Муниципальное бюджетное общеобразовательное учреждение

г. Подольск “Средняя школа № 30”

Проект на тему:

*«***Объектно ориентированное программирование***»*

Выполнил:

Ученик 11В класса

Сухих Даниил Алексеевич

Руководитель проекта:

Иванова Наталья Юрьевна

Г.Подольск

2021 г

Содержание индивидуального проекта:

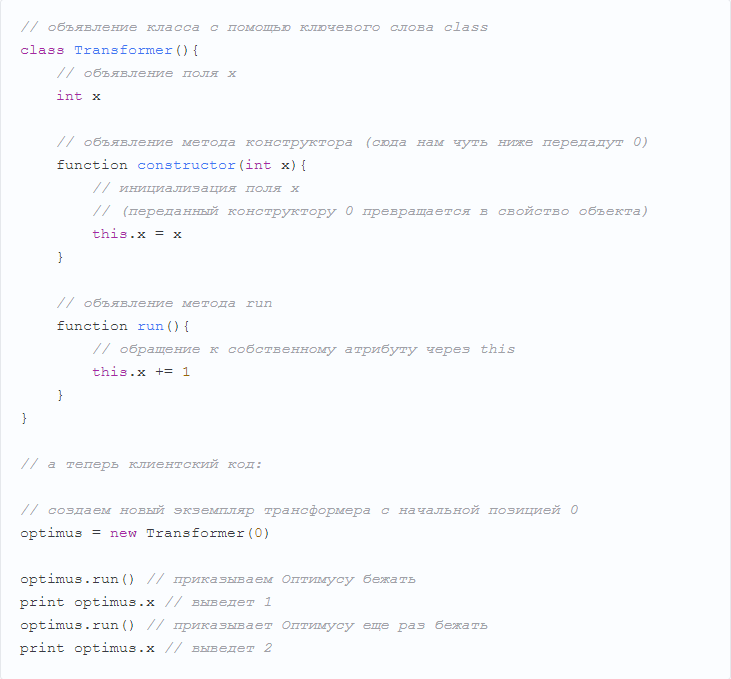
1. Введение в тему ООП
   1. Что такое ООП?
   2. Смысл парадигмы ООП.
   3. Надобность ООП в проектах различных типов.
2. Классы и объекты
   1. Что такое классы и объекты?
   2. Немного терминологии
   3. Листенинг кодово и иллюстрации
3. SRP
4. Ассоциация
5. Основные принципы ООП
   1. Наследование
      1. Перегрузка
      2. Неуместное применение
      3. Наследование статично
      4. Множественное наследование
   2. Полиморфизм
   3. Инкапсуляция
6. Абстрактные классы
7. Интерфейсы
8. Утиная типизация
9. ISP
10. Заключение
11. Источники

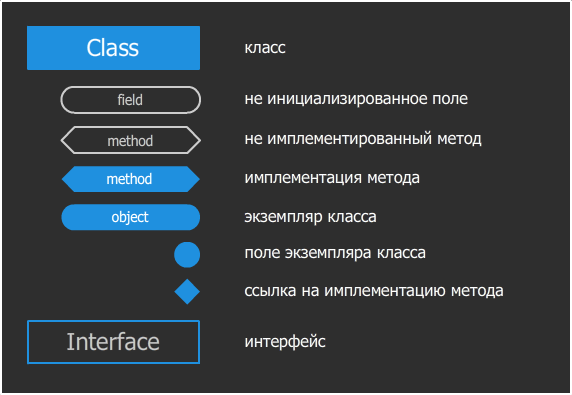
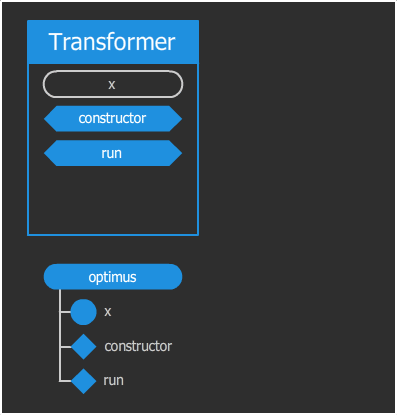
**Введение.**

ООП (Объектно-Ориентированное Программирование) стало неотъемлемой частью разработки многих современных проектов, но, не смотря на популярность, эта парадигма является далеко не единственной. Если вы уже умеете работать с другими парадигмами и хотели бы ознакомиться с оккультизмом ООП, то впереди вас ждет немного лонгрид и два мегабайта картинок и анимаций. В качестве примеров будут выступать трансформеры.  
  
