

Лекция 2: Основы объектно-ориентированного программирования

Методология процедурно-ориентированного программирования

Появление первых электронных вычислительных машин, или компьютеров, ознаменовало новый этап в развитии техники вычислений. Казалось, достаточно разработать последовательность элементарных действий, каждое из которых можно преобразовать в понятные компьютеру инструкции, и любая вычислительная задача будет решена. Эта идея оказалась настолько жизнеспособной, что долгое время доминировала над всем процессом разработки программ. Появились специализированные языки программирования, созданные для разработки программ, предназначенных для решения вычислительных задач. Примерами таких языков могут служить FOCAL (FOrmula CALculator) и FORTRAN (FORmula TRANslator).

Основой такой методологии разработки программ являлась процедурная, или алгоритмическая, организация структуры программного кода. Это было настолько естественно для решения вычислительных задач, что целесообразность такого подхода ни у кого не вызывала сомнений. Исходным в данной методологии было понятие алгоритма. Алгоритм - это способ решения вычислительных и других задач, точно описывающий определенную последовательность действий, которые необходимо выполнить для достижения заданной цели. Примерами алгоритмов являются хорошо известные правила нахождения корней квадратного уравнения или системы линейных уравнений.

При увеличении объемов программ для упрощения их разработки появилась необходимость разбивать большие задачи на подзадачи. В языках программирования возникло и закрепилось новое понятие процедуры. Использование процедур позволило разбивать большие задачи на подзадачи и таким образом упростило написание больших программ. Кроме того, процедурный подход позволил уменьшить объем программного кода за счет написания часто используемых кусков кода в виде процедур и их применения в различных частях программы.

Как и алгоритм, процедура представляет собой законченную последовательность действий или операций, направленных на решение отдельной задачи. В языках программирования появилась специальная синтаксическая конструкция, которая также получила название процедуры. Например, на языке Pascal описание процедуры выглядит следующим образом:

```
procedure printGreeting(name: String)
Begin
    Print("Hello, ");
    PrintLn(name);
End;
```

Назначение данной процедуры - вывести на экран приветствие Hello, Name, где Name передается в процедуру в качестве входного параметра.

Со временем вычислительные задачи становились все сложнее, а значит, и решающие их программы увеличивались в размерах. Их разработка превратилась в серьезную проблему. Когда программа становится все больше, ее приходится разделять на все более мелкие фрагменты. Основой для такого разбиения как раз и стала процедурная декомпозиция, при которой отдельные части программы, или модули, представляли собой совокупность процедур для решения одной или нескольких задач. Одна из основных особенностей процедурного программирования заключается в том, что оно позволило создавать библиотеки подпрограмм (процедур), которые можно было бы использовать повторно в различных проектах или в рамках одного проекта. При процедурном подходе для визуального представления алгоритма выполнения программы применяется так называемая **блок-схема**. Соответствующая система графических обозначений была зафиксирована в ГОСТ 19.701-90. Пример *блок-схемы* изображен на рисунке ([рис. 2.1](#)).

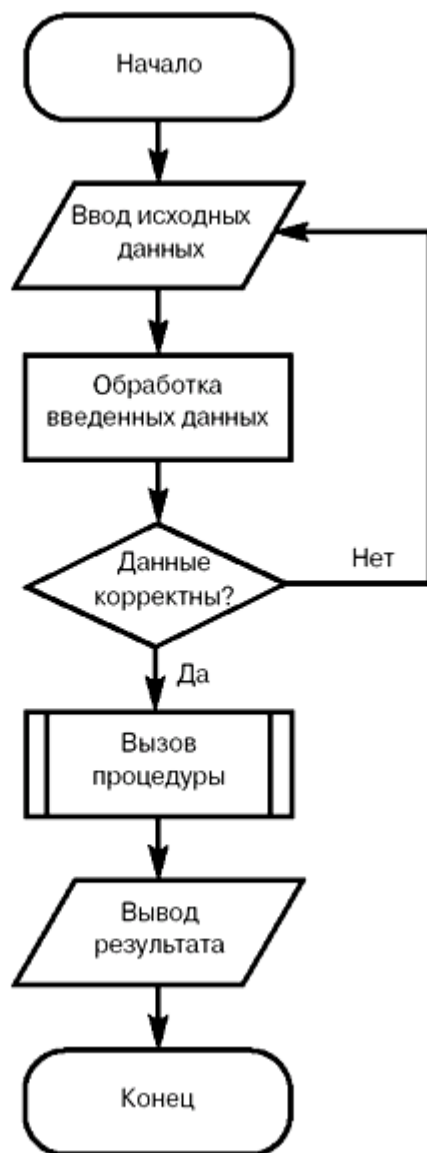


Рис. 2.1. Пример блок-схемы.

Появление и интенсивное использование условных операторов и оператора безусловного перехода стало предметом острых дискуссий среди специалистов по программированию. Дело в том, что бесконтрольное применение в программе оператора безусловного перехода `goto` может заметно усложнить понимание кода. Такие запутанные программы сравнивали с порцией спагетти (*bowl of spaghetti*), имея ввиду многочисленные переходы от одного фрагмента программы к другому, или, что еще хуже, возврат от конечных операторов программы к начальным. Ситуация казалась настолько драматичной, что многие предлагали исключить оператор `goto` из языков программирования. Именно с этого времени отсутствие безусловных переходов стали считать хорошим стилем программирования.

