумма увеличивается по мере опускания монет в автомат и уменьшается, когда продавец забирает деньги из автомата.

В некоторых случаях значения свойств объекта могут быть статическими (например, заводской номер автомата), поэтому в данном определении использован термин «обычно динамическими».

К числу свойств объекта относятся присущие ему или приобретаемые им характеристики, черты, качества или способности, делающие данный объект самим собой. Например, для лифта характерным является то, что он сконструирован для поездок вверх и вниз, а не горизонтально.

Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу объекта. Мы говорим «как правило», потому что в ряде случаев состав свойств объекта может изменяться. Примером может служить робот с возможностью самообучения. Робот первоначально может рассматривать некоторое препятствие как статическое, а затем обнаруживает, что это дверь, которую можно открыть. В такой ситуации по мере получения новых знаний изменяется создаваемая роботом модель мира.

Все свойства имеют некоторые значения. Эти значения могут быть простыми количественными характеристиками, а могут ссылаться на другой объект. Состояние лифта может описываться числом 3, означающим номер этажа, на котором лифт в данный момент находится. Состояние торгового автомата описывается в терминах других объектов, например имеющихся в наличии напитков. Конкретные напитки – это самостоятельные объекты, отличные от торгового автомата.

## 3.2. ПОВЕДЕНИЕ

Объекты не существуют изолированно, а подвергаются воздействию или сами воздействуют на другие объекты.

**Поведение** – это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений. Поведение объекта – это его наблюдаемая и проверяемая извне деятельность.

Операция – это услуга, которую можно запросить у любого объекта класса для воздействия на его поведение.

Например, клиент может активизировать операции push и pop для того, чтобы управлять объектом-стеком (добавить или изъять элемент).

В чисто объектно-ориентированном языке принято говорить о **передаче сообщений** между объектами. В C++ мы говорим, что один объект вызывает функцию-член другого. В основном понятие «сообщение» совпадает с понятием «операция над объектами».

Передача сообщений – это один уровень, задающий поведение. Из нашего определения следует, что состояние объекта также влияет на его поведение.

Рассмотрим торговый автомат. Мы можем сделать выбор, но поведение автомата будет зависеть от его состояния. Если мы не опустили в него достаточную сумму, скорее всего, ничего не произойдет. Если же денег достаточно, автомат выдаст нам желаемое (и тем самым изменит свое состояние).

Некоторые операции изменяют состояние. В связи со сказанным можно заключить, что состояние объекта представляет суммарный результат его поведения.

Операция – это услуга, которую класс может предоставить своим клиентам. На практике типичный клиент совершает над объектами операции следующих видов:

- **модификатор** – это операция, которая изменяет состояние объекта (например, set-функция);
- **селектор** – это операция, считывающая состояние объекта, но не меняющая состояние (например, get-функция);
- **конструктор** – это операция создания объекта и/или его инициализации; в C++ конструктор имеет то же имя, что и класс;
- **деструктор** – это операция, освобождающая ресурсы, которые использует объект, и/или разрушающая сам объект; в C++ имя дест-

руктора состоит из имени класса, перед которым ставится знак «тильда» – «~».

Две последние операции являются универсальными. Они обеспечивают инфраструктуру, необходимую для создания и уничтожения экземпляров класса. Если у класса есть конструктор, то он вызывается всегда, когда создается объект класса. Если у класса есть деструктор, то он вызывается всегда, когда объект класса уничтожается. Если программист не описал в классе конструктор и деструктор, то они будут созданы автоматически.

Объекты могут создаваться следующим образом:

- *автоматический объект* создается каждый раз, когда его описание встречается при выполнении программы, и уничтожается каждый раз при выходе из блока, в котором оно появилось;
- *статический объект* создается один раз, при запуске программы, и уничтожается один раз, при ее завершении;
- *объект в свободной памяти* создается с помощью операции new и уничтожается с помощью операции delete;
- *объект-член* создается как подобъект другого класса.

**Пример.** Расширим описание класса Stack, с тем чтобы программист мог задавать максимальный размер каждого создаваемого стека (размер массива s).

```
class Stack {
int *s, length, top;
. . .
public:
        Stack(int  n = 100){ // конструктор, n – максимальный размер,
                // значение максимального размера по умолчанию – 100
        length = n; s = new int [length]; top = 0;}
        ~Stack( ) { delete [ ] s; } // деструктор
        void push(const int el); // модификатор
        int pop( ); // модификатор
        bool isFull( ) const; // селектор
        bool isEmpty( ) const; // селектор
        . . .
};
```

Теперь мы можем объявить нужные нам стеки:

```
int len=100;
Stack st1(len), st2(200);
```

Конструктор с одним аргументом может служить также для преобразования типа своего аргумента в тип конструктора.

**Пример.** Рассмотрим определение класса complex.

```
class complex {
        double re, im;
public:
        complex(double r, double i);
        complex(double r);
        . . .
 };
```

Мы определили два конструктора, один из которых имеет один аргумент и служит для инициализации комплексного числа (его действительной части) значением вещественного числа. Теперь мы можем записать два эквивалентных оператора

```
complex a = complex(1);
complex a = 1;
```

Последнее присваивание имеет однозначный смысл с точки зрения предметной области. В то же время для стека аналогичное присваивание

```
Stack st = 100;
```

будет неоднозначным по смыслу и может быть потенциальным источником ошибок.

Можно запретить использование конструктора для таких преобразований, объявив его с ключевым словом explicit.

```
class Stack {
 public:
    explicit Stack(int n = 100); // конструктор, задающий максимальный
    . . .  // размер стека, нельзя использовать для преобразования
};
```

Для инициализации отдельных частей объекта с помощью конструктора служат **инициализаторы конструктора**. Важность инициализаторов в том, что только с их помощью можно инициализировать константные члены, члены, являющиеся ссылками, а также члены, являющиеся объектами класса, в котором есть один или несколько конструкторов, но отсутствует конструктор по умолчанию (конструктор без параметров).

Важно также, что такая инициализация выполняется эффективнее, поскольку создание объекта в C++ начинается с инициализации его атрибутов конструктором по умолчанию, после чего выполняется вызываемый конструктор. Использование инициализаторов позволяет сразу же вызвать нужный конструктор.

Для инциализаторов используется синтаксис следующего примера:

```
class Id_Stack {
const int id; Stack s;
public: Id_Stack(int i, int n): id(i), s(n){ };
};
```

В чисто объектно-ориентированных языках определять процедуры и функции вне классов не допускается. В гибридных языках, выросших из процедурных языков, таких как C++, допускается описывать операции как независимые от объектов подпрограммы.

Операции, определенные вне классов, называют **свободными подпрограммами.** В C++ они называются функциями-нечленами.

```
bool check_stack(Stack & my_stack, int el)
{
        Stack temp_stack;
        . . .  // используя дополнительный стек temp_stack, проверить,
                //есть ли в my_stack элемент el
}
```

Свободные подпрограммы – это процедуры и функции, которые выполняют роль операций высокого уровня над объектом или объектами одного или разных классов. Свободные подпрограммы обычно группируются в соответствии с классами, для которых они создаются.

## 3.3. ИДЕНТИЧНОСТЬ

**Идентичность** – это такое свойство объекта, которое отличает его от всех других объектов.

Источником ошибок в объектно-ориентированном программировании является неумение отличать имя объекта от самого объекта.

**Пример.** Определим точку на плоскости.

```
struct Point {
        int x, y; // координаты
        Point(void); // конструктор по умолчанию (0,0)
        Point(int xValue, int yValue); // конструктор
};
```

Теперь определим точку, отображаемую на экране дисплея (DisplayPoint). Ограничимся возможностями рисовать точку и перемещать ее по экрану, а также запрашивать ее положение. Мы записываем нашу абстракцию в виде следующего объявления на C++:

```
class DisplayPoint {
public:
    DisplayPoint( ); // конструктор по умолчанию (0,0)
    DisplayPoint(const Point& location); // конструктор
    ~DisplayPoint( ); // деструктор
    void draw( ); // рисует точку на экране
    void move(const Point& location); // перемещает точку
    Point location( ); // возвращает координаты
    . . .
};
```

Аргументы некоторых функций указаны с модификатором const. Он указывает, что значение объекта, передаваемого по ссылке или

указателю, в функции не изменится. Литералы, константы и аргументы, требующие преобразования типа, можно передавать как const&-аргументы и нельзя – не в качестве const&-аргументов.

Объявим экземпляры класса DisplayPoint:

```
DisplayPoint Item1;
DisplayPoint * Item2 = new DisplayPoint(Point(75,75));
DisplayPoint * Item3 = new DisplayPoint(Point(100,100));
DisplayPoint * Item4 = 0;
```

При выполнении этих операторов возникают четыре имени и три разных объекта (рис. 3.1, *а*). В памяти будут отведены четыре места под имена Item1, Item2, Item3, Item4. При этом Item1 будет именем объекта класса DisplayPoint, а три других – *указателями.* Кроме того, лишь Item2 и Item3 будут на самом деле указывать на объекты класса. У объектов, на которые указывают Item2 и Item3, к тому же нет имен, хотя на них можно ссылаться, «разыменовывая» соответствующие указатели (например, *Item2). Поэтому мы можем сказать, что Item2 указывает на отдельный объект класса, на имя которого мы можем косвенно ссылаться через *Item2.

Уникальная идентичность каждого объекта сохраняется на все время его существования, даже если его внутреннее состояние изменилось. При этом имя объекта не обязательно сохраняется.

Рассмотрим результат выполнения следующих операторов (рис. 3.1, *б*):

```
Item1.move(Item2 -> location( ));
Item4 = Item3;
Item4 -> move(Point(38, 100));
```

Объект Item1 и объект, на который указывает Item2, теперь относятся к одной и той же точке экрана. Указатель Item4 стал указывать на тот же объект, что и Item3. Хотя объект Item1 и объект, на который указывает Item2, имеют одинаковое состояние, они остаются разными объектами. Кроме того, мы изменили состояние объекта *Item3, использовав его новое косвенное имя Item4.