  
  
Прежде всего стоит ответить, зачем? Объектно-ориентированная идеология разрабатывалась как попытка связать поведение сущности с её данными и спроецировать объекты реального мира и бизнес-процессов в программный код. Задумывалось, что такой код проще читать и понимать человеком, т. к. людям свойственно воспринимать окружающий мир как множество взаимодействующих между собой объектов, поддающихся определенной классификации. Удалось ли идеологам достичь цели, однозначно ответить сложно, но де-факто мы имеем массу проектов, в которых с программиста будут требовать ООП.  
  
Не следует думать, что ООП каким-то чудным образом ускорит написание программ, и ожидать ситуацию, когда жители Вилларибо уже выкатили ООП-проект в работу, а жители Виллабаджо все еще отмывают жирный спагетти-код. В большинстве случаев это не так, и время экономится не на стадии разработки, а на этапах поддержки (расширение, модификация, отладка и тестирование), то бишь в долгосрочной перспективе. Если вам требуется написать одноразовый скрипт, который не нуждается в последующей поддержке, то и ООП в этой задаче, вероятнее всего, не пригодится. Однако, значительную часть жизненного цикла большинства современных проектов составляют именно поддержка и расширение. Само по себе наличие ООП не делает вашу архитектуру безупречной, и может наоборот привести к излишним усложнениям.  
  
Иногда можно столкнуться с критикой в адрес быстродействия ООП-программ. Это правда, незначительный оверхед присутствует, но настолько незначительный, что в большинстве случаев им можно пренебречь в пользу преимуществ. Тем не менее, в узких местах, где в одном потоке должны создаваться или обрабатываться миллионы объектов в секунду, стоит как минимум пересмотреть необходимость ООП, ибо даже минимальный оверхед в таких количествах может ощутимо повлиять на производительность. Профилирование поможет вам зафиксировать разницу и принять решение. В остальных же случаях, скажем, где львиная доля быстродействия упирается в IO, отказ от объектов будет преждевременной оптимизацией.  
  
В силу своей природы, объектно-ориентированное программирование лучше всего объяснять на примерах. Как и обещал, нашими пациентами будут трансформеры. Я не трансформеролог, и комиксов не читал, посему в примерах буду руководствоваться википедией и фантазией.

**Классы и объекты**

Сразу лирическое отступление: объектно-ориентированный подход возможен и без классов, но мы будем рассматривать, извиняюсь за каламбур, классическую схему, где классы — наше всё.  
  
Самое простое объяснение: класс — это чертеж трансформера, а экземпляры этого класса — конкретные трансформеры, например, Оптимус Прайм или Олег. И хотя они и собраны по одному чертежу, умеют одинаково ходить, трансформироваться и стрелять, они оба обладают собственным уникальным состоянием. Состояние — это ряд меняющихся свойств. Поэтому у двух разных объектов одного класса мы можем наблюдать разное имя, возраст, местоположение, уровень заряда, количество боеприпасов и т. д. Само наличие этих свойств и их типы описываются в классе.  
  
Таким образом, класс — это описание того, какими свойствами и поведением будет обладать объект. А объект — это экземпляр с собственным состоянием этих свойств.  
  
Мы говорим «свойства и поведение», но звучит это как-то абстрактно и непонятно. Привычнее для программиста будет звучать так: «переменные и функции». На самом деле «свойства» — это такие же обычные переменные, просто они являются атрибутами какого-то объекта (их называют полями объекта). Аналогично «поведение» — это функции объекта (их называют методами), которые тоже являются атрибутами объекта. Разница между методом объекта и обычной функцией лишь в том, что метод имеет доступ к собственному состоянию через поля.  
  
Итого, имеем методы и свойства, которые являются атрибутами. Как работать с атрибутами? В большинстве ЯП оператор обращения к атрибуту — это точка (кроме PHP и Perl). Выглядит это примерно вот так (псевдокод):



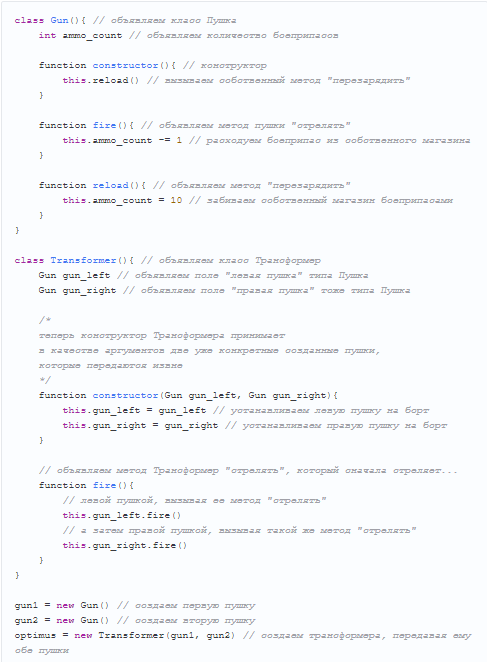
В картинках я буду использовать такие обозначения:  
  
  
  
Я не стал использовать UML-диаграммы, посчитав их недостаточно наглядными, хоть и более гибкими.  
  