Дальнейшее увеличение программных систем способствовало формированию новой точки зрения на процесс разработки программ и написания программных кодов, которая получила название методологии структурного программирования. Ее основой является процедурная декомпозиция предметной области решаемой задачи и организация отдельных модулей в виде совокупности процедур. В рамках этой методологии получило развитие нисходящее проектирование программ, или проектирование "сверху вниз". Пик популярности идей структурного программирования приходится на конец 70-х - начало 80-х годов.

В этот период основным показателем сложности разработки программ считался ее размер. Вполне серьезно обсуждались такие оценки сложности программ, как количество строк программного кода. Правда, при этом делались некоторые предположения относительно синтаксиса самих строк, которые должны были соответствовать определенным требованиям. Например, каждая строка кода должна была содержать не более одного оператора. Общая трудоемкость разработки программ оценивалась специальной единицей измерения - "человеко-месяц", или "человеко-год". А профессионализм программиста напрямую связывался с количеством строк программного кода, который он мог написать и отладить в течение, скажем, месяца.

Методология объектно-ориентированного программирования

Увеличение размеров программ приводило к необходимости привлечения большего числа программистов, что, в свою очередь, потребовало дополнительных ресурсов для организации их согласованной работы. В процессе разработки приложений заказчик зачастую изменял функциональные требования, что еще более усложняло процесс создания программного обеспечения.

Но не менее важными оказались качественные изменения, связанные со смещением акцента использования компьютеров. В эпоху "больших машин" основными потребителями программного обеспечения были такие крупные

заказчики, как большие производственные предприятия, финансовые компании, государственные учреждения. Стоимость таких вычислительных устройств для небольших предприятий и организаций была слишком высока.

Позже появились персональные компьютеры, которые имели гораздо меньшую стоимость и были значительно компактнее. Это позволило широко использовать их в малом и среднем бизнесе. Основными задачами в этой области являются обработка данных и манипулирование ими, поэтому вычислительные и расчетно-алгоритмические задачи с появлением персональных компьютеров отошли на второй план. Как показала практика, традиционные методы процедурного программирования не способны справиться ни с нарастающей сложностью программ и их разработки, ни с необходимостью повышения их надежности. Во второй половине 80-х годов возникла настоятельная потребность в новой методологии программирования, которая была бы способна решить весь этот комплекс проблем. Ею стало *объектно-ориентированное программирование* (ООП).

После составления технического задания начинается этап проектирования, или дизайна, будущей системы. Объектно-ориентированный подход к проектированию основан на представлении предметной области задачи в виде множества моделей для независимой от языка разработки программной системы на основе ее прагматики.

Последний термин нуждается в пояснении. Прагматика определяется целью разработки программной системы, например, обслуживание клиентов банка, управление работой аэропорта, обслуживание чемпионата мира по футболу и т.п. В формулировке цели участвуют предметы и понятия реального мира, имеющие отношение к создаваемой системе (см. рисунок 2.2). При объектно-ориентированном подходе эти предметы и понятия заменяются моделями, т.е. определенными формальными конструкциями.

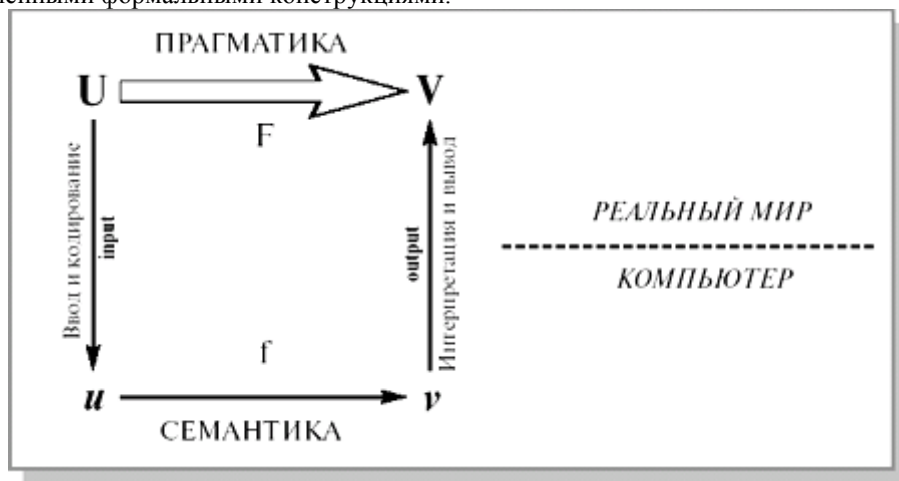


Рис. 2.2. Семантика (смысл программы с точки зрения выполняющего ее компьютера) и прагматика (смысл программы с точки зрения ее пользователей).

Модель содержит не все признаки и свойства представляемого ею предмета или понятия, а только те, которые существенны для разрабатываемой программной системы. Таким образом, модель "беднее", следовательно, проще представляемого ею предмета или понятия.

Простота модели по отношению к реальному предмету позволяет сделать ее формальной. Благодаря такому характеру моделей при разработке можно четко выделить все зависимости и операции над ними в создаваемой программной системе. Это упрощает как разработку и изучение (анализ) моделей, так и их реализацию на компьютере.

Объектно-ориентированный подход обладает такими преимуществами, как:

- уменьшение сложности программного обеспечения;
- повышение надежности программного обеспечения;
- обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов;
- обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

Более детально преимущества и недостатки *объектно-ориентированного программирования* будут рассмотрены в конце лекции, так как для их понимания необходимо знание основных понятий и положений ООП.