Ситуацию, когда объект именуется более чем одним способом несколькими синонимичными именами, называют **структурной зависимостью.**

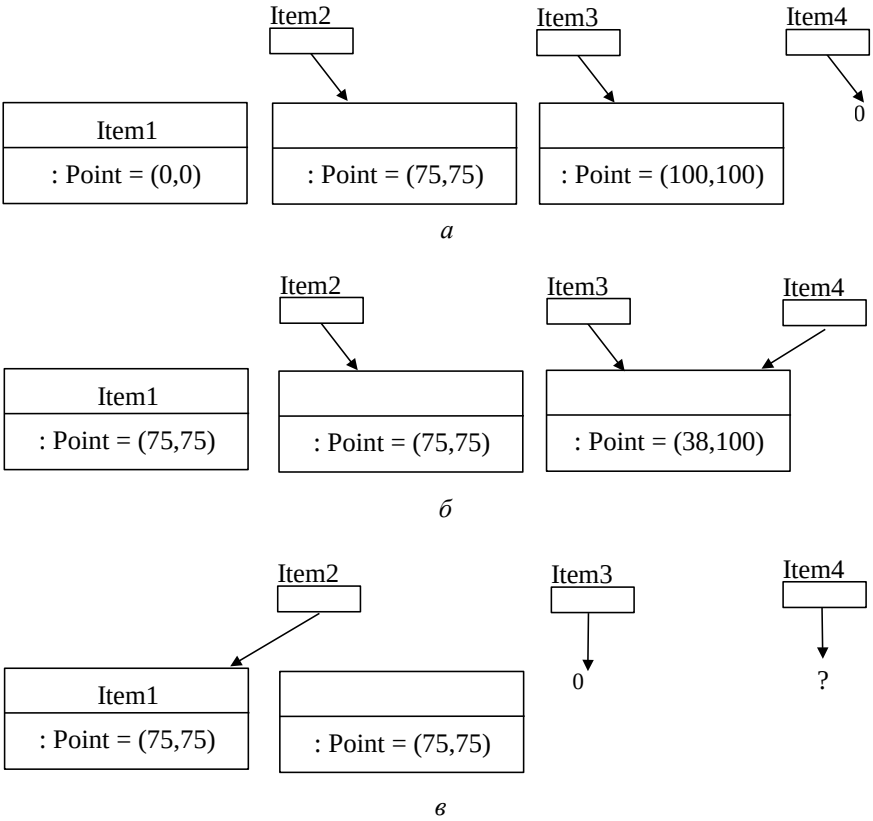


*Рис. 3.1.* Идентичность объектов

Структурная зависимость порождает в объектно-ориентированном программировании много проблем. Трудность распознания побочных эффектов при действиях с синонимичными объектами часто приводит к утечкам памяти, неправильному доступу к памяти и, хуже того, непрогнозируемому изменению состояния. Рассмотрим результат выполнения следующих действий (рис. 3.1, *в*):

```
delete Item3;
Item2 = &Item1;
```

В первой строке мы уничтожили объект через указатель Item3, теперь значение указателя Item4 оказывается бессмысленным. Эта ситуация называется висячей ссылкой.

Во второй строке создается синоним: Item2 указывает на тот же объект, что и Item1. К сожалению, при этом произошла утечка памяти: объект, на который первоначально указывал Item2, не именуется ни прямо, ни косвенно и его *идентичность потеряна*.

В языках типа C++ такая память освобождается только тогда, когда завершается программа, создавшая объект. Такие утечки памяти могут вызвать и просто неудобство, и крупные сбои, особенно если программа должна непрерывно работать длительное время. Представьте себе утечку памяти в программе управления спутником. Перезапуск компьютера на спутнике в нескольких миллионах километров от Земли очень неудобен.

Для создания нового объекта, имеющего то же состояние, что и у существующего, необходимо вызвать конструктор копирования, имеющий следующее описание:

```
DisplayPoint(const DisplayPoint &); // конструктор копирования
```

Отсутствие этого специального конструктора вызывает копирующий конструктор, действующий по умолчанию, который копирует объект поэлементно. Это разумно не всегда. Когда объект содержит ссылки или указатели на другие объекты, такая операция приводит к созданию синонимов указателей на эти объекты.

**Пример.** Модифицируем описание класса DisplayPoint так, чтобы каждый его экземпляр содержал указатель на точку:

```
class DisplayPoint {
        . . .
        Point * DPoint
        . . .
};
. . .
DisplayPoint Item1;
DisplayPoint Item2(Item1); // вызов конструктора копирования
```

Поэлементное копирование объекта Item1 приведет к тому, что указатели на агрегированные объекты типа Point у обоих объектов Item1 и Item2 будут указывать на один и тот же объект, содержащий местоположение отображаемой точки (рис. 3.2). Фактически имеем структурную зависимость: оба объекта будут ответственны за отображение одной и той же точки. Этого ли мы хотели достичь?

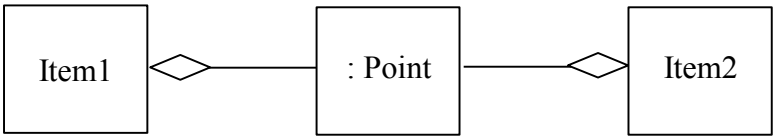


*Рис. 3.2.* Результат поэлементного копирования

Присваивание – это тоже копирование, и в C++ его смысл можно изменять. Например, мы могли бы добавить в определение класса DisplayPoint следующую строку:

```
DisplayPoint operator=(const DisplayPoint &);
```

Теперь мы можем записать

```
DisplayPoint Item5;
Item5 = Item1;
```

Как и в случае копирующего конструктора, если оператор присваивания не переопределен явно, то по умолчанию объект копируется поэлементно.

Понятие идентичности тесно связано с вопросом равенства. Равенство можно понимать двумя способами. Во-первых, два имени могут обозначать один и тот же объект (Item1 и Item2 на рис. 3.1, *в*). Во-вторых, это может быть равенство состояний у двух разных объектов (Item1 и Item2 на рис. 3.1, *б*).

В C++ нет предопределенного оператора равенства, поэтому мы должны определить равенство и неравенство, объявив эти операторы при описании:

```
int operator ==(DisplayPoint&) const;
int operator !=(DisplayPoint&) const;
```

## 3.4. ОТНОШЕНИЯ МЕЖДУ ОБЪЕКТАМИ

Сами по себе объекты не представляют никакого интереса: только в процессе их взаимодействия реализуется система. Например, самолет – это «совокупность элементов, каждый из которых по своей природе стремится упасть на землю, но за счет совместных непрерывных усилий преодолевающих эту тенденцию». Он летит только благодаря согласованным усилиям своих компонентов.

Отношения двух любых объектов основываются на предположениях, которыми один обладает относительно другого: об операциях, которые можно выполнять, и об ожидаемом поведении. Особый интерес для объектно-ориентированной технологии представляют два типа отношений между объектами: связь и агрегация.

Объект сотрудничает с другими объектами, посылая сообщения через связи, соединяющие его с ними. **Связь** – это конкретное соединение, через которое объект (клиент) запрашивает услугу у другого объекта (сервера) или управляет им.

Пусть есть два объекта A и B и связь между ними. Чтобы A мог послать B сообщение, B должен быть в каком-то смысле видим для A.

Перечислим следующие четыре способа обеспечить **видимость:**

- сервер глобален по отношению к клиенту;
- сервер (или указатель на него) передан клиенту в качестве параметра операции;
- сервер является частью клиента;
- сервер локально порождается клиентом в ходе выполнения какой-либо операции.

Если связи обозначают равноправные или «клиент-серверные» отношения между объектами, то агрегация описывает отношения целого и части, приводящие к соответствующей иерархии объектов, причем, идя от целого (агрегата), мы можем прийти к его частям (атрибутам).

**Пример.** Рассмотрим класс объектов, управляющих температурой в теплице – Controller. Пусть он имеет атрибут h класса Heater (нагреватель).

```
class Controller {
        Heater h;
        . . .
} C;
```

В данном случае объект C – целое, а h – его часть (часть его состояния). Исходя из контроллера, можно найти соответствующий нагреватель. Однако по h нельзя найти содержащий его объект (называемый также его контейнером), если только сведения о нем случайно не являются частью состояния h.

Агрегация может означать физическое вхождение одного объекта в другой, но не обязательно. Самолет состоит из крыльев, двигателей, шасси и прочих частей. С другой стороны, отношения акционера с его акциями – это агрегация, которая не предусматривает физического включения. Акционер монопольно владеет своими акциями, но они в него не входят физически.

# 4. КЛАССЫ

Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие этих двух понятий. В то время как объект обозначает конкретную сущность, определенную во времени и в пространстве, класс определяет лишь абстракцию существенного в объекте.

**Класс** – это множество объектов, имеющих общую структуру и общее поведение. Любой конкретный объект является экземпляром класса.

**Пример.** Рассмотрим сходства и различия между следующими классами: цветы, маргаритки, красные розы, желтые розы, лепестки и пчелы. Мы можем заметить следующее:

- маргаритка – цветок;
- роза – (другой) цветок;
- красная и желтая розы – розы;
- лепесток является частью обоих видов цветов;
- пчелы опыляют цветы и питаются их нектаром.

Из этого простого примера следует, что классы, как и объекты, не существуют изолированно. В каждой проблемной области абстракции, описывающие ее, взаимодействуют различными способами.

Известны три основных типа отношений между классами. Во-первых, это отношение «обобщение/специализация» (общее и частное), т. е. иерархия «является». Розы являются цветами, что значит:

розы являются специализированным частным случаем, подклассом более общего класса «цветы». Во-вторых, это отношение «целое/часть», т. е. иерархия «имеет». Например, лепестки являются частью цветов. В-третьих, это семантические, смысловые отношения, ассоциации. Например, пчелы ассоциируются с цветами.

Рассмотрим подробно следующие отношения между классами: ассоциация, агрегация, зависимость, наследование, инстанцирование.

## 4.1. АССОЦИАЦИЯ

Среди указанных отношений ассоциация – наиболее абстрактное и наиболее слабое отношение. Идентификация ассоциаций между классами часто проводится на ранних этапах разработки программы, когда необходимо обнаружить общие зависимости между абстракциями. В последующем эти слабые ассоциации часто уточняются и воплощаются в виде одного из более конкретных отношений между классами.

**Ассоциация** – семантическая (смысловая) зависимость, в которой не указывается направление (по умолчанию подразумевается двусторонняя связь) и не объясняется, как классы связаны друг с другом (их семантику можно только подразумевать, указав роли классов в этих отношениях). Создавая ассоциации, разработчик системы фиксирует участников семантических отношений, а также их роли и количество.

**Пример.** Рассмотрим два класса – «Компания» и «Человек». Если рассматривать человека как сотрудника компании, то смысл такой ассоциации – «Место работы» человека, человек выступает в роли «Работника», а компания – в роли «Работодателя». Однако между данными классами может быть и другая ассоциация: если человек является владельцем компании (причем одновременно может являться и ее работником), то он выступает в роли «Собственника», а компания – в роли его «Собственности», смысл ассоциации «Капитал».

**Кратность (мощность) ассоциации** – это количество ее участников. Различают три случая кратности ассоциации: «один-к-одному», «один-ко-многим», «многие-ко-многим».

Ассоциация «Продажа–Товары» имеет тип «один-ко-многим»: каждый экземпляр товара относится только к одной продаже, в то время

как каждой продаже может соответствовать совокупность проданных товаров.

Отношение «один-к-одному» обозначает очень узкую ассоциацию. Например, в розничной системе продаж примером могла бы быть связь между классом «Продажа» и классом «Снятие денег с кредитной карточки»: каждая продажа соответствует ровно одному снятию денег с данной кредитной карточки.

Отношение «многие-ко-многим» тоже нередко. Например, каждый объект класса «Покупатель» может инициировать сделку с несколькими объектами класса «Продавец», и каждый «Продавец» может взаимодействовать с несколькими объектами класса «Покупатель».