  
  
Что мы видим из кода?  
  
1. this — это специальная локальная переменная (внутри методов), которая позволяет объекту обращаться из своих методов к собственным атрибутам. Обращаю внимание, что только к собственным, то бишь, когда трансформер вызывает свой метод, либо меняет собственное состояние. Если снаружи обращение будет выглядеть так: optimus.x, то изнутри, если Оптимус захочет сам обратиться к своему полю x, в его методе обращение будет звучать так: this.x, то есть "*я (Оптимус) обращаюсь к своему атрибуту x*". В большинстве языков эта переменная называется this, но встречаются и исключения (например, self)  
  
2. constructor — это специальный метод, который автоматически вызывается при создании объекта. Конструктор может принимать любые аргументы, как и любой другой метод. В каждом языке конструктор обозначается своим именем. Где-то это специально зарезервированные имена типа \_\_construct или \_\_init\_\_, а где-то имя конструктора должно совпадать с именем класса. Назначение конструкторов — произвести первоначальную инициализацию объекта, заполнить нужные поля.  
  
3. new — это ключевое слово, которое необходимо использовать для создания нового экземпляра какого-либо класса. В этот момент создается объект и вызывается конструктор. В нашем примере, конструктору передается 0 в качестве стартовой позиции трансформера (это и есть вышеупомянутая инициализация). Ключевое слово new в некоторых языках отсутствует, и конструктор вызывается автоматически при попытке вызвать класс как функцию, например так: Transformer().  
  
4. Методы constructor и run работают с внутренним состоянием, а во всем остальном не отличаются от обычных функций. Даже синтаксис объявления совпадает.  
  
5. Классы могут обладать методами, которым не нужно состояние и, как следствие, создание объекта. В этом случае метод делают статическим.

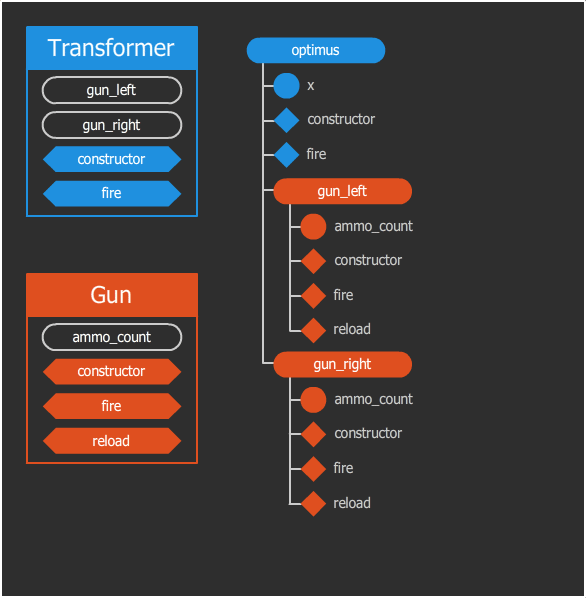
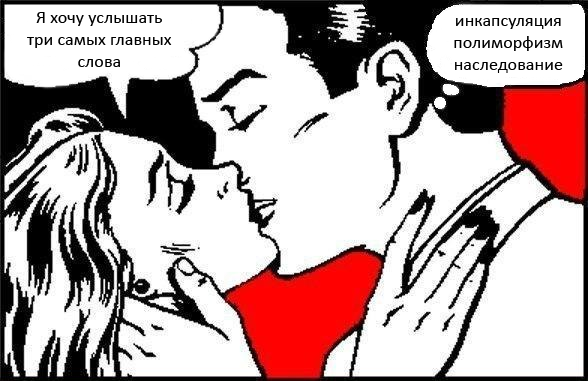
### **SRP**

(Single Responsibility Principle / Принцип единственной ответственности / Первый принцип [SOLID](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))). С ним вы, наверняка, уже знакомы из других парадигм: «одна функция должна выполнять только одно законченное действие». Этот принцип справедлив и для классов: «Один класс должен отвечать за какую-то одну задачу». К сожалению с классами сложнее определить грань, которую нужно пересечь, чтобы принцип нарушался.  
  
