Идея ООП

Собственно ООП

Для начала разберемся в том, что я собственно имею в виду. Весь мир мыслит объектами. Вы *сидите* настульях, столах, *пьете* кофе из чашек. Вы существуете в мире объектов, живёте среди них. Вы и сами объект. В первую очередь концепция *объектно ориентированного программирования* — это первая попытка человека изобразить программу в том виде, в котором человек видит и понимает окружающий мир. Если мыслить чуть глубже, то это попытка научить комплюхтер думать как человек. Из этого следует приятное свойство *ООП* — оно реально понятно людям. Познакомимся на небольшом примере.

Предположим, мы хотим получать чашки. Самые разнообразные. С точки зрения моей мамы, которая не программист, мы хотим построить завод, который будет по заказу производить разнообразные чашки, кружки и бокалы. С точки зрения меня, как программиста, мы хотим построить **завод**, который будет по заказу *производить* разнообразные чашки, кружки и бокалы.

В своем базисе у чашек, кружек и бокалов лежит идея какого-то абстрактного сосуда, *метод*использования которого — из него *пьют*, а *свойства* которого — он может содержать внутри жидкость и сделан из какого-то материала.

Видите, как мы ловко на простейшем примере уже получаем самые базовые понятия ООП? Для этого мы *абстрагировались* от определенных деталей реализации каждого предмета и получили набор общих свойств и всего такого там. Сам по себе этот факт очень важен. Он говорит нам о том, что объектно ориентированное программирование — это просто. Любой человек, который прочёл этот текст, сможет понять, что у нас есть какой-то сосуд, который мы из чего-то сделали, набрали в него воду и пьём. В технических терминах, у нас будет **абстрактный класс**, например, как в коде внизу. Выбрал PHP как язык с понятным объектным синтаксисом и чтобы показать, что полиморфизм — это не возможность создать сто методов в которые можно передавать разные параметры. С# не выбрал из-за точек, а Java лагает.

**abstract** **class** **Vessel**

{

**private** $material;

**private** $liquid;

**public** **function** **\_\_construct**(Material $material)

{

$this->material = material;

}

**public** **function** **fill**(Liquid $liquid)

{

$this->liquid = $liquid;

}

**public** **function** **pour**()

{

$liquid = $this->liquid;

$this->liquid = **null**;

**return** $liquid;

}

}

**class** **Cup** **extends** **Vessel**

{

*//Чем-то он отличается*

}

**class** **Wineglass** **extends** **Vessel**

{

*//Чем-то он отличается*

}

**class** **Mug** **extends** **Vessel**

{

*//Чем-то он отличается*

}

**class** **Human**

{

*// тут что-то происходит*

**public** **function** **drink**(Vessel $vessel)

{

$this->stomack->add($vessel->pour());

}

}

Дальше — больше. Для того, чтобы начать производить кружки, нам нужен **завод**. Мы делаем заказ заводу и он производит кружки, чашки или бокалы из того материала, который нам нужен, например, изстекла. В программировании всё как в жизни.

**class** **VesselFactory** { *//не самый изящный код, зато наглядненько. $className - это имя класса, который нам нужно получить от фабрики, $material - из чего сделать* **public** **function** **getVessel**($className, $material)