Класс может иметь ассоциацию с самим собой. Такая ассоциация называется **рефлексивной.**

## 4.2. АГРЕГАЦИЯ

Отношение агрегации между классами имеет непосредственное отношение к агрегации между их экземплярами.

**Пример.** Вернемся к классу Controller, который является абстракцией объектов, управляющих температурой в теплице (см. разд. 3.4).

Класс Controller – это целое, а экземпляр класса Heater (нагреватель) – одна из его частей. В рассмотренном случае мы имеем специальный случай агрегации – композицию.

**Композиция** – форма агрегирования, в которой целое владеет своими частями, имеющими одинаковое с ним время жизни. Части с нефиксированной кратностью могут быть созданы после создания агрегата, но, будучи созданными, живут и умирают вместе с ним. Такие части могут также быть явно удалены перед уничтожением агрегата. В случае композитного агрегирования объект в любой момент времени может быть частью только одного композита.

С точки зрения реализации в языке C++ композиция может быть осуществлена включением атрибута-части в класс-агрегат **по значению**, как это сделано в примере, приведенном в разд. 3.4. Менее обязывающим является включение **по ссылке.** Мы могли бы изменить описание атрибута h класса Controller:

```
Heater * h;
```

В этом случае класс Controller по-прежнему означает целое, но его часть, экземпляр класса Heater, содержится в целом косвенно. Теперь объекты живут отдельно друг от друга: мы можем создавать и уничтожать экземпляры классов независимо.

Агрегация является направленным отношением. Объект Heater входит в объект Controller, а не наоборот. Физическое вхождение одного в другое нельзя «зациклить», а указатели – можно (каждый из двух объектов может содержать указатель на другой).

## 4.3. ЗАВИСИМОСТЬ

**Пример.** Пусть управление температурой каждый объект класса Controller осуществляет в соответствии с задаваемым ему планом. План представим в виде экземпляра класса Plan.

```
class Plan;
class Controller{
        . . .
        void process(Plan& );
        . . .
};
```

Класс Plan упомянут как часть описания функции-члена process; это дает нам основание сказать, что класс Controller пользуется услугами класса Plan.

Отношение **зависимости (использования)** между классами означает, что изменение в спецификации одного класса может повлиять на другой класс, который его использует, причем обратное в общем случае неверно. Можно сказать, что один из классов (клиент) пользуется услугами другого (сервера).

Один класс может использовать другой по-разному. В нашем примере это происходит при описании интерфейсной функции. Отношение использования также имеет место, если в реализации какой-либо операции происходит объявление локального объекта используемого класса.

## 4.4. НАСЛЕДОВАНИЕ

### 4.4.1. НАСЛЕДСТВЕННАЯ ИЕРАРХИЯ

**Наследование** – это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других классов (множественное наследование).

**Пример.** Рассмотрим определение типа Shape (фигура) для использования в графической системе. Предположим, у нас есть два класса:

```
class Point { ... }; //Точка
class Color { ... }; //Цвет
```

Мы можем определить Shape следующим образом:

```
enum Kind { circle, triangle, square };   // перечисление: окружность,
                                          // треугольник, квадрат
class Shape {
        Kind k,           //поле типа (какая фигура?)
        Point center,  // центр фигуры
        Color col,      //цвет фигуры
public:
        void move(Point to); // переместить
        void draw( );            // нарисовать
        void rotate(int);         // повернуть
        Point isCenter( );// возвращает значение центра фигуры
};
```

«Поле типа» k необходимо, чтобы такие операции, как draw и rotate, могли определить, с каким видом фигуры они имеют дело. Функцию draw( ) можно определить следующим образом:

```
void Shape :: draw( ) {
        switch(k) {
                case circle: . . . break; // нарисовать окружность
                case triangle: . . . break; // нарисовать треугольник
                case square: . . . // нарисовать квадрат
        }
}
```

Таким образом, функции должны «знать» обо всех возможных видах фигур. Поэтому код любой такой функции растет с добавлением новой фигуры в систему. Если мы определили новую фигуру, каждую операцию над фигурой нужно просмотреть и, вероятно, модифицировать. У нас есть возможность добавить новую фигуру к системе, только если мы имеем исходные тексты каждой функции. Так как добавление новой фигуры связано с внесением изменений в код каждой важной операции над фигурами, оно требует большого мастерства и потенциально влечет появление ошибок в коде, управляющем другими (старыми) фигурами.

Рассмотрим решение, использующее механизм наследования. Разделим *общие свойства* всех фигур (все они имеют цвет, их можно нарисовать и т. д.) и *особые свойства* фигур определенного типа (у окружности есть радиус, она вычерчивается специфическим способом и т. д.).

Сначала мы описываем класс, который определяет общие свойства всех фигур:

```
class Shape {
        Point center;
        Color col;
        . . .
public:
        void move(Point to) { center =to; ... draw ( ); }
        virtual void draw( ) = 0;
        virtual void rotate(int angle)=0;
        . . .
};
```

Описание virtual означает, что функция является **виртуальной** – замещается в классе, производном от данного.

Функция, интерфейс вызова которой может быть определен, а реализация – нет, объявляется **чисто виртуальной**, для чего используется синтаксис «= 0». Например, реализации функций draw и rotate могут быть определены только для конкретных фигур, поэтому эти функции вообще *не реализованы* в классе Shape.

Таким образом, виртуальные (и чисто виртуальные) функции являются полиморфными. Особенности полиморфизма, связанного с виртуальными функциями, будут рассмотрены в разд. 4.4.2.

Для описания полиморфизма используются два различных понятия: операция и метод. У класса есть **операции**, которые определяют его поведение, и **методы** – реализации данных операций. При этом каждый потомок класса может предоставить метод, реализующий любую унаследованную операцию, отличный от соответствующего метода предка. Чисто виртуальная функция (абстрактная операция) не имеет соответствующего метода.

Для определения конкретной фигуры мы должны сказать, что она является фигурой, и указать особые свойства (в том числе определить чисто виртуальные функции):

```
class Circle: public Shape {
        int radius;
public:
        Circle(Point cntr, Color cl, int rds) {. . .};
        void draw( ){. . .};
        void rotate(int) {};//функция ничего не делает
};
```

Классы Circle и Shape называются подклассом (потомком) и суперклассом (надклассом, родительским классом) соответственно. В C++ подкласс, как правило, называют производным классом, а суперкласс – базовым классом.

Класс Circle создан для того, чтобы оперировать с его объектами. Классы, для которых создаются экземпляры, называют **конкретными.** С другой стороны, в классе Shape описаны чисто виртуальные функции, не имеющие реализации по определению, и создание экземпляров данного класса запрещено языком C++. Классы, экземпляры которых не создаются, называются **абстрактными**. Ожидается, что подклассы абстрактных классов доопределят их до жизнеспособной абстракции, наполняя класс содержанием. Классы, у которых нет потомков, называют **листовыми.**

Самый общий класс в иерархии классов называется **корневым.** В большинстве приложений корневых классов бывает несколько, и они отражают наиболее общие абстракции предметной области.

У класса обычно бывает два вида клиентов: собственные подклассы и экземпляры классов. Часто полезно иметь для них разные интерфейсы. В частности, мы хотим показать только внешне видимое пове-

дение для клиентов-экземпляров, но нам нужно открыть служебные функции и атрибуты клиентам-подклассам.

Для этой цели описание класса разделяется на три части:

- открытую (public), видимую всем клиентам;
- защищенную (protected), видимую самому классу, его подклассам и друзьям (friend);
- закрытую (private), видимую только самому классу и его друзьям.

Таким образом, язык C++ позволяет достаточно гибко найти компромисс между наследованием и инкапсуляцией.

Наследование подразумевает, что подклассы повторяют и могут дополнять структуры их суперклассов. В нашем примере класс Circle содержит элементы структуры суперкласса Shape и более специализированный элемент – радиус.

Поведение суперклассов также наследуется. Например, с помощью операции move класса Shape можно переместить экземпляр класса Circle. При этом в производном классе допускается добавление новых и переопределение существующих методов.

Например, опишем класс круг – SolidCircle – закрашенная окружность:

```
class SolidCircle: public Circle {
protected:   Color fillcol; // цвет заполнения
public:      SolidCircle(Point cntr, Color cl, int rds, Color fcl):
                    Circle(Point cntr, Color cl, int rds), fillcol (fcl) {. . .};
      void draw( ){ Circle :: draw ( ); . . . };
};
```

Функция draw сначала вызывает аналогичную функцию родительского класса, которая рисует границу круга, а затем сама заполняет ее цветом.

В большинстве объектно-ориентированных языков программирования при реализации метода подкласса разрешается вызывать напрямую метод какого-либо суперкласса. Как видно из примера, это допускается и в том случае, если метод подкласса имеет такое же имя и фактически переопределяет метод суперкласса. В C++ для этого имя суперкласса добавляется в качестве префикса, тем самым формируется квалифицированное имя метода.

Поскольку конструкторы не наследуются, производный класс должен иметь собственные конструкторы. При создании объекта в случае иерархии, состоящей из нескольких уровней, сначала вызываются конструкторы суперклассов (начиная с самого верхнего уровня в порядке объявления). После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса.

Если конструктор суперкласса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации (см. описание конструктора класса SolidCircle). Если в конструкторе производного класса явный вызов конструктора суперкласса отсутствует, автоматически вызывается конструктор суперкласса по умолчанию.

Не наследуется и операция присваивания, поэтому ее также требуется явно определить в классе.

Деструкторы также не наследуются, и если программист не описал в производном классе деструктор, он формируется по умолчанию и вызывает деструкторы всех суперклассов. В отличие от конструкторов, при написании деструктора производного класса в нем не требуется явно вызывать деструкторы суперклассов, поскольку это будет сделано автоматически. Для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов.

### 4.4.2. НАСЛЕДОВАНИЕ И ТИПИЗАЦИЯ

Вопросы построения наследственных иерархий тесно связаны с типизацией (в языках с сильной типизацией), поскольку при использовании наследования формируется и система типов.

При определении класса его суперкласс можно объявить public (как в нашем примере). В этом случае открытые и защищенные члены суперкласса становятся открытыми и защищенными членами подкласса. Таким образом, подкласс считается также и подтипом, т. е. должен выполнять все обязательства суперкласса. В частности, он обеспечивает совместимое с суперклассом подмножество интерфейса и обладает неразличимым с точки зрения клиентов суперкласса поведением. Именно в этом случае можно говорить об иерархии «является».

Если при определении класса объявить его суперкласс как private, это будет означать, что, наследуя структуру и поведение суперкласса, подкласс уже не будет его подтипом. Открытые и защищенные члены суперкласса станут закрытыми членами подкласса, и, следовательно, они будут недоступны подклассам более низкого уровня. В этом случае подкласс и суперкласс обладают несовместимыми интерфейсами с точки зрения клиента.

Если объявить суперкласс protected, то открытые и защищенные элементы такого суперкласса станут защищенными элементами подкласса. Однако с точки зрения клиента интерфейсы класса и суперкласса несовместимы.