Существуют попытки формализовать данный принцип с помощью описания назначения класса одним предложением без союзов, но это очень спорная методика, поэтому доверьтесь своей интуиции и не бросайтесь в крайности. Не нужно делать из класса швейцарский нож, но и плодить миллион классов с одним методом внутри — тоже глупо.

## Ассоциация

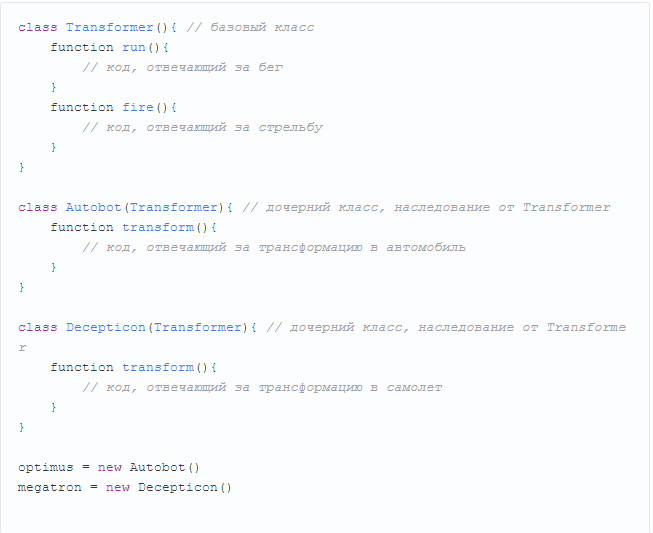
Традиционно в полях объекта могут храниться не только обычные переменные стандартных типов, но и другие объекты. А эти объекты могут в свою очередь хранить какие-то другие объекты и так далее, образуя дерево (иногда граф) объектов. Это отношение называется ассоциацией.  
  
Предположим, что наш трансформер оборудован пушкой. Хотя нет, лучше двумя пушками. В каждой руке. Пушки одинаковые (принадлежат к одному классу, или, если будет угодно, выполненные по одному чертежу), обе одинаково умеют стрелять и перезаряжаться, но в каждой есть свое хранилище боеприпасов (собственное состояние). Как теперь это описать в ООП? С помощью ассоциации:



  
  
this.gun\_left.fire() и this.gun\_right.fire() — это обращения к дочерним объектам, которые происходят так же через точки. По первой точке мы обращаемся к атрибуту себя (this.gun\_right), получая объект пушки, а по второй точке обращаемся к методу объекта пушки (this.gun\_right.fire()).  
  
Итог: робота сделали, табельное оружие выдали, теперь разберемся, что тут происходит. В данном коде один объект стал составной частью другого объекта. Это и есть ассоциация. Она в свою очередь бывает двух видов:  
  
1. Композиция — случай, когда на фабрике трансформеров, собирая Оптимуса, обе пушки ему намертво приколачивают к рукам гвоздями, и после смерти Оптимуса, пушки умирают вместе с ним. Другими словами, жизненный цикл дочернего объекта совпадает с жизненным циклом родительского.  
  
2. Агрегация — случай, когда пушка выдается как пистолет в руку, и после смерти Оптимуса этот пистолет может подобрать его боевой товарищ Олег, а затем взять в свою руку, либо сдать в ломбард. То бишь жизненный цикл дочернего объекта не зависит от жизненного цикла родительского, и может использоваться другими объектами.  
  
Ортодоксальная ООП-церковь проповедует нам фундаментальную троицу — инкапсуляцию, полиморфизм и наследование, на которых зиждется весь объектно-ориентированный подход. Разберем их по порядку.  
  


**Наследование**

Наследование — это механизм системы, который позволяет, как бы парадоксально это не звучало, наследовать одними классами свойства и поведение других классов для дальнейшего расширения или модификации.  
  
Что если, мы не хотим штамповать одинаковых трансформеров, а хотим сделать общий каркас, но с разным обвесом? ООП позволяет нам такую шалость путем разделения логики на сходства и различия с последующим выносом сходств в родительский класс, а различий в классы-потомки. Как это выглядит?  
  