{

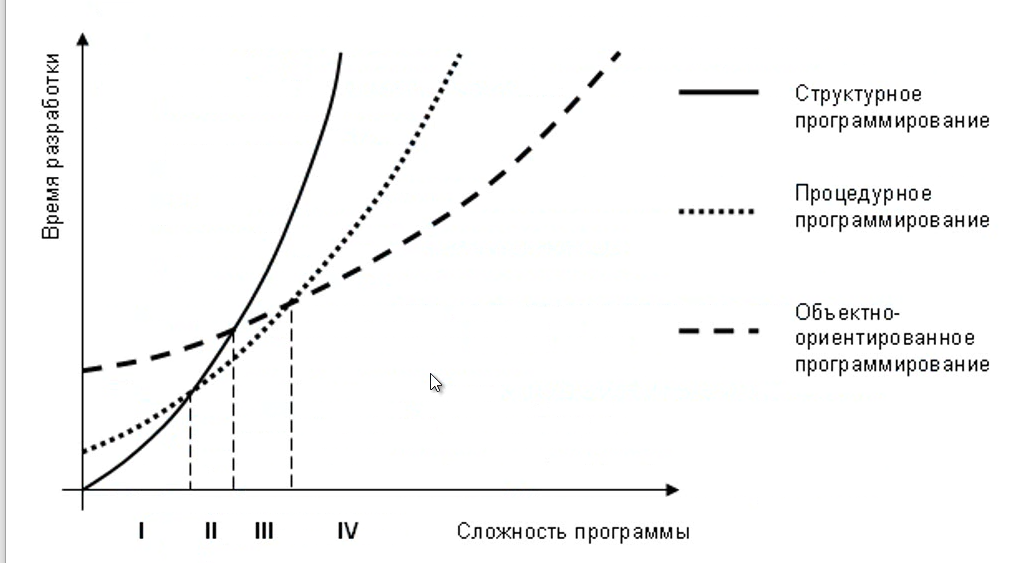
**return** $className($material);

}

}

Лучшая практика — это представить систему так, будто вы не программист, а строитель. Почему паттерны проектирования так хороши? Потому что они как раз таки позволяют абстрагироваться от технической реализации и создавать структуры у себя в голове такими, какими мы бы их делали не в коде, а используя гвозди, молоток и доски. Многие мои коллеги жаловались, что не поняли Банду Четырёх потому что там примеры кода на C++ или Smalltalk. Там идея совсем не в коде, на него вообще можно хер забить, там идея в том, что мы с вами можем строить программу так же, как и дом, и автомобиль. Понятным для человека образом, копируя инженерый гений из абсолютно других сфер. Программист сможет думать как инженер только тогда, когда начнет видеть матрицу вокруг. Декомпозировать предметы, которые его окружают, называть методы/интерфейсы/свойства всего, что попадается на глаза. Начиная от наших любимых кружек и заканчивая любимыми пиратами.

Дров в огонь подбрасывает то, что в современном мире миллионы языков, который либо предоставляют нам кастрированный ООП(привет Javascript, Go), либо технически ориентированный(C#, Java, C++), либо прекрасный и при этом пугающий своими возможностями(Python, Ruby), либо вообще PHP. И это не камень в огород этих языков, потому что у них есть назначение, применение и они хороши и пусть такими и будут.



1. class Shape {
2. public:
3. virtual void draw() const = 0; *// = 0 means it is "pure virtual"*
4. *// ...*
5. };

// Base class

class Shape

{

public:

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// Derived class

class Rectangle: public Shape

{

public:

int getArea()

{

return (width \* height);

}

};

int main(void)

{

Rectangle Rect;

Rect.setWidth(5);

Rect.setHeight(7);

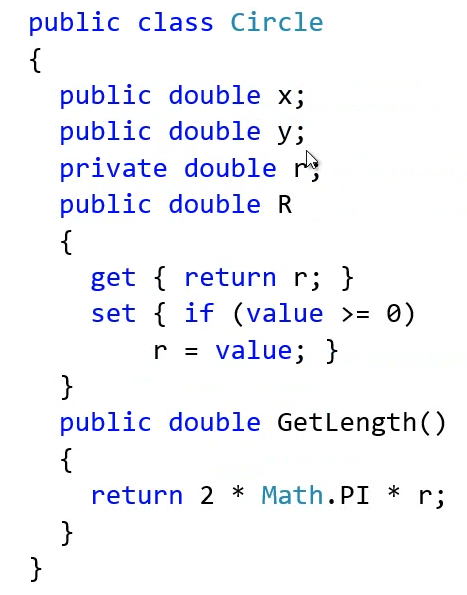
// Print the area of the object.