**Пример.** Продолжим рассмотрение наследственной иерархии, связанной с фигурами в графической системе. Сделаем следующие объявления:

```
Circle C1;
SolidCircle SC1, SC2;
```

Присвоение объекту A значения объекта B в языке C++ допустимо, если тип объекта B совпадает с типом объекта A или является его подтипом.

Поскольку SolidCircle является открытым подклассом Circle, следующий оператор присваивания правомочен:

```
C1 = SC1;
```

Хотя он формально и правилен, но опасен: любые дополнения в состоянии подкласса по сравнению с состоянием суперкласса срезаются. Таким образом, дополнительный атрибут fillcol, определенный в подклассе SolidCircle, будет потерян при копировании, поскольку его просто некуда записать в объекте класса Circle.

Следующий оператор недопустим:

```
SC2 = C1; // ошибка; атрибут fillcol отсутствует у C1
```

Изменим описание класса SolidCircle на следующее:

```
class SolidCircle: Circle{. . .};
```

Теперь суперкласс Circle по умолчанию объявлен закрытым. Поскольку класс SolidCircle не является теперь подтипом Circle, мы уже не сможем присваивать экземпляры подкласса объектам суперкласса, как в случае объявления суперкласса в качестве открытого.

```
C1 = SC1; // теперь нельзя
```

Отметим, что унаследованной функции можно назначить тот же, что и в суперклассе, атрибут доступа в подклассе путем явной квалификации.

```
class SolidCircle: Circle{
public:
        . . .
        Circle:: move;
};
```

Правила C++ запрещают делать унаследованный элемент в подклассе «более открытым», чем в суперклассе. Например, член, объявленный в суперклассе защищенным, не может быть сделан в подклассе открытым посредством явного упоминания.

С наследованием связан особый тип полиморфизма – **включение (чистый полиморфизм).** Данный тип полиморфизма реализуется при вызове виртуальных функций для указателей (ссылок) на объекты. При открытом наследовании указатель родительского класса может указывать на объекты всех подклассов. Если виртуальная функция имеет различные реализации в подклассах, то выбор, какую ее реализацию вызывать, определяется с учетом выяснения подтипа на этапе выполнения, т. е. виртуальная функция вызывается в зависимости не от *типа указателя*, а от реального *типа объекта*, на который он указывает. Данная ситуация называется **механизмом позднего связывания.**

Чистый полиморфизм позволяет взаимодействовать с объектом, не зная, к какому конкретному классу он относится. Это происходит за счет общего интерфейса классов в открытой иерархии наследования.

**Пример.** Опишем наследственную иерархию

```
class One { public:   virtual f( ){return 1;} };
class Two : public One { public: virtual f( ){return 2;} };
```

Рассмотрим следующий фрагмент кода:

```
One one, *p;
Two two;
p = &one; p -> f( ); // p указывает на объект типа One, f ( ) возвратит 1
p = &two; p -> f( ); // p указывает на объект типа Two, f ( ) возвратит 2
one.f ( ); // f( ) возвратит 1
one = two; one.f( ); // one – объект типа One, f ( ) возвратит 1
```

С другой стороны, если бы функция f( ) не была виртуальной, то ее описание в обоих классах было бы не включением, а перегрузкой, и во всех случаях возвращалось бы значение 1.

Функция, объявленная виртуальной в суперклассе, подчиняется правилам включения в подклассе, даже если ключевое слово virtual не указано, при условии идентичности сигнатур (и типов возвращаемых значений). Иначе вместо включения имеет место перегрузка.

Для безопасного выявления типа объекта, на который направлен во время работы программы указатель родительского класса или ссылка на него, служит механизм **динамического определения типа**, специально введенный в язык C++.

Одним из инструментов данного механизма является оператор приведения dynamic_cast. Рассмотрим случай с указателем.

```
dynamic_cast < Type* > (p);
```

Рассмотрим приведение потомка к типу родителя, которое называется **повышающим приведением**. Если p имеет тип Type* или является указателем на открытый производный класс для Type, то результат будет точно такой же, как при простом присваивании p указателю типа Type*. В противном случае возвращается нуль.

**Пример.** Рассмотрим следующий фрагмент кода (напомним, что Circle теперь является закрытым родительским классом для SolidCircle):

```
Shape *S; Circle *C; SolidCircle *SC;
. . .
S = C; // правильно
S = dynamic_cast < Shape* > (C); // эквивалентно предыдущему
S = dynamic_cast < Shape* > (SC); // правильно, возвратится 0
S = SC; // ошибка
```

Приведение родителя к типу потомка называется **понижающим приведением**. Оно применяется только к классам, имеющим виртуальные функции, когда действует механизм позднего связывания. Если p указывает на объект типа Type или его подтипа, то оператор возвращает указатель на объект, иначе возвращается нуль.

**Пример.** Определим функцию, выясняющую, имеет ли объект, указатель на который служит ее аргументом, тип Circle или его подтип.

```
bool isCircle(Shape* ptr){
        Circle* cptr = dynamic_cast < Circle*> (ptr);
        return cptr != 0;
}
```

Использование функции иллюстрирует следующий фрагмент:

```
Circle C; Triangle T; SolidCircle SC;
. . .
isCircle(&C); // true
isCircle(&T); // false
isCircle(&SC); // true или false, в зависимости от того, является
                // наследование для SolidCircle открытым или закрытым
```

Другой инструмент механизма динамического определения типа – **класс type_info**, служащий для представления информации о типе. Он определен в заголовочном файле typeinfo.h. Объекты данного класса можно сравнивать на равенство и неравенство, а операция name выдает имя типа. Получить объект данного класса, хранящий информацию о типе нужного объекта, можно с помощью операции typeid.

```
Circle C; Shape* PS = &C;
strcmp(typeid(PC).name( ), "Circle"); // true
```

Если суперкласс содержит хотя бы один виртуальный метод, то рекомендуется всегда снабжать этот класс виртуальным деструктором, даже если он ничего не делает. Наличие такого виртуального деструктора предотвратит некорректное удаление объектов производного класса, адресуемых через указатель на суперкласс, так как в противном случае деструктор производного класса вызван не будет.

**Задача.** Укажите ошибочные строки в функции main среди отмеченных буквами А, Б, В, Г, Д, Е.

```
class Shape {. . .}S;
class Circle: public Shape {. . .}C;
class SolidCircle: Circle {. . .}SC;
class Square: Shape {. . .}SQ;
void main( ) {
        S=C; // A
        C=S; // Б
        C=SC; // В
        SC=C; // Г
        S=SQ; // Д
        SC=SQ; // Е
}
```

**Задача.** Что напечатает программа?

```
class Base { public:
          void virtual info1( ){printf("Base1\n");}
          void info2( ){printf("Base2\n");}
};
class Derived : public Base { public:
          void virtual info1( ){printf("Derived1\n");}
          void info2( ){printf("Derived2\n");}
};
void main( ) {
          Base B, *PB; Derived D;
          PB = &B; PB->info1( );
          PB = &D; PB->info1( ); PB->info2( );
          B=D; B.info1( );
}
```

### 4.4.3. МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

Рассмотрим две проблемы, которые возникают при множественном наследовании: конфликт имен между суперклассами и повторное наследование.

**Конфликт имен** происходит тогда, когда в двух или более суперклассах случайно оказывается элемент (переменная или операция) с одинаковым именем.

**Пример.** Определим абстракцию «Работающий студент». Для этого введем более общие абстракции «Работник» и «Студент». Абстракция «Работающий студент» будет наследовать компоненты обеих общих абстракций.

```
class Worker {
public:
        int     ID_profession; // код профессии
        char*   Name; // имя
};
class Student {
public:
        int     ID_university; // код университета
        char*   Name; // имя
};
class Student_Worker: public Student, public Worker { . . . };
```

Рассмотрим последовательность действий.

```
Student_Worker He;
. . .
He.ID_profession; // правильно
He.Name; // неправильно – двусмысленно
```

Конфликт имен элементов подкласса может быть разрешен полной квалификацией имени члена класса, т. е. к именам добавляют префиксы, которые указывают имена тех классов, откуда они пришли.

```
He.Worker :: Name; // правильно
```

**Повторное наследование** возникает тогда, когда в многоуровневой наследственной иерархии какой-либо класс дважды является суперклассом для другого класса.

Продолжим пример с работающим студентом. Анализируя глубже полученную иерархию наследования, мы обнаружим, что и работник,

и студент имеют ряд общих признаков, в частности имя. Разумно ввести еще более общую абстракцию «Человек».

```
class Person {
public:      char* Name; // имя
}

class Worker : public Person {
public:      int ID_profession; // код профессии
}

class Student : public Person {
public:      int ID_university; // код университета
}
```

Наследственная иерархия класса Student_Worker представлена на рис. 4.1.

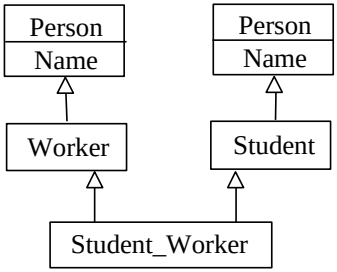


*Рис. 4.1.* Наследственная иерархия класса Student_Worker

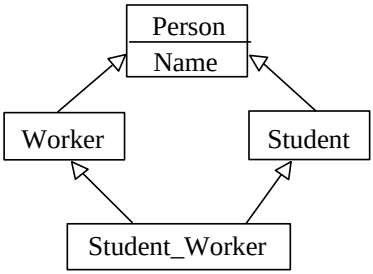


*Рис. 4.2.* Ромбовидная структура наследования

Для доступа к одной из копий унаследованного элемента необходимо воспользоваться явной квалификацией, т.е. добавить к его имени префикс в виде имени класса-источника.

```
He.ID_profession; // правильно
He.Name; // неправильно – двусмысленно
He.Person :: Name; // неправильно – двусмысленно
He.Worker :: Name; // правильно
He.Student :: Name; // правильно
```

Продолжая анализ полученной иерархии, заметим, что работающий студент имеет всего одно имя. В результате объект класса Student_Worker должен использовать единственную копию элемента Name, унаследованную от Person. В результате приходим к **ромбовидной структуре наследования**, когда повторяющийся суперкласс в производном классе представлен одним и тем же (совместно используемым) объектом (см. рис. 4.2).