Оптимус Прайм и Мегатрон — оба трансформеры, но один является автоботом, а второй десептиконом. Допустим, что различия между автоботами и десептиконами будут заключаться только в том, что автоботы трансформируются в автомобили, а десептиконы — в авиацию. Все остальные свойства и поведение не будут иметь никакой разницы. В таком случае можно спроектировать систему наследования так: общие черты (бег, стрельба) будут описаны в базовом классе «Трансформер», а различия (трансформация) в двух дочерних классах «Автобот» и «Десептикон».



### **Перегрузка**

Если же в классе-потомке переопределить уже существующий метод в классе-родителе, то сработает перегрузка. Это позволяет не дополнять поведение родительского класса, а модифицировать. В момент вызова метода или обращения к полю объекта, поиск атрибута происходит от потомка к самому корню — родителю. То есть, если у автобота вызвать метод fire(), сначала поиск метода производится в классе-потомке — Autobot, а поскольку его там нет, поиск поднимается на ступень выше — в класс Transformer, где и будет обнаружен и вызван. Следует отметить, что модификация нарушает [LSP](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%BF%D0%BE%D0%B4%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B8_%D0%91%D0%B0%D1%80%D0%B1%D0%B0%D1%80%D1%8B_%D0%9B%D0%B8%D1%81%D0%BA%D0%BE%D0%B2) из набора принципов [SOLID](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), но мы рассматриваем только техническую возможность.

### **Неуместное применение**

Любопытно, что чрезмерно глубокая иерархия наследования может привести к обратному эффекту — усложнению при попытке разобраться, кто от кого наследуется, и какой метод в каком случае вызывается. К тому же, не все архитектурные требования можно реализовать с помощью наследования. Поэтому применять наследование следует без фанатизма. Существуют рекомендации, призывающие предпочитать композицию наследованию там, где это уместно. Любая критика наследования, которую я встречал, подкрепляется неудачными примерами, когда наследование используется в качестве [золотого молотка](https://ru.wikipedia.org/wiki/%D0%97%D0%BE%D0%BB%D0%BE%D1%82%D0%BE%D0%B9_%D0%BC%D0%BE%D0%BB%D0%BE%D1%82%D0%BE%D0%BA). Но это совершенно не означает, что наследование в принципе всегда вредит. Мой нарколог говорил, что первый шаг — это признать, что у тебя зависимость от наследования.  
  
Как при описании отношений двух сущностей определить, когда уместно наследование, а когда — композиция? Можно воспользоваться популярной шпаргалкой: спросите себя, сущность А является сущностью Б? Если да, то скорее всего, тут подойдет наследование. Если же сущность А является частью сущности Б, то наш выбор — композиция.  
  
Применительно к нашей ситуации это будет звучать так:

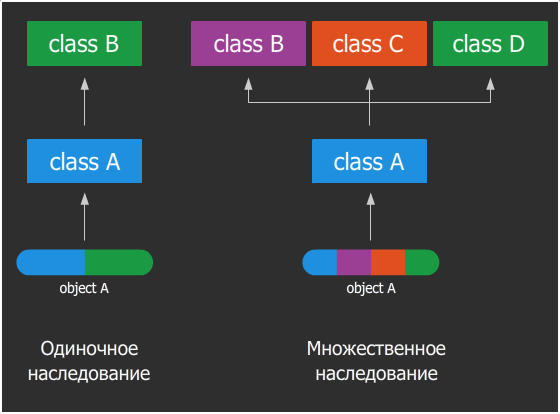
1. Автобот является Трансформером? Да, значит выбираем наследование.
2. Пушка является частью Трансформера? Да, значит — композиция.

Для самопроверки попробуйте обратную комбинацию, получится фигня. Эта шпаргалка помогает в большинстве случаев, но бывают и другие факторы, на которые стоит опираться при выборе между композицией и наследованием. Кроме того, эти методы можно комбинировать для решения разного типа задач.

### **Наследование статично**

Еще одно важное отличие наследования от композиции в том, что наследование имеет статическую природу и устанавливает отношения классов только на этапе интерпретации/компиляции. Композиция же, как мы видели в примерах, позволяет менять отношение сущностей на лету прямо в рантайме — иногда это очень важно, поэтому об этом нужно помнить при выборе отношений (если конечно нет желания использовать [метапрограммирование](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%B0%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

### **Множественное наследование**

Мы рассмотрели ситуацию, когда два класса унаследованы от общего потомка. Но в некоторых языках можно сделать и наоборот — унаследовать один класс от двух и более родителей, объединив их свойства и поведение. Возможность наследоваться от нескольких классов вместо одного — это множественное наследование.  
  