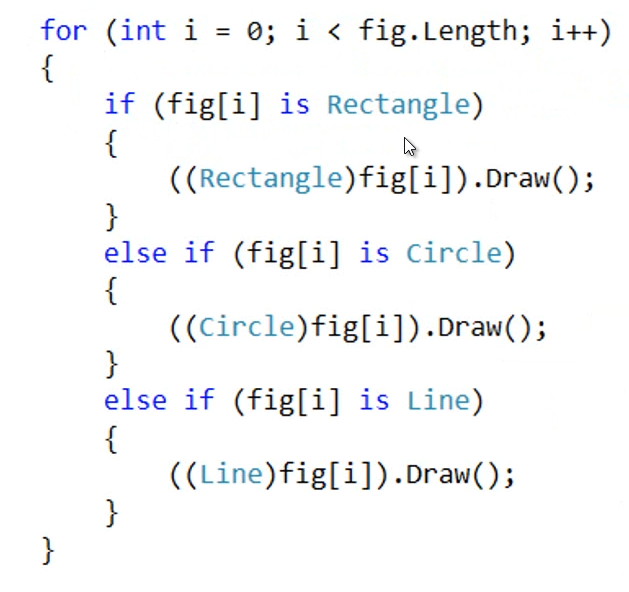
cout << "Total area: " << Rect.getArea() << endl;

return 0;

}

**Абстракция**  
  
Абстракция — это мощнейшее средство программирования. Именно то, что позволяет нам строить большие системы и поддерживать контроль над ними. Вряд ли мы когда-либо подошли бы хотя бы близко к сегодняшнему уровню программ, если бы не были вооружены таким инструментом. Однако как абстракция соотносится с ООП?   
  
Во-первых, абстрагирование не является атрибутом исключительно ООП, да и вообще программирования. Процесс создания уровней абстракции распространяется практически на все области знаний человека. Так, мы можем делать суждения о материалах, не вдаваясь в подробности их молекулярной структуры. Или говорить о предметах, не упоминая материалы, из которых они сделаны. Или рассуждать о сложных механизмах, таких как компьютер, турбина самолёта или человеческое тело, не вспоминая отдельных деталей этих сущностей.   
  
Во-вторых, абстракции в программировании были всегда, начиная с записей Ады Лавлейс, которую принято считать первым в истории программистом. С тех пор люди бесперерывно создавали в своих программах абстракции, зачастую имея для этого лишь простейшие средства. Так, Абельсон и Сассман в своей небезызвестной [книге](http://mitpress.mit.edu/sicp/) описывают, как создать систему решения уравнений с поддержкой комплексных чисел и даже полиномов, имея на вооружении только процедуры и связные списки. Так какие же дополнительные средства абстрагирования несёт в себе ООП? Понятия не имею. Выделение кода в подпрограммы? Это умеет любой высокоуровневый язык. Объединение подпрограмм в одном месте? Для этого достаточно модулей. Типизация? Она была задолго до ООП. Пример с системой решения уравнений хорошо показывает, что построение уровней абстракции не столько зависит от средств языка, сколько от способностей программиста.   
  
**Инкапсуляция**  
  
Главный козырь инкапсуляции в сокрытии реализации. Клиентский код видит только интерфейс, и только на него может рассчитывать. Это развязывает руки разработчикам, которые могут решить изменить реализацию. И это действительно круто. Но вопрос опять же в том, причём тут ООП? Всевышеперечисленные парадигмы подразумевают сокрытие реализации. Программируя на C вы выделяете интерфейс в header-файлы, Oberon позволяет делать поля и методы локальными для модуля, наконец, абстракция во многих языках строится просто посредствам подпрограмм, которые также инкапсулируют реализацию. Более того, объектно-ориентированные языки сами зачастую нарушают правило инкапсуляции, предоставляя доступ к данным через специальные методы — getters и setters в Java, properties в C# и т.д. (В комментариях выяснили, что некоторые объекты в языках программирования не являются объектами с точки зрения ООП: data transfer objects отвечают исключительно за перенос данных, и поэтому не являются полноценными сущностями ООП, и, следовательно, для них нет необходимости сохранять инкапсуляцию. С другой стороны, методы доступа лучше сохранять для поддержания гибкости архитектуры. Вот так всё непросто.) Более того, некоторые объектно-ориентированные языки, такие как Python, вообще не пытаются что-то скрыть, а расчитывают исключительно на разумность разработчиков, использующих этот код.