В C++ механизмом задания ромбовидной структуры наследования является **виртуальное наследование**, когда повторяющийся суперкласс объявляется «виртуальным базовым классом». Для задания виртуального наследования используется синтаксис следующего примера.

```
class Person { . . .};
class Worker : public virtual Person {. . .};
class Student : public virtual Person {. . .};
class Student_Worker: public Student, public Worker {. . . };
```

**Задача.** Укажите ошибочные строки в функции main среди отмеченных буквами А, Б, В, Г, Д, Е.

```
class Transport // Транспортное средство
{public: String Registration_Number; // регистрационный номер
};
class Land_Transport: public Transport { // Сухопутное транспортное средство
public:   int Shaft; // ведущая ось
};
class Water_Transport: public Transport { // Водное транспортное средство
public:   int Displacement; // водоизмещение
};
class Amphibia: public Land_Transport, public Water_Transport {}; // Амфибия
      void main( ) {
            Amphibia amph;
            . . .
            amph.Shaft; // А
            amph.Displacement; // Б
            amph.Water_Transport :: Displacement; // В
```

```
        amph.Registration_Number; // Г
        amph.Water_Transport :: Registration_Number; // Д
        amph.Transport :: Registration_Number; // Е
}
```

## 4.5. ИНСТАНЦИРОВАНИЕ

**Пример.** Представим, что нам необходимы стек целых чисел и стек контроллеров, управляющих температурой. Мы могли бы описать два стека:

```
class IntStack {
        int stack[100];
        . . .
};
class ControllerStack {
        Controller* stack[100];
        . . .
};
```

Другой, более разумный, подход – создать универсальный стек, который мог бы хранить элементы любого нужного нам типа. Для этого мы можем описать стек, содержащий указатели на нетипизированные элементы:

```
class Stack {
        void* stack[100];
        . . .
};
```

Однако это небезопасно с точки зрения типов. Никто не гарантирует нам, что пользователь не поместит в стек элемент одного типа, а взять захочет элемент другого типа.

Для реализации нашей идеи необходимо воспользоваться **шаблоном** или **параметризованным классом**. Шаблон служит для построения других классов и может быть параметризован другими классами, объектами или операциями. Использование шаблонов реализует в язы-

ке C++ особый тип полиморфизма – **параметрический полиморфизм.**

```
template <class Type> class Stack {
        Type stack[100];
        . . .
public:
        void push(Type);
        Type pop( );
        . . .
};
. . .
template <class Type> void Stack <Type> :: push(Type el) { . . .}
template <class Type> Type Stack <Type> :: pop( ) { . . .}
```

Префикс template <class Type> делает Type параметром объявления, которому этот префикс предшествует.

**Инстанцирование** – подстановка фактических параметров шаблона вместо формальных. В результате создается конкретный класс, который может иметь экземпляры.

Объявим нужные нам стеки:

```
typedef Stack < int > IntStack // синоним класса стеков целых чисел
typedef Stack < Controller* > ControllerStack // синоним класса стеков
                                                // контроллеров
IntStack IS; // стек для целых чисел
ControllerStack CS; // стек для контроллеров
```

Объекты IS и CS – это экземпляры совершенно различных классов, которые даже не имеют общего суперкласса. Тем не менее они получены из одного параметризованного класса Stack.

Инстанцирование безопасно с точки зрения типов. По правилам C++ будет отвергнута любая попытка поместить в стек или извлечь из него что-либо, кроме целых чисел или указателей на экземпляры класса Controller соответственно.

В языке C++ можно определять шаблоны не только классов, но и функций. В качестве примера рассмотрим определение шаблона функции, служащей для определения максимального из двух элементов.

```
template <class Type > Type max(Type x, Type y){
        return (x > y) ? x : y;
};
```

Теперь мы можем использовать один и тот же шаблон для целых и вещественных чисел.

```
int i = 1, j = 2, k;
double a = 1.5, b = 1.2, c, d;
k = max(i, j); c = max(a, b); d = max <double> (i, a);
```

Кроме того, можно использовать этот шаблон и для объектов некоторого класса, если в нем переопределен оператор «>».

Каждая версия класса или функции, создаваемая по шаблону, содержит одинаковый базовый код; изменяется только то, что связано с параметрами шаблона. При этом эффективность работы версий, создаваемых для различных типов данных, может сильно различаться. Если для какого-либо типа данных существует более эффективный код, можно либо предусмотреть для этого типа специальную реализацию отдельных операций, либо полностью переопределить (специализировать) шаблон класса.

Так, для специализации операции требуется определить вариант ее кода, указав в заголовке конкретный тип данных.

```
void Stack < Circle*> :: push(Circle* el) { . . .}
```

При специализации целого класса после описания обобщенного варианта класса помещается полное описание специализированного класса, при этом требуется заново определить все его методы:

```
class Stack < Circle*> {. . .}
```

## 4.6. СТАТИЧЕСКИЕ ЭЛЕМЕНТЫ КЛАССА

Обычно для каждого объекта необходима своя копия переменных, описанных в классе. Однако в некоторых ситуациях требуется, чтобы в классе были данные, общие для всех его экземпляров – переменные класса.

Переменная класса (статическая переменная) в C++ описывается с ключевым словом static, она создается один раз как часть класса, а не для каждого конкретного экземпляра данного класса. Функция, которой требуется доступ к статическим переменным, но не требуется, чтобы она вызывалась для конкретного экземпляра класса, также описывается как статическая (static).

Обращаться к статическим элементам можно, как к обычным элементам, через любой объект, а также без указания объекта с использованием квалифицированного имени.

Статические переменные должны быть дополнительно описаны и инициализированы вне определения класса как глобальные переменные (даже если первоначально они описаны в закрытой или защищенной части класса), после инициализации к ним можно обращаться даже до определения объектов данного класса.

При наследовании статические атрибуты наследуются, но это атрибуты суперкласса, единые для всех подклассов.

**Пример.** Используем статическую переменную для подсчета объектов-фигур, потомков класса Shape.

```
class Shape {
        static int count; // счетчик фигур – статическая переменная
          . . .
public:
        Shape( ) {count++;}
        ~ Shape( ) {count--;}
        static int get_count( ) {return count;} // статическая get-функция
        . . .
};
int Shape :: count = 0; // описание и инициализация вне класса
. . .
int c; Circle C; Triangle T; SolidCircle SC;
c = C.get_count( ); // три эквивалентных
c = Shape :: get_count( ); // обращения к
c = Circle :: get_count( ); // статической функции
```

Хотя класс Shape и является абстрактным, мы описали в нем конструктор и деструктор, цель которых состоит лишь в изменении статической переменной – счетчика. При объявлении объекта подкласса

будет вызван конструктор класса Shape, а при уничтожении – деструктор. По этой причине изменять счетчик в конструкторе и деструкторе производных классов не нужно.

Статические члены используются также для реализации на языке C++ утилит. **Утилитами** называют совокупность глобальных переменных и свободных подпрограмм, сгруппированных в форме объявления класса. В этом случае глобальные переменные и свободные подпрограммы рассматриваются как члены класса, причем именно как статические. Введение утилит позволяет приблизить реализацию системы на языке C++ к набору классов и взаимодействующих объектов, как в чисто объектно-ориентированных языках.

**Задача.** Что напечатает программа?

```
class Point {public: static int count;. . .};
int Point :: count=0;
void main( ) {
        Point P1, P2;
        P1.count =1; P2.count =2; printf("%d\n", P2.count);
        P1.count++; printf("%d\n", P1.count);
}
```

## 4.7. ИНТЕРФЕЙСЫ

Когда с помощью объектно-ориентированного подхода начали разрабатывать крупные программные системы, выяснилось, что кроме классов нужны дополнительные уровни абстракции. В частности, если сложный объект имеет разное поведение (играет разные роли), в зависимости от того, с кем он взаимодействует, то бывает удобно скрыть все функции, не нужные в данный момент. А точнее: на время данного взаимодействия сделать доступными все необходимые функции и только их. Для описания таких групп функций удобно использовать понятие **интерфейса.** В данном контексте интерфейс удобно рассматривать как абстрактный класс, содержащий только чисто виртуальные функции и не имеющий собственных данных.

**Пример.** Все элементы управления телевизором можно разделить на несколько групп: пользовательские (громкость, номер канала), специальные (частота канала) и аппаратные (параметры электрических цепей). При этом пользователь работает с пользовательскими органа-

ми управления, настройщик – со специальными, а телемастер – с аппаратными. При этом, если телевизор исправен и настроен, пользователю нет необходимости видеть и менять состояние специальных и аппаратных органов управления. Поэтому пользовательские элементы управления обычно выносятся на переднюю панель телевизора, специальные закрыты небольшой дверцей, а аппаратные вообще погружены внутрь корпуса. Если бы все было на поверхности, пользователь мог бы сделать все то же, что и раньше, но для него оказались бы доступными специальные и аппаратные органы управления и он мог бы случайно испортить настройки. Кроме того, передняя панель была бы загромождена настолько, что мало кто смог бы ориентироваться в обилии кнопок, ручек и т. п.

Между интерфейсами могут существовать отношения наследования, ассоциации и зависимости, аналогичные одноименным отношениям между классами. Между классом и интерфейсом могут существовать отношения реализации и зависимости. Будем говорить, что класс **реализует** (или **поддерживает**) интерфейс, если он содержит методы, реализующие все операции интерфейса. Интерфейс может реализовываться несколькими классами (например, пользовательские элементы радиоприемника и телевизора совпадают), а класс может реализовывать несколько интерфейсов (например, телевизор реализует три интерфейса). С другой стороны, класс может зависеть от нескольких интерфейсов, при этом предполагается, что какие-то классы эти интерфейсы реализуют.

```
# define interface struct  // функции – элементы интерфейса – открыты
interface IUser{ // пользовательский интерфейс
        virtual void change_channel(int)=0;
        virtual void change_sound(int)=0;
};
class Televisor: public IUser, public ISpecial, public IApparatus {
// класс Televisor реализует пользовательский, специальный и
// аппаратный интерфейсы
        int channel, sound_level;
        . . .
public:
        virtual void change_channel(int number) {channel = number;}
        virtual void change_sound(int rel) {sound += rel;}
```

```
    . . .
};
class User { // класс User зависит от интерфейса IUser
public: User(IUser *IU);
. . .
};
. . .
Televisor* My_TV; . . .
User My(My_TV);
```

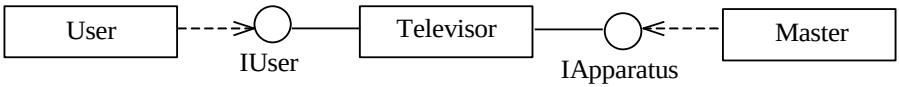


*Рис. 4.3.* Интерфейсы

На рис. 4.3 показано, что пользователь взаимодействует с телевизором посредством интерфейса IUser, а телемастер – посредством интерфейса IApparatus.

## 4.8. КАТЕГОРИИ КЛАССОВ

Когда система разрастается до десятка классов, можно заметить группы классов, связанные внутри и слабо зацепляющиеся с другими. Такие группы классов образуют категории.

**Категория классов** – это агрегат, состоящий из классов и других категорий классов.

Категория классов не имеет операций или состояний в явном виде, они содержатся в ней неявно в описаниях агрегированных классов.

Некоторые классы в категории могут быть открытыми, т. е. экспортироваться для использования за пределы категории. Остальные классы могут быть частью реализации, т. е. не использоваться никакими классами, внешними к этой категории.

Между категориями и классами может существовать отношение использования. Категория может использовать невложенные категории и классы. С другой стороны, и классы могут использовать категории.

В C++ категории классов реализуются с помощью введения **пространств имен –** namespace. Однако пространство имен позволяет реализовать более широкое понятие. Оно может включать в себя классы, другие пространства имен, свободные подпрограммы и глобальные (внутри пространства имен) данные.