  
  
Вообще, в кругах иллюминатов бытует мнение, что множественное наследование — это грех, оно несет за собой [ромбовидную проблему](https://ru.wikipedia.org/wiki/%D0%A0%D0%BE%D0%BC%D0%B1%D0%BE%D0%B2%D0%B8%D0%B4%D0%BD%D0%BE%D0%B5_%D0%BD%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) и неразбериху с конструкторами. Кроме того, задачи, которые решаются множественным наследованием, можно решать другими механизмами, например, механизмом интерфейсов (о котором мы тоже поговорим). Но справедливости ради, следует отметить, что множественное наследование удобно использовать для реализации [примесей](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BC%D0%B5%D1%81%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

### **Абстрактные классы**

Кроме обычных классов в некоторых языках существуют абстрактные классы. От обычных классов они отличаются тем, что нельзя создать объект такого класса. Зачем же нужен такой класс, спросит читатель? Он нужен для того, чтобы от него могли наследоваться потомки — обычные классы, объекты которых уже можно создавать.  
  
Абстрактный класс наряду с обычными методами содержит в себе абстрактные методы без имплементации (с сигнатурой, но без кода), которые обязан имплементировать программист, задумавший создать класс-потомок. Абстрактные классы не обязательны, но они помогают установить контракт, обязующий имплементировать определенный набор методов, дабы уберечь программиста с плохой памятью от ошибки имплементации.

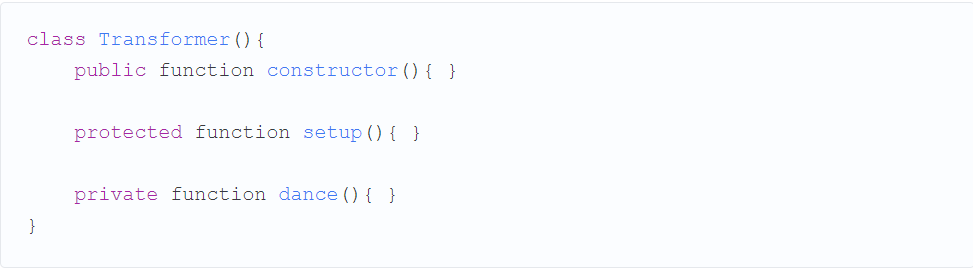
## Полиморфизм

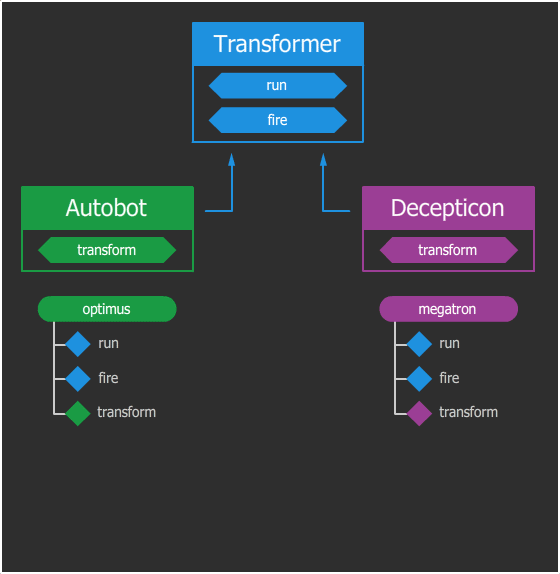
Полиморфизм — свойство системы, позволяющее иметь множество реализаций одного интерфейса. Ничего непонятно. Обратимся к трансформерам.  
  
Положим, у нас есть три трансформера: Оптимус, Мегатрон и Олег. Трансформеры боевые, стало быть обладают методом attack(). Игрок, нажимая у себя на джойстике кнопку «воевать», сообщает игре, чтобы та вызвала метод attack() у трансформера, за которого играет игрок. Но поскольку трансформеры разные, а игра интересная, каждый из них будет атаковать каким-то своим способом. Скажем, Оптимус — объект класса Автобот, а Автоботы снабжаются пушками с плутониевыми боеголовками (да не прогневаются фанаты трансформеров). Мегатрон — Десептикон, и стреляет из плазменной пушки. Олег — басист, и он обзывается. А в чем польза?  
  
Польза полиморфизма в данном примере заключается в том, что код игры ничего не знает о реализации его просьбы, кто как должен атаковать, его задача просто вызвать метод attack(), сигнатура которого одинакова для всех классов персонажей. Это позволяет добавлять новые классы персонажей, или менять методы существующих, не меняя код игры. Это удобно.