  
  
**Наследование**  
  
Наследование — это одна из немногих новых вещей, которые действительно вышли на сцену благодаря ООП. Нет, объектно-ориентированные языки не создали новую идею — наследование вполне можно реализовать и в любой другой парадигме — однако ООП впервые вывело эту концепцию на уровень самого языка. Очевидны и плюсы наследования: когда вас почти устраивает какой-то класс, вы можете создать потомка и переопределить какую-то часть его функциональности. В языках, поддерживающих множественное наследование, таких как C++ или Scala (в последней — за счёт traits), появляется ещё один вариант использования — mixins, небольшие классы, позволяющие «примешивать» функциональность к новому классу, не копируя код.   
  
Значит, вот оно — то, что выделяет ООП как парадигму среди других? Хмм… если так, то почему мы так редко используем его в реальном коде? Помните, я говорил про 95% кода, подчиняющихся правилам доминирующей парадигмы? Я ведь не шутил. В функцинальном программировании не меньше 95% кода использует неизменяемые данные и функции без side-эффектов. В модульном практически весь код логично расфасован по модулям. Преверженцы структурного программирования, следуя заветам Дейкстры, стараются разбивать все части программы на небольшие части. Наследование используется гораздо реже. Может быть в 10% кода, может быть в 50%, в отдельных случаях (например, при наследовании от классов фреймворка) — в 70%, но не больше. Потому что в большинстве ситуаций это просто не нужно.   
  
Более того, наследование опасно для хорошего дизайна. Настолько опасно, что Банда Четырех (казалось бы, проповедники ООП) в своей книге рекомендуют при возможности заменять его на делегирование. Наследование в том виде, в котором оно существует в популярных ныне языках ведёт к хрупкому дизайну. Унаследовавшись от одного предка, класс уже не может наследоваться от других. Изменение предка так же становится опасным. Существуют, конечно, модификаторы private/protected, но и они требуют неслабых экстрасенсорных способностей для угадывания, как класс может измениться и как его может использовать клиентский код. Наследование настолько опасно и неудобно, что крупные фреймворки (такие как Spring и EJB в Java) отказываются от них, переходя на другие, не объектно-ориентированные средства (например, метапрограммирование). Последствия настолько непредсказуемы, что некоторые библиотеки (такие как Guava) прописывает своим классам модификаторы, запрещающие наследование, а в новом языке Go было решено вообще отказаться от иерархии наследования.   
  
**Полиморфизм**  
  
Пожалуй, полиморфизм — это лучшее, что есть в объектно-ориентированном программировании. Благодаря полиморфизму объект типа Person при выводе выглядит как «Шандоркин Адам Имполитович», а объект типа Point — как "[84.23 12.61]". Именно он позволяет написать «Mat1 \* Mat2» и получить произведение матриц, аналогично произведению обычных чисел. Без него не получилось бы и считывать данные из входного потока, не заботясь о том, приходят они из сети, файла или строки в памяти. Везде, где есть интерфейсы, подразумевается и полиморфизм.   
  