Заметим, что пространства имен связаны с идеей модульности, выполняя функцию логического группирования элементов (в отличие от модулей, осуществляющих в первую очередь физическое группирование элементов).

**Пример.** Объединим все классы, разработанные для использования в графической системе, в одну категорию. Предоставим пользователю описания в файле GraphSys.h:

```
namespace GraphSys{
        class Point{ ... };
        class Color{ ... };
        class Shape {... };
        class Circle: public Shape{... };
        class SolidCircle: public Circle {... };
        . . .
}
```

Реализация указанных классов находится в файле GraphSys.cpp:

```
# include "GraphSys.h"
namespace GraphSys{
        Circle :: draw( ){...}
        SolidCircle :: draw( ){...}
        . . .
}
```

Обращение к членам пространства имен осуществляется с использованием явной квалификации:

```
GraphSys :: Circle C;
GraphSys :: SolidCircle SC;
```

С другой стороны, описание using в пользовательском коде позволяет не использовать все время явную квалификацию:

```
# include "GraphSys.h"
using namespace GraphSys;
void user_func( ){
        Circle C;
        . . .
        C.draw( );
}
```

# 5. ОСНОВНЫЕ КОНСТРУКЦИИ ЯЗЫКА UML

Поскольку использование объектно-ориентированного подхода особенно важно при разработке сложных программных продуктов, модели предметной области, которые приходится строить в этих случаях, тоже будут сложны. Поэтому большое значение при объектно-ориентированном подходе имеют средства, позволяющие визуализировать, сохранять и документировать принимаемые решения. Одним из таких средств является унифицированный язык моделирования UML. Разработка системы средствами UML выполняется в виде построения набора диаграмм, позволяющих описать определенные части моделируемой реальности в определенных аспектах. В данном разделе мы рассмотрим основные элементы ряда диаграмм UML.

## 5.1. ДИАГРАММА КЛАССОВ

Основными элементами, отображаемыми на **диаграмме классов**, являются классы и отношения между ними.

Графическое изображение класса представлено на рис. 5.1.

Каждый класс должен иметь имя. На некоторых значках классов полезно перечислять несколько атрибутов и операций класса. «На некоторых», потому что для большинства тривиальных классов это не нужно. Если мы не хотим видеть на диаграмме атрибуты и операции класса, мы удаляем разделяющие черты и пишем только имя класса.


*Рис. 5.1.* Значок класса

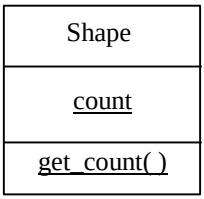


*Рис. 5.2.* Статические члены


*Рис. 5.3.* Утилита класса

Атрибут может обозначаться именем, могут быть также указаны дополнительные элементы в соответствии со следующим синтаксисом:

видимость имя : тип [кратность] = значение_по_умолчанию {строка свойств}

Видимость обозначается посредством добавления в качестве префикса к имени следующих символов: + (открытый), # (защищенный), – (закрытый).

Кратность (количество элементов указанного типа, составляющих атрибут) показывается в виде последовательности разделенных запятой спецификаций интервалов в формате: «нижняя граница .. верхняя граница» и конкретных значений. Для обозначения неограниченного количества используется символ *. Например:

| | |
|---|---|
| 1 | в точности один; |
| * | ноль или больше; |
| 0 .. * | ноль или больше; |
| 1 .. * | один или больше; |
| 0 .. 1 | ноль или один; |
| 3 .. 7 | указанный интервал. |

Приведем примеры описания атрибутов:

| | |
|---|---|
| A | только имя; |
| – A : C [10] | видимость, имя, тип и кратность; |
| – A : C = E | видимость, имя, тип и значение по умолчанию. |

Строка свойств, указанная в фигурных скобках, описывает дополнительные свойства атрибута (примеры свойств приведены ниже).

Операции обычно изображаются внутри значка класса только своим именем. Можно также указывать дополнительные элементы в формате:

видимость имя (формальные_параметры)
: тип_возвращаемого_значения {строка свойств}

Формальные параметры записываются в формате:

вид имя : тип = значение_по_умолчанию

Вид принимает одно из следующих значений:

in – входной параметр (значение по умолчанию);

out – выходной параметр;

inout – смешанный, т. е. и входной, и выходной.

Если параметр помечен как входной, то функция не может изменять его значения, но может его использовать (передача параметров по значению); если параметр помечен как выходной, то функция не может использовать его значение, но может присвоить ему некоторое значение; наконец, если параметр помечен как смешанный, то функция может и читать его значение, и изменять (передача параметров по ссылке).

Приведем примеры описания операций:

display( ) только имя;

display( ): bool имя и тип возвращаемого значения;

+set(n: Name, s: String) видимость, имя и формальные параметры.

Статические члены класса (атрибуты и операции) изображаются подчеркнутыми (см. рис. 5.2, соответствующий примеру в разд. 4.6).

Утилита изображается обычным значком класса со стереотипом «utility» (рис. 5.3). Добавление к стандартному элементу языка **стереотипа** (некоторого ключевого слова в скобках « ») позволяет приспособить данный элемент для решения специальной проблемы.

Интерфейс на диаграмме классов можно изобразить развернутым и сокращенным. В случае развернутого способа интерфейс изображается на диаграмме как класс со стереотипом «interface» и без секции атрибутов (рис. 5.4). Сокращенное изображение интерфейса представляет небольшой кружок с именем интерфейса возле него (рис. 5.5).


*Рис. 5.4.* Развернутое изображение интерфейса


*Рис. 5.5.* Сокращенное изображение интерфейса

Рассмотрим обозначения отношений между классами.

Значок ассоциации изображается в виде линии, соединяющей два класса (рис. 5.6). При изображении ассоциации ей можно поставить в соответствие текстовую пометку, документирующую имя этой связи или подсказывающую ее роль. Ассоциации часто отмечаются существительными, например «Место работы», описывающими природу связи. К полюсам ассоциации (концам линии) также можно добавить имена (роли, которые классы играют в ассоциации). Одна пара классов может иметь более одной ассоциативной связи.

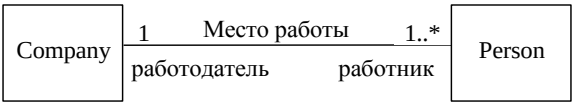


*Рис. 5.6.* Ассоциация

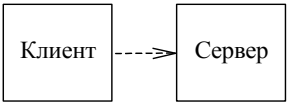


*Рис. 5.7.* Зависимость

Возле значка ассоциации можно указать ее кратность. Приведя кратность возле одного полюса, показывают, что на этом полюсе именно столько объектов должно соответствовать каждому объекту на другом полюсе. Если кратность явно не указана, то подразумевается, что она не определена.

На рис. 5.6 изображена ассоциация из примера в разд. 4.1 .

Отношение зависимости изображается пунктирной линией со стрелкой, направленной от клиента к серверу (рис. 5.7).

Значок наследования (обобщения) включает большую незакрашенную стрелку, которая указывает на суперкласс (рис. 5.8). Место классов в наследственной иерархии можно уточнить следующим образом. Имена абстрактных классов указываются курсивом. К имени корневого класса добавляется свойство root, а к имени листового класса – свойство leaf. Аналогично имена чисто виртуальных функций-членов указываются курсивом, а к имени операции, которая не может быть виртуальной, добавляется свойство leaf.

**Пример.** На рис. 5.8 показана наследственная иерархия, связанная с геометрическими фигурами (см. разд. 4.4.1). Абстрактный класс Shape отмечен как корневой класс, имеющий чисто виртуальные функции draw, rotate и листовую функцию get_center. Функция draw в протоколе класса Circle является полиморфной. Класс SolidCircle объявлен как листовой.

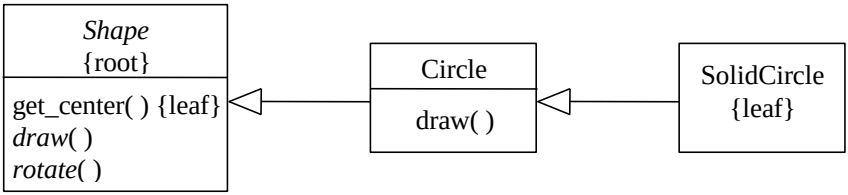


*Рис. 5.8.* Обобщение

Значок агрегации – линия с добавлением незакрашенного ромба на конце, обозначающем агрегат (рис. 5.9). Экземпляры класса на другом конце стрелки будут частями экземпляров класса-агрегата. Отношение композиции обозначается линией с закрашенным ромбом на конце (рис. 5.10).

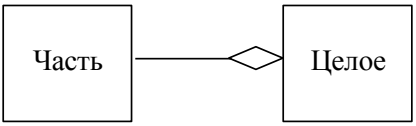


*Рис. 5.9.* Агрегация

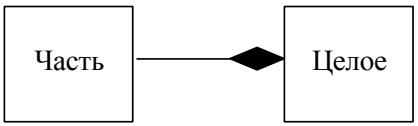


*Рис. 5.10.* Композиция

Параметризованный класс изображается значком обычного класса с пунктирным прямоугольником в правом верхнем углу, в котором указаны параметры (рис. 5.11). Моделировать инстанцирование можно двумя способами. Во-первых, можно явным образом определить отношение **реализации** (изображается в виде пунктирной линии с большой незакрашенной стрелкой) со стереотипом «bind», показывающее, что источник инстанцирует целевой шаблон. Фактические параметры, соответствующие формальным параметрам шаблона, указываются в виде

<Формальный параметр1 –> Фактический параметр1, . . .>

Во-вторых, моделировать можно неявно, для чего требуется объявить класс, имя которого включает имя шаблона и фактические параметры, указанные таким же способом, что и в первом случае.

Пример использования данных способов показан на рис. 5.11, 5.12 для стека целых значений, рассмотренного в разд. 4.5.

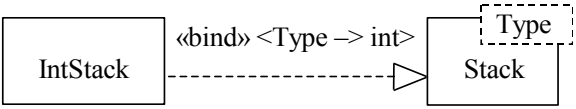


*Рис. 5.11.* Явный способ

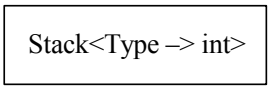


*Рис. 5.12.* Неявный способ

**Пример.** На рис. 5.13 представлена система классов, служащая для моделирования процесса обслуживания системы управления климатом в теплице. Класс Plan включает правила управления климатом и имеет операцию execute (выполнить). Имеется ассоциация между этим классом и классом Controller (контроллер, управляющий климатом): экземпляры класса Plan задают климат, который должны поддерживать экземпляры класса Controller.