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. ООП является одним из наиболее интенсивно развивающихся направлений теоретического и прикладного программирования.

Объекты

По определению будем называть **объектом** понятие, абстракцию или любой предмет с четко очерченными границами, имеющий смысл в контексте рассматриваемой прикладной проблемы. Введение *объектов* преследует две цели:

- понимание прикладной задачи (проблемы);
- введение основы для реализации на компьютере.

Примеры *объектов*: форточка, Банк "Империал", Петр Сидоров, дело № 7461, сберкнижка и т.д.

Каждый *объект* имеет определенное время жизни. В процессе выполнения программы, или функционирования какой-либо реальной системы, могут создаваться новые *объекты* и уничтожаться уже существующие.

Гради Буч дает следующее определение *объекта*:

Объект - это мыслимая или реальная сущность, обладающая характерным поведением и отличительными характеристиками и являющаяся важной в предметной области.

Каждый объект имеет состояние, обладает четко определенным поведением и уникальной идентичностью.

Состояние

Рассмотрим пример. Любой человек может находиться в некотором положении (*состоянии*): стоять, сидеть, лежать, и - в то же время совершать какие либо действия.

Например, человек может прыгать, если он стоит, и не может - если он лежит, для этого ему потребуется сначала встать. Также в *объектно-ориентированном программировании* состояние объекта может определяться наличием или отсутствием связей между моделируемым объектом и другими объектами. Более подробно все возможные связи между объектами будут рассмотрены в разделе "Типы отношений между классами".

Например, если у человека есть удочка (у него есть связь с объектом "Удочка"), он может ловить рыбу, а если удочки нет, то такое действие невозможно. Из этих примеров видно, что набор действий, которые может совершать человек, зависит от параметров объекта, его моделирующего.

Для рассмотренных выше примеров такими характеристиками, или атрибутами, объекта "Человек" являются:

- текущее положение человека (стоит, сидит, лежит);
- наличие удочки (есть или нет).

В конкретной задаче могут появиться и другие свойства, например, физическое состояние, здоровье (больной человек обычно не прыгает).

Состояние (state) - совокупный результат поведения объекта: одно из стабильных условий, в которых объект может существовать, охарактеризованных количественно; в любой момент времени состояние объекта включает в себя перечень (обычно статический) свойств объекта и текущие значения (обычно динамические) этих свойств.

Поведение

Для каждого объекта существует определенный набор действий, которые с ним можно произвести. Например, возможные действия с некоторым файлом операционной системы ПК:

- создать;
- открыть;
- читать из файла;
- писать в файл;
- закрыть;
- удалить.

Результат выполнения действий зависит от состояния объекта на момент совершения действия, т.е. нельзя, например, удалить файл, если он открыт кем-либо (заблокирован). В то же время действия могут менять внутреннее состояние объекта - при открытии или закрытии файла свойство "открыт" принимает значения "да" или "нет", соответственно.

Программа, написанная с использованием ООП, обычно состоит из множества объектов, и все эти объекты взаимодействуют между собой. Обычно говорят, что взаимодействие между объектами в программе происходит посредством передачи сообщений между ними.

В терминологии объектно-ориентированного подхода понятия "действие", "сообщение" и "метод" являются синонимами. Т.е. выражения "выполнить действие над объектом", "вызвать метод объекта" и "послать сообщение объекту для выполнения какого-либо действия" эквивалентны. Последняя фраза появилась из следующей модели. Программу, построенную по технологии ООП, можно представить себе как виртуальное пространство, заполненное объектами, которые условно "живут" некоторой жизнью. Их активность проявляется в том, что они вызывают друг у друга методы, или посылают друг другу сообщения. Внешний интерфейс объекта, или набор его методов, - это описание того, какие сообщения он может принимать.

Поведение (behavior) - действия и реакции объекта, выраженные в терминах передачи сообщений и изменения состояния; видимая извне и воспроизводимая активность объекта.

Уникальность

Уникальность - это то, что отличает объект от других объектов. Например, у вас может быть несколько одинаковых монет. Даже если абсолютно все их свойства (атрибуты) одинаковы (год выпуска, номинал и т.д.) и при этом вы можете использовать их независимо друг от друга, они по-прежнему остаются разными монетами.

В машинном представлении под параметром уникальности объекта чаще всего понимается адрес размещения объекта в памяти.

Identity (уникальность) объекта состоит в том, что всегда можно определить, указывают две ссылки на один и тот же объект или на разные объекты. При этом два объекта могут во всем быть похожими, их образ в памяти может представляться одинаковыми последовательностями байтов, но, тем не менее, их Identity может быть различна.

Наиболее распространенной ошибкой является понимание уникальности как имени ссылки на объект. Это неверно, т.к. на один объект может указывать несколько ссылок, и ссылки могут менять свои значения (ссылаются на другие объекты).

Итак, **уникальность** (identity) - свойство объекта; то, что отличает его от других объектов.

Классы

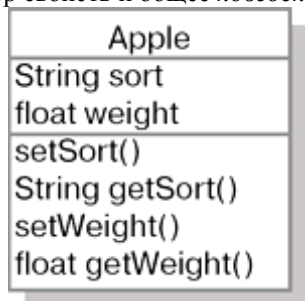
Все монеты из предыдущего примера принадлежат одному и тому же классу объектов (именно с этим связана их одинаковость). Номинальная стоимость монеты, металл, из которого она изготовлена, форма - это атрибуты класса. Совокупность атрибутов и их значений характеризует объект. Наряду с термином "атрибут" часто используют термины "свойство" и "поле", которые в объектно-ориентированном программировании являются синонимами.