Мне правда нравится полиморфизм. Поэтому я даже не стану говорить о его [проблемах](http://en.wikipedia.org/wiki/Double_dispatch) в мейнстримовых языках. Я также промолчу про узость подхода диспетчеризации только по типу, и про то, как это [могло бы быть сделано](http://constarg.wordpress.com/2011/01/06/clojure-multimethods/). В большинстве случаев он работает как надо, а это уже неплохо. Вопрос в другом: является ли полиморфизм тем самым принципом, отличающим ООП от других парадигм? Если бы вы спросили меня (а раз уж вы читаете этот текст, значит, можно считать, что спросили), я бы ответил «нет». И причина всё в тех же процентах использования в коде. Возможно, интерфейсы и полиморфные методы встречаются немного чаще наследования. Но сравните количество строк кода, занимаемое ими, с количеством строк, написанных в обычном процедурном стиле — последних всегда больше. Глядя на языки, поощряющие такой стиль программирования, я не могу назвать их полиморфными. Языки с поддержкой полиморфизма — да, так нормально. Но не полиморфные языки.



**Принципы объектно-ориентированного программирования**

Все объектно-ориентированные языки используют три базовых принципа объектно-ориентированного программирования.

• Инкапсуляция. Как данный язык скрывает внутренние особенности реализации объекта?

• Наследование. Как данный язык обеспечивает возможность многократного использования программного кода?

• Полиморфизм. Как данный язык позволяет интерпретировать родственные объекты унифицированным образом?

Перед тем как начать рассмотрение синтаксических особенностей реализации каждого из этих принципов, важно понять базовую роль каждого из них. Поэтому здесь предлагается краткая теоретическая информация по соответствующим вопросам, просто для того, чтобы исключить разночтения, которые могут возникать у людей, ощущающих постоянную нехватку времени на теорию из-за жестких сроков окончания их проектов.

**Инкапсуляция**

Первым принципом ООП является инкапсуляция. По сути, она означает возможность скрыть средствами языка несущественные детали реализации от пользователя объекта. Предположим, например, что мы используем класс DatabaseReader, который имеет два метода Open() и Close().

**// DatabaseReader инкапсулирует средства работы с базой данных.**

DatabaseReader dbObj = new DatabaseReader();

dbObj.Open(@"C:\Employees.mdf");

// **Работа с базой данных.**..

dbObj.Close();

Вымышленный класс DatabaseReader инкапсулирует внутренние возможности размещения, загрузки, обработки и закрытия файла данных. Пользователи объекта приветствуют инкапсуляцию, поскольку этот принцип ООП позволяет упростить задачи программирования. Нет необходимости беспокоиться о многочисленных строках программного кода, который выполняет работу класса DatabaseReader "за кулисами". Bсe, что требуется от вас, – это создание экземпляра и отправка подходящих сообщений (например, "открыть файл Employees.mdf, размещенный на моем диске C").

Одним из аспектов инкапсуляции является защита данных. В идеале данные состояния объекта должны определяться, как приватные, a не открытые (как было в предыдущих главах). В этом случае "внешний мир" будет вынужден "смиренно просить" право на изменение или чтение соответствующих значений.

**Наследование**

Следующим принципом ООП является наследование, означающее способность языка обеспечить построение определений новых классов на основе определений существующих классов. В сущности, наследование позволяет расширить возможности поведения базового класса (называемого также родительским классом) с помощью построения подкласса (называемого производным классам или дочерним классом), наследующего функциональные возможности родительского класса. На рис. 4.3 иллюстрируется отношение подчиненности ("is-а") для родительских и дочерних классов.

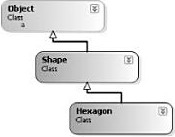


Рис. 4.3. Отношение подчиненности для родительских и дочерних классов

Можно прочитать эту диаграмму так: "Шестиугольник (hexagon) является формой (shape), которая является объектом (object)". При создании классов, связанных этой формой наследования, вы создаете отношения подчиненности между типами. Отношение подчиненности часто называется классическим наследованием.

Вспомните из главы 3, что System.Object является предельным базовым классом любой иерархии .NET. Здесь класс Shape (форма) расширяет Object (объект). Можно предположить, что Shape определяет некоторый набор свойств, полей, методов и событий, которые будут общими для всех форм. Класс Hexagon (шестиугольник) расширяет Shape и наследует функциональные возможности, определенные в рамках Shape и Object, вдобавок к определению своих собственных членов (какими бы они ни были).