## Инкапсуляция

Инкапсуляция — это контроль доступа к полям и методам объекта. Под контролем доступа подразумевается не только можно/неможно, но и различные валидации, подгрузки, вычисления и прочее динамическое поведение.  
  
Во многих языках частью инкапсуляции является сокрытие данных. Для этого существуют модификаторы доступа (опишем те, которые есть почти во всех ООП языках):

* publiс — к атрибуту может получить доступ любой желающий
* private — к атрибуту могут обращаться только методы данного класса
* protected — то же, что и private, только доступ получают и наследники класса в том числе



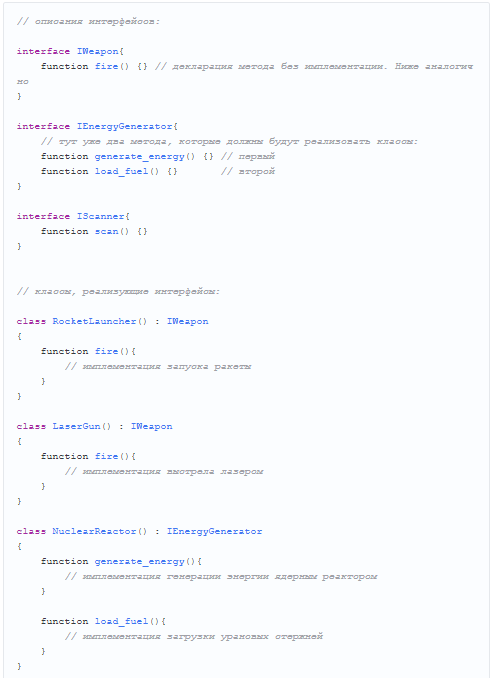
  
  
Сей пример наглядно иллюстрирует, как наследование становится одним из способов дедуплицировать код ([DRY-принцип](https://ru.wikipedia.org/wiki/Don%E2%80%99t_repeat_yourself)) с помощью родительского класса, и одновременно предоставляет возможности для мутации в классах-потомках.

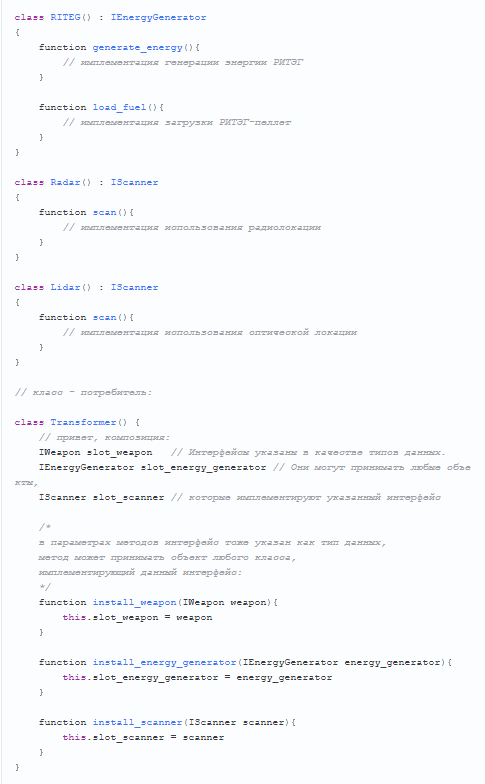
## Интерфейсы

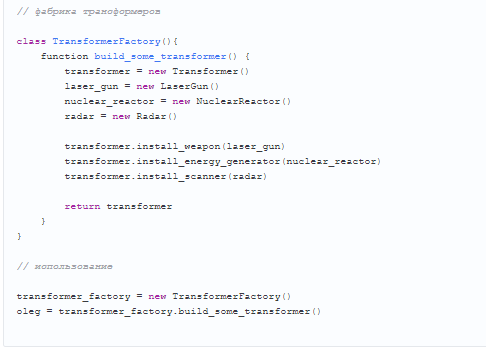
Задача интерфейса — снизить уровень зависимости сущностей друг от друга, добавив больше абстракции.  
  
Не во всех языках присутствует этот механизм, но в ООП языках со статической типизацией без них было бы совсем худо. Выше мы рассматривали абстрактные классы, затрагивая тему контрактов, обязующих имплементировать какие-то абстрактные методы. Так вот интерфейс очень смахивает на абстрактный класс, но является не классом, а просто пустышкой с перечислением абстрактных методов (без имплементации). Другими словами, интерфейс имеет декларативную природу, то есть, чистый контракт без капельки кода.  
  