Все объекты одного и того же класса описываются одинаковыми наборами атрибутов. Однако объединение объектов в классы определяется не наборами атрибутов, а семантикой. Так, например, объекты "конюшня" и "лошадь"

могут иметь одинаковые атрибуты: цена и возраст. При этом они могут относиться к одному *классу*, если рассматриваются в задаче просто как товар, либо к разным *классам*, если в рамках поставленной задачи будут использоваться по-разному, т.е. над ними будут совершаться различные действия.

Объединение *объектов* в *классы* позволяет рассмотреть задачу в более общей постановке. *Класс* имеет имя (например, "лошадь"), которое относится ко всем *объектам* этого *класса*. Кроме того, в *классе* вводятся имена атрибутов, которые определены для *объектов*. В этом смысле описание *класса* аналогично описанию типа структуры или записи (record), широко применяющихся в процедурном программировании; при этом каждый *объект* имеет тот же смысл, что и экземпляр структуры (переменная или константа соответствующего типа).

Формально *класс* - это шаблон *поведения объектов* определенного типа с заданными параметрами, определяющими *состояние*. Все экземпляры одного *класса* (*объекты*, порожденные от одного *класса*) имеют один и тот же набор свойств и общее *поведение*, то есть одинаково реагируют на одинаковые сообщения.



В соответствии с **UML** (Unified Modelling Language - унифицированный язык моделирования), *класс* имеет следующее графическое представление.

Класс изображается в виде прямоугольника, состоящего из трех частей. В верхней части помещается название *класса*, в средней - свойства *объектов класса*, в нижней - действия, которые можно выполнять с *объектами* данного *класса* (методы).

Каждый *класс* также может иметь специальные методы, которые автоматически вызываются при создании и уничтожении *объектов* этого *класса*:

- **конструктор** (constructor) - выполняется при создании *объектов*;
- **деструктор** (destructor) - выполняется при уничтожении *объектов*.

Обычно *конструктор* и *деструктор* имеют специальный синтаксис, который может отличаться от синтаксиса, используемого для написания обычных методов *класса*.

Инкапсуляция

Инкапсуляция (encapsulation) - это сокрытие реализации *класса* и отделение его внутреннего представления от внешнего (интерфейса). При использовании объектно-ориентированного подхода не принято применять прямой доступ к свойствам какого-либо *класса* из методов других *классов*. Для доступа к свойствам *класса* принято задействовать специальные методы этого *класса* для получения и изменения его свойств.

Внутри *объекта* данные и методы могут обладать различной степенью открытости (или доступности). Степени доступности, принятые в языке Java, подробнее будут рассмотрены в дальнейших лекциях. Они позволяют более тонко управлять свойством *инкапсуляции*.

Открытые члены *класса* составляют внешний интерфейс *объекта*. Это та функциональность, которая доступна другим *классам*. Закрытыми обычно объявляются все свойства *класса*, а также вспомогательные методы, которые являются деталями реализации и от которых не должны зависеть другие части системы.

Благодаря сокрытию реализации за внешним интерфейсом *класса* можно менять внутреннюю логику отдельного *класса*, не меняя код остальных компонентов системы. Это свойство называется **модульностью**.

Обеспечение доступа к свойствам *класса* только через его методы также дает ряд преимуществ. Во-первых, так гораздо проще контролировать корректные значения полей, ведь прямое обращение к свойствам отслеживать невозможно, а значит, им могут присвоить некорректные значения.

Во-вторых, не составит труда изменить способ хранения данных. Если информация станет храниться не в памяти, а в долговременном хранилище, таком как файловая система или база данных, потребуется изменить лишь ряд методов одного *класса*, а не вводить эту функциональность во все части системы.

Наконец, программный код, написанный с использованием данного принципа, легче отлаживать. Для того, чтобы узнать, кто и когда изменил свойство интересующего нас *объекта*, достаточно добавить вывод отладочной информации в тот метод *объекта*, посредством которого осуществляется доступ к свойству этого *объекта*. При использовании прямого доступа к свойствам *объектов* программисту пришлось бы добавлять вывод отладочной информации во все участки кода, где используется интересующий нас *объект*.

Наследование

Наследование (inheritance) - это отношение между *классами*, при котором *класс* использует структуру или *поведение* другого *класса* (одиночное наследование), или других (множественное наследование) *классов*. Наследование вводит иерархию "общее/частное", в которой **подкласс** наследует от одного или нескольких более общих **суперклассов**. Подклассы обычно дополняют или переопределяют унаследованную структуру и *поведение*.

В качестве примера можно рассмотреть задачу, в которой необходимо реализовать *классы* "Легковой автомобиль" и "Грузовой автомобиль". Очевидно, эти два *класса* имеют общую функциональность. Так, оба они имеют 4 колеса, двигатель, могут перемещаться и т.д. Всеми этими свойствами обладает любой автомобиль, независимо от того,

грузовой он или легковой, 5- или 12-местный. Разумно вынести эти общие свойства и функциональность в отдельный *класс*, например, "Автомобиль" и наследовать от него *классы* "Легковой автомобиль" и "Грузовой автомобиль", чтобы избежать повторного написания одного и того же кода в разных *классах*.