В мире ООП есть и другая форма многократного использования программного кода - это модель локализации/делегирования (также известная, как отношение локализации, "has-a"). Эта форма многократного использования программного кода не используется дли создания отношений "класс-подкласс". Скорее данный класс может определить член-переменную другого класса и открыть часть или все свои функциональные возможности для "внешнего мира".

Например, если создается модель автомобиля, то вы можете отобразить тот факт, что автомобиль "имеет" ("has-a") радио. Было бы нелогично пытаться получить класс Car (автомобиль) из Radio (радио) или наоборот. (Радио является автомобилем? Я думаю, нет.) Скорее, есть два независимых класса, работающие вместе, где класс-контейнер создает и представляет функциональные возможности содержащегося в нем класса.

public class Radio {

 public void Power(bool turnOn) { Console.WriteLine("Radio on: {0}", turnOn); }

}

public class Car {

 // **Car содержит ("has-a") Radio.**

 private Radio myRadio = new Radio();

 public void TurnOnRadio(bool onOff) {

**// Делегат для внутреннего объекта.**

  myRadio.Power(onOff);

 }

}

Здесь тип-контейнер (Car) несет ответственность за создание содержащегося объекта (Radio). Если объект Car "желает" сделать поведение Radio доступным для экземпляра Car, он должен пополнить свой собственный открытый интерфейс некоторым набором функций, Которые будут действовать на содержащийся тип. Заметим, что пользователь объекта не получит никакой информации о том, что класс Car использует внутренний объект Radio.

static void Main(string[] args) {

 // **Вызов внутренне передается Radio.**

 Car viper = new Car();

 viper.TurnOnRadio(true);

}

**Полиморфизм**

Третьим принципом ООП является полиморфизм. Он характеризует способность языка одинаково интерпретировать родственные объекты. Эта особенность объектно-ориентированного языка позволяет базовому классу определить множество членов (формально называемых полиморфным интерфейсом) для всех производных классов. Полиморфный интерфейс типа класса строится с помощью определения произвольного числа виртуальных, или абстрактныхчленов. Виртуальный член можно изменить (или, говоря более формально, переопределить) в производном классе, тогда как абстрактный метод должен переопределяться. Когда производные типы переопределяют члены, определенные базовым классом, они, по существу, переопределяют свой ответ на соответствующий запрос.

Чтобы проиллюстрировать понятие полиморфизма, снова используем иерархию форм. Предположим, что класс Shape определил метод Draw(), не имеющий параметров и не возвращающий ничего. С учетом того, что визуализация для каждой формы оказывается уникальной, подклассы (такие как Hexagon и Circle) могут переопределить соответствующий метод так, как это требуется для них (рис. 4.4).

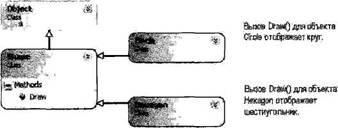


Рис. 4.4. Классический полиморфизм

После создания полиморфного интерфейса можно использовать различные предположения, касающиеся программного кода. Например, если Hexagon и Circle являются производными от одного общего родителя (Shape), то некоторый массив типов Shape может содержать любой производный класс. Более того, если Shape определяет полиморфный интерфейс для всех производных типов {в данном примере это метод Draw(), то можно предположить, что каждый член в таком массиве имеет эти функциональные возможности. Проанализируйте следующий метод Main(), в котором массиву типов, производных от Shape, дается указание визуализировать себя с помощью метода Draw().

static void Main(string [] args) {

**// Создание массива элементов, производных от Shape.**

 Shape[] myShapes = new Shape[3];

 myShapes[0] = new Hexagon();

 myShapes[1] = new Circle();

 myShapes[2] = new Hexagon();

 // **Движение по массиву и отображение элементов.**

 foreach (Shape s in myShapes) s.Draw();

 Console.ReadLine();

}