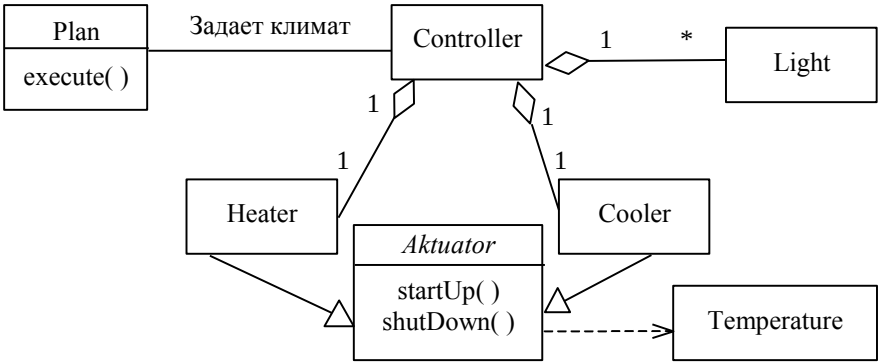


*Рис. 5.13.* Классы для описания процесса обслуживания системы управления климатом

Эта диаграмма также показывает, что класс Controller является агрегатом: его экземпляры содержат в точности по одному экземпляру классов Heater (нагреватель) и Cooler (охлаждающее устройство) и любое число экземпляров класса Light (лампочка). Оба класса Heater и Cooler являются подклассами абстрактного запускающего процесс класса Actuator, который предоставляет протоколы startUp и shutDown (начать и прекратить соответственно) и использует класс Temperature.

Связь интерфейса с реализующим его элементом можно графически представить двумя способами. Простая форма позволяет изобразить отношения между интерфейсом и реализующим его классом в виде кружка с одной стороны класса (см. рис. 4.3). Если интерфейс

представлен в расширенной форме, то отношение реализации изображается явно, как для инстанцирования, но без стереотипа (рис. 5.14). Зависимость между классом и интерфейсом показывается аналогично зависимости между классами.

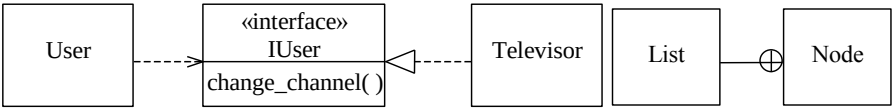


*Рис. 5.14.* Отношения между классами и интерфейсом

*Рис. 5.15.* Вложение классов

Если некоторый класс описан внутри другого класса, то он называется **вложенным** и недоступен вне объемлющего его класса. Обозначение вложения классов приведено на рис. 5.15.

**Пример.** Список состоит из узлов, связанных между собой с помощью указателей. Если класс «Узел» используется только для определения класса «Список», то его описание целесообразно скрыть путем вложения.

```
class List{
        class Node{ . . .};
        Node *pbeg, *pend; // указатели на начало и конец списка
         . . .
};
```

## 5.2. ДИАГРАММА ОБЪЕКТОВ

**Диаграмма объектов** показывает существующие объекты и связи между ними в некоторый момент времени.

Диаграмма объектов может рассматриваться как прототип: она представляет отношения, которые могут возникнуть среди элементов данного множества экземпляров классов.

Объект на диаграмме объектов изображается значком, показанным на рис. 5.16. Он совпадает со значком класса, но имя объекта подчеркивается и отсутствуют горизонтальные линии, разделяющие текст внутри значка объекта на части.


*Рис. 5.16.* Значок объекта



*Рис. 5.17.* Значок связи

Имя объекта может быть записано в одной из следующих форм:

A – только имя объекта;

: C – только класс объектов (анонимный экземпляр);

A : C – имя объекта и класса.

Если класс объекта никогда не указывается, то класс считается безымянным. Если указывается только имя класса, то объект считается безымянным. Каждый значок, не содержащий имени объекта, обозначает отдельный безымянный объект.

Объекты взаимодействуют с другими объектами через связи, которые изображаются на диаграмме прямыми линиями (рис 5.17).

На значках объектов бывает полезно указать несколько их атрибутов. Синтаксис атрибутов совпадает с синтаксисом атрибутов класса и позволяет указать их текущее значение (рис. 5.18). Имена атрибутов объектов должны соответствовать атрибутам, определенным в классе объекта или в любом из его суперклассов.

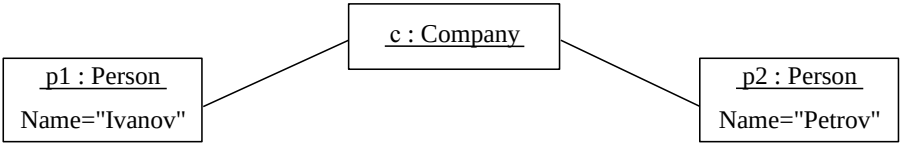


*Рис. 5.18.* Диаграмма объектов

На рис. 5.18 приведен пример диаграммы объектов, соответствующий диаграмме классов на рис. 5.6.

## 5.3. ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТЕЙ И КОММУНИКАЦИИ

На **диаграммах последовательностей** и **диаграммах коммуникации** отражаются множество объектов и отношения между ними, включая сообщения, которыми они обмениваются.

На **диаграммах последовательностей** внимание акцентируется прежде всего на временной упорядоченности сообщений. Преимущество диаграммы последовательностей в том, что на ней легче читается порядок посылки сообщений.

На диаграмме последовательностей объекты изображаются горизонтально вдоль верхней границы. Обычно инициирующий взаимодействие объект размещается слева, а остальные правее (тем дальше, чем более подчиненным является объект). Из значка каждого объекта выходит вертикальная пунктирная линия, называемая «линией жизни». Эта линия показывает пределы существования объекта.

Отправления сообщений (вызовы операций) показываются горизонтальными стрелками. Линия, обозначающая посылку сообщения, проводится от линии жизни клиента к линии жизни сервера. Первое сообщение показывается на самом высоком уровне, второе – ниже и т. д.

Вызов операции обозначается ее именем, кроме того, здесь могут быть приведены фактические параметры операции. Ответное сообщение (возвращаемое значение) указывается пунктирной стрелкой от сервера к клиенту.

**Пример.** Диаграмма последовательностей приведена на рис. 5.19. На диаграмме отражено взаимодействие трех объектов. Сценарий начинается с вызова объектом A операции f1 над объектом B. Это порождает вызов объектом B операции f2 над объектом C, что потребует вызова объектом C операции f3 над собой. Когда эта операция будет выполнена, объект B передаст возвращаемое значение r объекту A, который затем вызывает операцию f4 над объектом C.

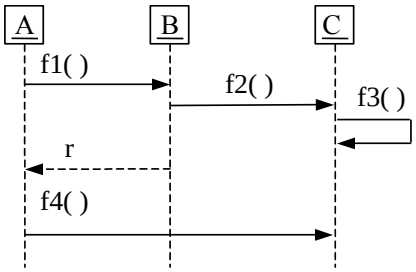


*Рис. 5.19.* Диаграмма последовательностей

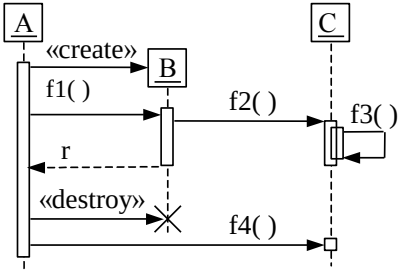


*Рис. 5.20.* Расширенная диаграмма последовательностей

Фрагмент программы на C++, соответствующий данной диаграмме, может быть следующим.

```
class Class_C{
public: void f2( ){ f3( ); . . .} void f3( ){. . .} void f4( ){. . .}
}C;
class Class_B {
public: int f1( ) { C.f2( ); . . .}
}B;
class Class_A{
public: void execute( ){ int r=B.f1( ); C.f4( ); . . .}
};
void main( ){ Class_A A; A.execute( ); }
```

Большая часть объектов, представленных на диаграмме последовательностей, существует на протяжении всего взаимодействия, поэтому их изображают в верхней части диаграммы, а их линии жизни прорисованы сверху донизу. Объекты могут создаваться и во время взаимодействий. Линии жизни таких объектов начинаются с получения сообщения со стереотипом «create». Объекты могут также уничтожаться во время взаимодействий; в таком случае их линии жизни заканчиваются получением сообщения со стереотипом «destroy», а в качестве визуального образа используется большая буква X.

На диаграммах может быть изображен **фокус управления.** Он изображается в виде вытянутого прямоугольника, показывающего промежуток времени, в течение которого объект выполняет какое-либо действие непосредственно или с помощью подчиненной процедуры (рис. 5.20). Верхняя грань прямоугольника выравнивается по временной оси с моментом начала действия, нижняя – с моментом его завершения. Вложенность фокуса управления, вызванную рекурсией (т. е. обращением к собственной операции) или обратным вызовом со стороны другого объекта, можно показать, расположив другой фокус управления чуть правее своего родителя.

На рис. 5.20 показан пример, аналогичный предыдущему, в котором, однако, объект B создается и уничтожается во время взаимодействия.

Чаще всего приходится моделировать неветвящиеся последовательные потоки управления. Однако можно моделировать и более сложные потоки, содержащие циклы и ветвления.

Цикл изображается в виде рамки, в которую вписано ключевое слово loop (рис. 5.21). В квадратных скобках указывается условие выполнения цикла.

Ветвление также изображается в виде рамки, в которую вписано ключевое слово alt (рис. 5.22). Область оператора alt разделяется на части, каждая из которых имеет свое собственное условие. Если такое условие является истинным, то выполняются функции, предусмотренные в соответствующей части.

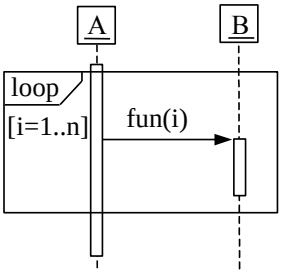


*Рис. 5.21.* Цикл

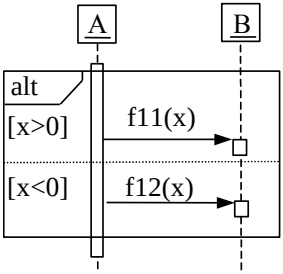


*Рис. 5.22.* Ветвление

**Диаграмма коммуникации** акцентирует внимание на структурной организации объектов, принимающих и отправляющих сообщения.

На диаграмме изображаются объекты и связи между ними, как на диаграмме объектов. Рядом с соответствующей связью на диаграмме можно записать набор сообщений. Каждое сообщение состоит из следующих трех элементов: направление вызова, вызов операции, порядковый номер.

Направление сообщения показывается стрелкой, указывающей на объект-сервер.

Вызов операции обозначается так же, как и на диаграмме последовательностей.

Порядковый номер показывает относительный порядок посылки сообщений. Сообщение с меньшим порядковым номером посылается до сообщения с большим номером. Нумерация начинается с единицы и добавляется как префикс к вызову операции. Для отображения вложенных сообщений используется следующая нотация: 1 – первое сообщение; 1.1 – первое сообщение, вложенное в сообщение 1; 1.2 – второе сообщение, вложенное в сообщение 1, и т. д.