Отношение обобщения обозначается сплошной линией с треугольной стрелкой на конце. Стрелка указывает на более общий *класс* (**класс-предок** или *суперкласс*), а ее отсутствие - на более специальный *класс* (**класс-потомок** или *подкласс*).

Использование *наследования* способствует уменьшению количества кода, созданного для описания схожих сущностей, а также способствует написанию более эффективного и гибкого кода.

В рассмотренном примере применено одиночное *наследование*. Некоторый *класс* также может наследовать свойства и *поведение* сразу нескольких *классов*. Наиболее популярным примером применения множественного *наследования* является проектирование системы учета товаров в зоомагазине.

Все животные в зоомагазине являются наследниками *класса* "Животное", а также наследниками *класса* "Товар". Т.е. все они имеют возраст, нуждаются в пище и воде и в то же время имеют цену и могут быть проданы.

Множественное *наследование* на диаграмме изображается точно так же, как одиночное, за исключением того, что линии *наследования* соединяют *класс-потомок* сразу с несколькими *суперклассами*.

Не все объектно-ориентированные языки программирования содержат языковые конструкции для описания множественного *наследования*.

В языке Java множественное *наследование* имеет ограниченную поддержку через интерфейсы и будет рассмотрено в лекции 8.

Полиморфизм

Полиморфизм является одним из фундаментальных понятий в *объектно-ориентированном программировании* наряду с *наследованием* и *инкапсуляцией*. Слово "полиморфизм" греческого происхождения и означает "имеющий много форм". Чтобы понять, что оно означает применительно к *объектно-ориентированному программированию*, рассмотрим пример.

Предположим, мы хотим создать векторный графический редактор, в котором нам нужно описать в виде *классов* набор графических примитивов - Point, Line, Circle, Box и т.д. У каждого из этих *классов* определим метод draw для отображения соответствующего примитива на экране.

Очевидно, придется написать код, который при необходимости отобразить рисунок будет последовательно перебирать все примитивы, на момент отрисовки находящиеся на экране, и вызывать метод draw у каждого из них. Человек, незнакомый с *полиморфизмом*, вероятнее всего, создаст несколько массивов (отдельный массив для каждого типа примитивов) и напишет код, который последовательно переберет элементы из каждого массива и вызовет у каждого элемента метод draw. В результате получится примерно следующий код:

```
...
//создание пустого массива, который может содержать объекты Point с максимальным объемом 1000
Point[] p = new Point[1000];

Line[] l = new Line[1000];
Circle[] c = new Circle[1000];
Box[] b = new Box[1000];
...
// предположим, в этом месте происходит заполнение всех массивов соответствующими объектами
...
for(int i = 0; i < p.length; i++) {
    //цикл с перебором всех ячеек массива.
    //вызов метода draw() в случае, если ячейка не пустая.
    if(p[i]!=null) p[i].draw();
}

for(int i = 0; i < l.length;i++) {
    if(l[i]!=null) l.draw();
}

for(int i = 0; i < c.length;i++) {
    if(c[i]!=null) c.draw();
}
```



```

for(int i = 0; i < b.length;i++) {
    if(b[i]!=null) b.draw();
}
...

```

Недостатком написанного выше кода является дублирование практически идентичного кода для отображения каждого типа примитивов. Также неудобно то, что при дальнейшей модернизации нашего графического редактора и добавлении возможности рисовать новые типы графических примитивов, например Text, Star и т.д., при таком подходе придется менять существующий код и добавлять в него определения новых массивов, а также обработку содержащихся в них элементов.

Используя *полиморфизм*, мы можем значительно упростить реализацию подобной функциональности. Прежде всего, создадим общий родительский *класс* для всех наших *классов*. Пусть таким *классом* будет Point. В результате получим иерархию *классов*, которая изображена на рисунке 2.3.

У каждого из дочерних *классов* метод draw переопределен таким образом, чтобы отображать экземпляры каждого *класса* соответствующим образом.

Для описанной выше иерархии *классов*, используя *полиморфизм*, можно написать следующий код:

```

...
Point p[] = new Point[1000];
p[0] = new Circle();
p[1] = new Point();
p[2] = new Box();
p[3] = new Line();
...
for(int i = 0; i < p.length;i++) {
    if(p[i]!=null) p[i].draw();
}
...

```

В описанном выше примере массив p[] может содержать любые *объекты*, порожденные от наследников *класса* Point. При вызове какого-либо метода у любого из элементов этого массива будет выполнен метод того *объекта*, который содержится в ячейке массива. Например, если в ячейке p[0] находится *объект* Circle, то при вызове метода draw следующим образом:

```

p[0].draw()

```

нарисуется круг, а не точка.

В заключение приведем формальное определение *полиморфизма*.

Полиморфизм (polymorphism) - положение теории типов, согласно которому имена (например, переменных) могут обозначать *объекты* разных (но имеющих общего родителя) *классов*. Следовательно, любой *объект*, обозначаемый полиморфным именем, может по-своему реагировать на некий общий набор операций [2].

В процедурном программировании тоже существует понятие *полиморфизма*, которое отличается от рассмотренного механизма в *ООП*. Процедурный *полиморфизм* предполагает возможность создания нескольких процедур или функций с одним и тем же именем, но разным количеством или различными типами передаваемых параметров. Такие одноименные функции называются *перегруженными*, а само явление - *перегрузкой* (overloading). *Перегрузка* функций существует и в *ООП* и называется *перегрузкой* методов.