Обычно в языках, в которых есть интерфейсы, нет множественного наследования классов, но есть множественное наследование интерфейсов. Это позволяет классу перечислить интерфейсы, которые он обязуется имплементировать.  
  
Классы с интерфейсами состоят в отношении «многие ко многим»: один класс может имплементировать множество интерфейсов, и каждый интерфейс, в свою очередь, может имплементироваться многими классами.  
  
У интерфейса двустороннее применение:

1. По одну сторону интерфейса — классы, имплементирующие данный интерфейс.
2. По другую сторону — потребители, которые используют этот интерфейс в качестве описания типа данных, с которым они (потребители) работают.

Например, если какой-то объект помимо основного поведения, может быть сериализован, то пускай он имплементирует интерфейс «Сериализуемый». А если объект можно склонировать, то пусть он имплементирует еще один интерфейс — «Клонируемый». И если у нас есть какой-то транспортный модуль, который передает объекты по сети, он будет принимать любые объекты, имплементирующие интерфейс «Сериализуемый».  
  
Представим, что каркас трансформера оборудован тремя слотами: слот для оружия, для генератора энергии и для какого-нибудь сканера. Эти слоты обладают определенными интерфейсами: в каждый слот можно установить только подходящее оборудование. В слот для оружия можно установить ракетную установку или лазерную пушку, в слот для генератора энергии — ядерный реактор или РИТЭГ (радиоизотопный термоэлектрический генератор), а в слот для сканера — радар или лидар. Суть в том, что каждый слот имеет универсальный интерфейс подключения, а уже конкретные устройства должны соответствовать этому интерфейсу. К примеру, на материнских платах используется несколько типов слотов: слот для процессора позволяет подключать различные процессоры, подходящие под данный сокет, а слот SATA — любой SSD или HDD накопитель или даже CD/DVD.  
  
Обращаю внимание, что получившаяся система слотов у трансформеров — это пример использования композиции. Если же оборудование в слотах будет сменным в ходе жизни трансформера, то тогда это уже агрегация. Для наглядности, мы будем называть интерфейсы, как принято в некоторых языках, добавляя заглавную «И» перед именем: IWeapon, IEnergyGenerator, IScanner.







### **Утиная типизация**

Явление, которое мы наблюдаем в получившейся архитектуре, называется утиной типизацией: *если что-то крякает как утка, плавает как утка, и выглядит как утка, то, скорее всего — это утка*.  
  
Переводя это на язык трансформеров, звучать будет так: если что-то стреляет как пушка, и перезаряжается как пушка, скорее всего это пушка. Если устройство генерирует энергию, скорее всего это генератор энергии.  
  
В отличие от иерархической типизации наследования, при утиной типизации трансформеру пофиг, какого класса пушку ему дали, и пушка ли это вообще. Главное, что эта штуковина умеет стрелять! Это не достоинство утиной типизации, а скорее компромисс. Может быть и обратная ситуация, как на этой картинке ниже:

### **ISP**

(Interface Segregation Principle / Принцип разделения интерфейса / Четвертый принцип SOLID) призывает не создавать жирные универсальные интерфейсы. Вместо этого интерфейсы нужно разделять на более мелкие и специализированные, это поможет гибче их комбинировать в имплементирующих классах, не заставляя имплементировать лишние методы.

**Заключение**

В условиях современных требований наличие в вашем коде слова class не делает из вас ООП-программиста. Ибо если вы не используете описанные в статье механизмы (полиморфизм, композицию, наследование и т. д.), а вместо этого применяете классы лишь для группировки функций и данных, то это не ООП. То же самое можно решить какими-нибудь неймспейсами и структурами данных. Не путайте, иначе на собеседовании будет стыдно.  
  
Хочется закончить свою песнь важными словами. Любые описанные механизмы, принципы и паттерны, как и ООП в целом не стоит применять там, где это бессмысленно или может навредить. Это ведет к появлению статей со странными заголовками типа «Наследование — причина преждевременного старения» или «Синглтон может приводить к онкологическим заболеваниям».  
  
Я серьезно. Если рассмотреть случай с синглтоном, то его повсеместное применение без знания дела, стало причиной серьезных архитектурных проблем во многих проектах. И любители забивать гвозди микроскопом любезно его нарекли антипаттерном. Будьте благоразумны.  
  
К сожалению, в проектировании не существует однозначных рецептов на все случаи жизни, где что применять уместно, а где неуместно. Это будет постепенно укладываться в голове с опытом.

**Источники:**

<https://habr.com/ru/post/463125/>

PHP 7 в подлиннике. Дмитрий Котеров