**Пример.** Диаграмма коммуникации, соответствующая диаграмме последовательностей с рис. 5.20, показана на рис. 5.23.

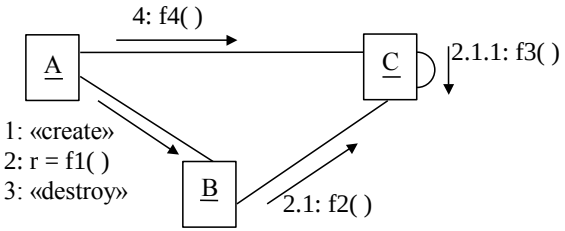


*Рис. 5.23.* Диаграмма коммуникации

Циклы моделируются вставкой после номера сообщения итерационного выражения (рис. 5.24). Итерационное выражение изображается в виде звездочки, за которой следует перечисление в квадратных скобках. Перечисление может быть опущено, если надо обозначить цикл без дальнейшей детализации.

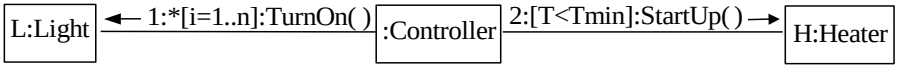


*Рис. 5.24.* Циклы и условия

Для моделирования условия после порядкового номера сообщения ставится выражение в квадратных скобках (рис. 5.24). У всех альтернативных ветвей должен быть один и тот же порядковый номер, но условия на каждой ветви должны быть заданы так, чтобы два из них не выполнялись одновременно (не перекрывались).

# 6. МНОГОКРАТНОЕ ИСПОЛЬЗОВАНИЕ ПРОГРАММНЫХ СИСТЕМ. ОСНОВЫ ВИЗУАЛЬНОГО И КОМПОНЕНТНОГО ПРОГРАММИРОВАНИЯ

Одним из главных преимуществ объектно-ориентированного программирования является возможность *многократно использовать* разработанное программное обеспечение. Под этим понимается возмож-

ность использовать фрагмент кода, разработанный для одной системы, в других системах. Разумеется, этот фрагмент кода должен представлять собой логически более или менее законченную часть. Реальное ускорение процесса разработки программного обеспечения возможно только тогда, когда конкретная программная система разрабатывается не с нуля, а с использованием готовых составных частей.

Многократное использование кода – это идеальный вариант передачи знаний. Труд, знания и опыт разработчика, заложенные в готовом продукте – программном коде, передаются без потерь.

Конечно, и раньше для процедурно-ориентированных языков программирования типа Фортран, Паскаль или Си создавались *библиотеки функций*, реализующих наиболее употребимые алгоритмы обработки данных.

В рамках объектно-ориентированного подхода разрабатываются *библиотеки классов*. Использование библиотек классов повышает скорость разработки программ и экономит значительные усилия разработчиков. Однако нельзя сказать, что у них нет недостатков.

Использование библиотек классов требует определенных усилий для их изучения и понимания того, как они устроены. Современные среды программирования используют различные графические средства построения программ, которые позволяют визуально манипулировать программой и упростить разработку классов, однако их возможности при работе с библиотеками классов ограниченны.

Библиотека классов должна быть написана на том же языке программирования, что и разрабатываемая программа.

Для того чтобы воспользоваться библиотекой, необходимо написать программу с вызовами нужных функций или порождением необходимых классов и оттранслировать эту программу.

Подобные соображения привели к появлению концепции компонента. **Компонент** – это программный модуль или объект, который готов для использования в качестве составного блока программы и которым можно *визуально манипулировать* во время разработки программ.

Чем отличается компонент от класса в библиотеке классов или функции в библиотеке функций?

Во-первых, компонент – это объект, т. е. компонент объединяет состояние и интерфейс. Состояние компонента может быть изменено только с помощью посылки сообщений (вызова операций).

Во-вторых, у компонента имеются два типа интерфейсов: интерфейс времени выполнения и интерфейс времени проектирования.

**Интерфейс времени выполнения** – это тот интерфейс, который управляет работой компонента.

**Интерфейс времени проектирования** – это интерфейс, который позволяет узнать, каков интерфейс времени выполнения. Интерфейс времени проектирования позволяет включать компоненты в современные среды программирования. Кроме того, интерфейс времени проектирования позволяет опрашивать компонент уже во время выполнения, динамически проверяя, какие методы и интерфейсы он реализует.

В-третьих, неважно, на каком языке программирования разработан компонент. Более того, совершенно не обязательно, чтобы компонент был реализован с помощью объектно-ориентированного языка программирования. Он должен удовлетворять определенным внешним характеристикам, но его реализация полностью закрыта. В идеале компонент должен быть нейтрален по отношению к языку программирования, т. е. его можно использовать в программе на любом языке, который поддерживает компонентную технологию.

Компонент может быть включен непосредственно в программу (локальный компонент) или существовать независимо, в том числе и на другом компьютере.

Отличительными признаками **локального компонента** является то, что он включается в программу на стадии разработки и существует только как часть вызывающей его программы. Физически копия компонента может быть одна для нескольких программ, либо каждая программа создает собственную копию. Однако логически они независимы, и вызывающая программа управляет созданием и уничтожением компонента.

Другой вид компонентов – это **независимые компоненты**, существующие вне использующей их программы. Чаще всего такие компоненты применяются в распределенных системах, состоящих из большого числа ЭВМ, соединенных между собой сетью.

Компонентное построение прежде всего используется для построения *графического интерфейса* программ. По мере того как графические возможности систем становятся все более изощренными, затраты на создание интерфейсов программ растут. При этом их программирование и отладка не только трудоемки, но и требуют высокой квалификации. Разнообразные библиотеки классов, хотя и упрощают в какой-

то мере программирование, ограничивают гибкость. Для построения удобных в использовании графических программ достаточно часто нужно использовать весь арсенал средств, предоставляемых графической подсистемой операционной системы.

Средства проектирования пользовательских графических интерфейсов в большинстве случаев работают в режиме графических редакторов, при котором программист рисует на экране монитора диалоги, помещая разнообразные элементы (кнопки, списки, поля) простым движением манипулятора «мышь».

Компоненты легко вписываются в концепцию графического редактора, позволяя программисту манипулировать ими с той же легкостью, что и стандартными графическими элементами.

Но наличие графического интерфейса не является обязательным признаком компонента (в отличие от возможности визуального манипулирования). Компонент может реализовывать вычислительную задачу или связывать программу с другой подсистемой.

Компонент может реализовывать какой-либо графический элемент, например, в качестве компонента может выступать кнопка на экране, при нажатии которой происходит выполнение какого-либо действия, или более сложный элемент, например календарь с расписанием отправлений поездов.

# 7. ОСНОВЫ ТЕСТИРОВАНИЯ В ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ СИСТЕМАХ

Без этапа тестирования программного обеспечения процесс разработки будет неполным.

Тестирование является достаточно независимым процессом, применимым к программному обеспечению, разработанному с помощью любого метода проектирования. Тем не менее объектно-ориентированный подход привносит свои особенности.

Традиционно тестирование делится на тестирование элементов, интеграционное тестирование и системное тестирование.

**На уровне элементов** тестирование объектно-ориентированных программ различается по следующим показателям:

- определение единиц тестирования;

- тестирование наследования;
- тестирование полиморфизма.

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

**Тестирование наследования** состоит в тестировании методов, унаследованных классом от своего суперкласса. Если суперкласс уже прошел тестирование, нужно ли повторять тестирование для унаследованных методов? Вопреки достаточно распространенным надеждам программистов перетестирование необходимо. Основная причина та, что методы выполняются в новом контексте.

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

**Интеграционное тестирование** представляет собой тестирование того, как отдельные элементы программы работают вместе. Свойства объектно-ориентированных языков исключают целый ряд возможных ошибок, прежде всего за счет строгого определения внешних интерфейсов классов и объектов. Однако это не означает, что интеграционное тестирование становится легче.

**Системное тестирование** проверяет всю программную систему целиком и строится в большинстве случаев по принципу «черного ящика», когда тестирующий знает только внешние характеристики системы, но не знает, как она работает.

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК


1. *Бадд Т.* Объектно-ориентированное программирование в действии / Т. Бадд. – СПб.: Питер, 1997.

2. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / Г. Буч. – М.: Изд-во «Бином», 2001.

3. *Буч Г.* Язык UML. Руководство пользователя / Г. Буч, Д. Рамбо, А. Джекобсон. – М.: ДМК, 2006.

4. *Иванова Г.С.* Объектно-ориентированное программирование / Г.С. Иванова, Т.Н. Ничушкина, Е.К. Пугачев. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2007.

5. *Объектно-ориентированный* анализ и проектирование с примерами приложений / Г. Буч, Р.А. Максимчук, М.У. Энгл, Б.Дж. Янг, Д. Коналлен, К.А. Хьюстон. – М.: И.Д. «Вильямс», 2010.

6. *Павловская Т.А.* C/C++. Программирование на языке высокого уровня / Т.А. Павловская.– СПб.: Питер, 2010.

7. *Павловская Т.А.* C++. Объектно-ориентированное программирование: Практикум / Т.А. Павловская, Ю.А. Щупак. – СПб.: Питер, 2008.

8. *Подбельский В.В.* Язык C++: учеб. пособие / В.В. Подбельский. – М.: Финансы и статистика, 2007.

9. *Пол А.* Объектно-ориентированное программирование на C++ / А. Пол. – СПб.; М.: Невский диалект; Изд-во «Бином», 1999.

10. *Романовский К.Ю.* Объектно-ориентированный подход и диаграммы классов в UML // Объектно-ориентированное визуальное моделирование / К.Ю. Романовский, С.В. Кузнецов, Д.В. Кознов. – СПб.: Изд-во С.-Петербургского ун-та, 1999. – С. 21–56.

11. *Страуструп Б.* Язык программирования C++ / Б. Страуструп. – СПб.; М.: Невский диалект; Изд-во «Бином», 2008.

12. *Пышкин Е.В.* Основные концепции и механизмы объектно-ориентированного программирования / Е.В. Пышкин. – СПб.: БХВ-Петербург, 2005.

13. *Хабибуллин И.Ш.* Программирование на языке высокого уровня. C/C++ / И.Ш. Хабибуллин. – СПб.: БХВ-Петербург, 2006.

14. *Фридман А.Л.* Основы объектно-ориентированного программирования на языке C++ / А.Л. Фридман. – М.: Горячая линия – Телеком, 2001.

15. *Фридман А.Л.* Основы объектно-ориентированной разработки программных систем / А.Л. Фридман. – М.: Финансы и статистика, 2000.

**ОГЛАВЛЕНИЕ**

**Лисицин Даниил Валерьевич**

**ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ**

**2-е изд-е, перераб. и доп.**

**Конспект лекций**

Редактор *И.Л. Кескевич*
Выпускающий редактор *И.П. Брованова*
Корректор *И.Е. Семенова*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *С.И. Ткачева*

---

Подписано в печать 24.08.2010. Формат 60 × 84 1/16. Бумага офсетная. Тираж 200 экз.
Уч.-изд. л. 5,11. Печ. л. 5,5. Изд. № 154. Заказ № . Цена договорная

---

Отпечатано в типографии
Новосибирского государственного технического университета
630092, г. Новосибирск, пр. К. Маркса, 20