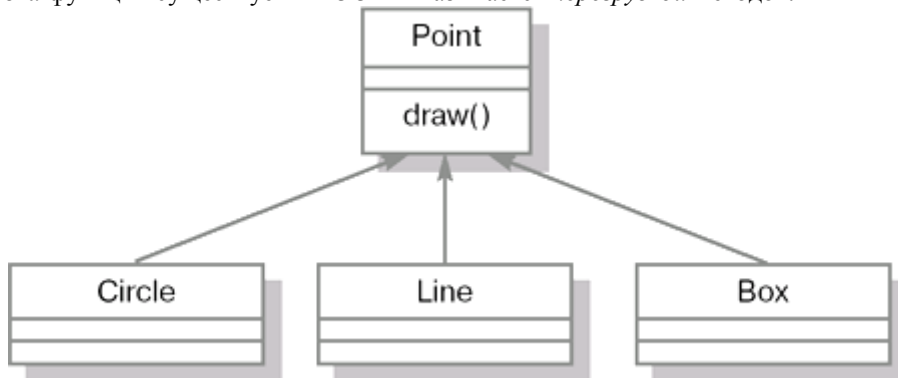


Рис. 2.3. Пример иерархии *классов*.

Примером использования *перегрузки* методов в языке Java может служить *класс* PrintWriter, который применяется, в частности, для вывода сообщений на консоль. Этот *класс* имеет множество методов println, которые различаются типами и/или количеством входных параметров. Вот лишь несколько из них:

```

void println()
    // переход на новую строку
void println(boolean x)
    // выводит значение булевской переменной (true или false)
void println(String x)
    // выводит строку – значение текстового параметра.

```

Определенные сложности возникают при вызове *перегруженных методов*. В Java существуют специальные правила, которые позволяют решать эту проблему. Они будут рассмотрены в соответствующей лекции.

Типы отношений между классами

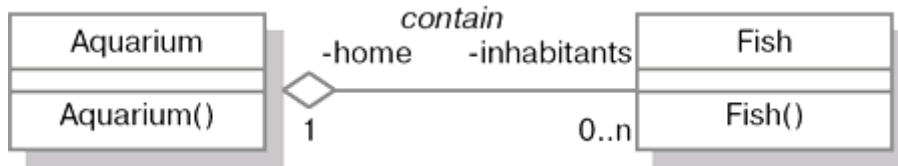
Как правило, любая программа, написанная на объектно-ориентированном языке, представляет собой некоторый набор связанных между собой *классов*. Можно провести аналогию между написанием программы и строительством дома. Подобно тому, как стена складывается из кирпичей, компьютерная программа с использованием *ООП* строится из *классов*. Причем эти *классы* должны иметь представление друг о друге, для того чтобы сообща выполнять поставленную задачу.

Возможны следующие связи между *классами* в рамках объектной модели (приводятся лишь наиболее простые и часто используемые виды связей, подробное их рассмотрение выходит за рамки этой ознакомительной лекции):

- *агрегация* (Aggregation);
- *ассоциация* (Association);
- *наследование* (Inheritance);
- *метаклассы* (Metaclass).

Агрегация

Отношение между *классами* типа "содержит" (contain) или "состоит из" называется агрегацией, или включением. Например, если аквариум наполнен водой и в нем плавают рыбки, то можно сказать, что аквариум агрегирует в себе воду и рыбок.



Такое отношение включения, или агрегации (aggregation), изображается линией с ромбиком на стороне того *класса*, который выступает в качестве владельца, или контейнера. Необязательное название отношения записывается посередине линии.

В нашем примере отношение `contain` является двунаправленным. *Объект класса* `Aquarium` содержит несколько *объектов* `Fish`. В то же время каждая рыбка "знает", в каком именно аквариуме она живет. Каждый *класс* имеет свою роль в агрегации, которая указывает, какое место занимает *класс* в данном отношении. Имя роли не является обязательным элементом обозначений и может отсутствовать на диаграмме. В примере можно видеть роль `home` *класса* `Aquarium` (аквариум является домом для рыбок), а также роль `inhabitants` *класса* `Fish` (рыбки являются обитателями аквариума). Название роли обычно совпадает с названием соответствующего поля в *классе*. Изображение такого поля на диаграмме излишне, если уже указано имя роли. Т.е. в данном случае *класс* `Aquarium` будет иметь свойство (поле) `inhabitants`, а *класс* `Fish` - свойство `home`.

Число *объектов*, участвующих в отношении, записывается рядом с именем роли. Запись `"0..n"` означает "от нуля до бесконечности". Приняты также обозначения:

- `"1..n"` - от единицы до бесконечности;
- `"0"` - ноль;
- `"1"` - один;
- `"n"` - фиксированное количество;
- `"0..1"` - ноль или один.

Код, описывающий рассмотренную модель и явление агрегации, может выглядеть, например, следующим образом:

```
// определение класса Fish
public class Fish {
    // определения поля home (ссылка на объект Aquarium)
    private Aquarium home;

    public Fish() {
    }
}

// определение класса Aquarium
public class Aquarium {
    // определения поля inhabitants (массив ссылок на объекты Fish)
    private Fish inhabitants[];
    public Aquarium() {
    }
}
```

Ассоциация

Если *объекты* одного *класса* ссылаются на один или более *объектов* другого *класса*, но ни в ту, ни в другую сторону отношение между *объектами* не носит характера "владения", или контейнеризации, такое отношение называют *ассоциацией* (association). Отношение *ассоциации* изображается так же, как и отношение агрегации, но линия, связывающая *классы*, - простая, без ромбика.

В качестве примера можно рассмотреть программиста и его компьютер. Между этими двумя *объектами* нет агрегации, но существует четкая взаимосвязь. Так, всегда можно установить, за какими компьютерами работает какой-

либо программист, а также какие люди пользуются отдельно взятым компьютером. В рассмотренном примере имеет место *ассоциация* "многие-ко-многим".



В данном случае между экземплярами *классов* `Programmer` и `Computer` в обе стороны используется отношение "0..n", т.к. программист, в принципе, может не работать с компьютером (если он теоретик или на пенсии). В свою очередь, компьютер может никем не использоваться (если он новый и еще не установлен).

Код, соответствующий рассмотренному примеру, будет, например, следующим:

```
public class Programmer {
    private Computer computers[];
    public Programmer() {
    }
}

public class Computer {
    private Programmer programmers[];
    public Computer() {
    }
}
```

Наследование

Наследование является важным случаем отношений между двумя или более *классами*. Подробно оно рассматривалось выше.

Метаклассы

Итак, любой *объект* имеет структуру, состоящую из полей и методов. *Объекты*, имеющие одинаковую структуру и семантику, описываются одним *классом*, который и является, по сути, определением структуры *объектов*, порожденных от него.

В свою очередь, каждый *класс*, или описание, всегда имеет строгий шаблон, задаваемый языком программирования или выбранной объектной моделью. Он определяет, например, допустимо ли множественное *наследование*, какие существуют ограничения на именование *классов*, как описываются поля и методы, набор существующих типов данных и многое другое. Таким образом, *класс* можно рассматривать как *объект*, у которого есть свойства (имя, список полей и их типы, список методов, список аргументов для каждого метода и т.д.). Также *класс* может обладать *поведением*, то есть поддерживать методы. А раз для любого *объекта* существует шаблон, описывающий свойства и *поведение* этого *объекта*, значит, его можно определить и для *класса*. Такой шаблон, задающий различные *классы*, называется *метаклассом*.

Чтобы представить себе, что такое *метакласс*, рассмотрим пример некой бюрократической организации. Будем считать, что все *классы* в такой системе представляют собой строгие инструкции, которые описывают, что нужно сделать, чтобы породить новый *объект* (например, нанять нового служащего или открыть новый отдел). Как и полагается *классам*, они описывают все свойства новых *объектов* (например, зарплату и профессиональный уровень для сотрудников, площадь и имущество для отделов) и их *поведение* (обязанности служащих и функции подразделений).

В свою очередь, написание новой инструкции можно строго регламентировать. Скажем, необходимо использовать специальный бланк, придерживаться правил оформления и заполнить все обязательные поля (например, номер инструкции и фамилии ответственных работников). Такая "инструкция инструкций" и будет представлять собой *метакласс* в ООП.

Итак, *объекты* порождаются от *классов*, а *классы* - от *метакласса*. Он, как правило, в системе только один. Но существуют языки программирования, в которых можно создавать и использовать собственные *метаклассы*, например язык Python. В частности, функциональность *метакласса* может быть следующая: при формировании *класса* он будет просматривать список всех методов в *классе* и, если имя метода имеет вид `set_XXX` или `get_XXX`, автоматически создавать поле с именем `XXX`, если такого не существует.

Поскольку *метакласс* сам является *классом*, то нет никакого смысла в создании "мета-мета-классов".

В языке Java также есть *метакласс*. Это *класс*, который так и называется - `Class` (описывает *классы*), он располагается в основной библиотеке `java.lang`. Виртуальная машина использует его по прямому назначению. Когда загружается очередной `.class`-файл, содержащий описание нового *класса*, JVM порождает *объект класса* `Class`, который будет хранить его структуру. Таким образом, Java использует концепцию *метакласса* в самых практических целях. С помощью `Class` реализована поддержка статических (`static`) полей и методов. Наконец, этот *класс* содержит ряд методов, полезных для разработчиков. Они будут рассмотрены в следующих лекциях.

Достоинства ООП

От любой методики разработки программного обеспечения мы ждем, что она поможет нам в решении наших задач. Но одной из самых значительных проблем проектирования является сложность. Чем больше и сложнее программная система, тем важнее разбить ее на небольшие, четко очерченные части. Чтобы справиться со сложностью, необходимо абстрагироваться от деталей. В этом смысле *классы* представляют собой весьма удобный инструмент.

- *Классы* позволяют проводить конструирование из полезных компонентов, обладающих простыми инструментами, что позволяет абстрагироваться от деталей реализации.
- Данные и операции над ними образуют определенную сущность, и они не разносятся по всей программе, как нередко бывает в случае процедурного программирования, а описываются вместе. Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.
- *Инкапсуляция* позволяет привнести свойство *модульности*, что облегчает распараллеливание выполнения задачи между несколькими исполнителями и обновление версий отдельных компонентов.

ООП дает возможность создавать расширяемые системы. Это одно из основных достоинств ООП, и именно оно отличает данный подход от традиционных методов программирования. Расширяемость означает, что существующую систему можно заставить работать с новыми компонентами, причем без внесения в нее каких-либо изменений. Компоненты могут быть добавлены на этапе исполнения программы.

Полиморфизм оказывается полезным преимущественно в следующих ситуациях.

- Обработка разнородных структур данных. Программы могут работать, не различая вида *объектов*, что существенно упрощает код. Новые виды могут быть добавлены в любой момент.
- Изменение *поведения* во время исполнения. На этапе исполнения один *объект* может быть заменен другим, что позволяет легко, без изменения кода, адаптировать алгоритм в зависимости от того, какой используется *объект*.
- Реализация работы с наследниками. Алгоритмы можно обобщить настолько, что они уже смогут работать более чем с одним видом *объектов*.
- Создание "каркаса" (framework). Независимые от приложения части предметной области могут быть реализованы в виде набора универсальных *классов*, или каркаса (framework), и в дальнейшем расширены за счет добавления частей, специфичных для конкретного приложения.

Часто многоразового использования программного обеспечения не удается добиться из-за того, что существующие компоненты уже не отвечают новым требованиям. *ООП* помогает этого достичь без нарушения работы уже имеющихся клиентов, что позволяет извлечь максимум из многоразового использования компонентов.

- Сокращается время на разработку, которое может быть отдано другим задачам.
- Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.
- Когда некий компонент используется сразу несколькими клиентами, улучшения, вносимые в его код, одновременно оказывают положительное влияние и на множество работающих с ним программ.
- Если программа опирается на стандартные компоненты, ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает использование.

Недостатки ООП

Документирование *классов* - задача более трудная, чем это было в случае процедур и модулей. Поскольку любой метод может быть переопределен, в документации должно говориться не только о том, что делает данный метод, но и о том, в каком контексте он вызывается. Ведь переопределенные методы обычно вызываются не клиентом, а самим каркасом. Таким образом, программист должен знать, какие условия выполняются, когда вызывается данный метод. Для абстрактных методов, которые пусты, в документации должно говориться о том, для каких целей предполагается использовать переопределяемый метод.

В сложных иерархиях *классов* поля и методы обычно наследуются с разных уровней. И не всегда легко определить, какие поля и методы фактически относятся к данному *классу*. Для получения такой информации нужны специальные инструменты, вроде навигаторов *классов*. Если конкретный *класс* расширяется, то каждый метод обычно сокращают перед передачей сообщения базовому *классу*. Реализация операции, таким образом, рассредотачивается по нескольким *классам*, и чтобы понять, как она работает, нам приходится внимательно просматривать весь код.

Методы, как правило, короче процедур, поскольку они осуществляют только одну операцию над данными, зато их намного больше. В коротких методах легче разобраться, но они неудобны тем, что код для обработки сообщения иногда "размазан" по многим маленьким методам.

Инкапсуляцией данных не следует злоупотреблять. Чем больше логики и данных скрыто в недрах *класса*, тем сложнее его расширять. Отправной точкой здесь должно быть не то, что клиентам не разрешается знать о тех или иных данных, а то, что клиентам для работы с *классом* этих данных знать не требуется.

Многие считают, что *ООП* является неэффективным. Как же обстоит дело в действительности? Мы должны проводить четкую грань между неэффективностью на этапе выполнения, неэффективностью в смысле распределения памяти и неэффективностью, связанной с излишней универсализацией.

1. Неэффективность на этапе выполнения. В языках типа Smalltalk сообщения интерпретируются во время выполнения программы путем осуществления их поиска в одной или нескольких таблицах и за счет выбора подходящего метода. Конечно, это медленный процесс. И даже при использовании наилучших методов оптимизации Smalltalk-программы в десять раз медленнее оптимизированных C-программ.

В гибридных языках типа Object Pascal и C++ отправка сообщения приводит лишь к вызову через указатель процедурной переменной. На некоторых машинах сообщения выполняются лишь на 10% медленнее, чем обычные процедурные вызовы. И поскольку сообщения встречаются в программе гораздо реже других операций, их воздействие на время выполнения влияния практически не оказывает.

Однако существует другой фактор, который влияет на время выполнения: это *инкапсуляция* данных. Рекомендуются не предоставлять прямой доступ к полям *класса*, а выполнять каждую операцию над данными через методы. Такая схема приводит к необходимости выполнения процедурного вызова каждый раз при доступе к данным. Однако если *инкапсуляция* используется только там, где она необходима (т.е. в тех случаях, когда это становится преимуществом), то замедление вполне приемлемое.

2. Неэффективность в смысле распределения памяти. Динамическое связывание и проверка типа на этапе выполнения требуют по ходу работы информации о типе *объекта*. Такая информация хранится в дескрипторе типа и он выделяется один на *класс*. Каждый *объект* имеет невидимый указатель на дескриптор типа для своего *класса*. Таким образом, в объектно-ориентированных программах необходимая дополнительная память выражается в одном указателе для *объекта* и в одном дескрипторе типа для *класса*.

3. Излишняя универсальность. Неэффективность также может означать, что в программе реализованы избыточные возможности. В библиотечном *классе* часто содержится больше методов, чем это реально необходимо. А поскольку лишние методы не могут быть удалены, они становятся мертвым грузом. Это не влияет на время выполнения, но сказывается на размере кода.

Одно из возможных решений - строить базовый *класс* с минимальным числом методов, а затем уже реализовывать различные расширения этого *класса*, которые позволят нарастить функциональность. Другой подход - дать компоновщику возможность удалять лишние методы. Такие интеллектуальные компоновщики уже существуют для различных языков и операционных систем.

Но нельзя утверждать, что *ООП* неэффективен. Если *классы* используются лишь там, где это действительно необходимо, то потеря эффективности из-за повышенного расхода памяти и меньшей производительности незначительна. Кроме того, надежность программного обеспечения и быстрота его написания часто бывает важнее, чем производительность.