Лекция 7

# Объектно-ориентированное программирование. Часть 3

# Как правильно использовать ООП ?

# Как правильно использовать ООП ?

**Хорошее ООП**

- снижает сложность
- привносит модульность
- повышает совместимость
- локализует изменения
- абстрагирует от реализации
- упрощает код
- легко тестируется

**Плохое ООП**

- привносит сложность
- сохраняет процедурный подход
- причиняет неудобство

# Принципы DRY и SOLID

## DRY (Don't Repeat Yourself)

Выявляйте одинаковые программные сущности и объединяйте их.

## SOLID

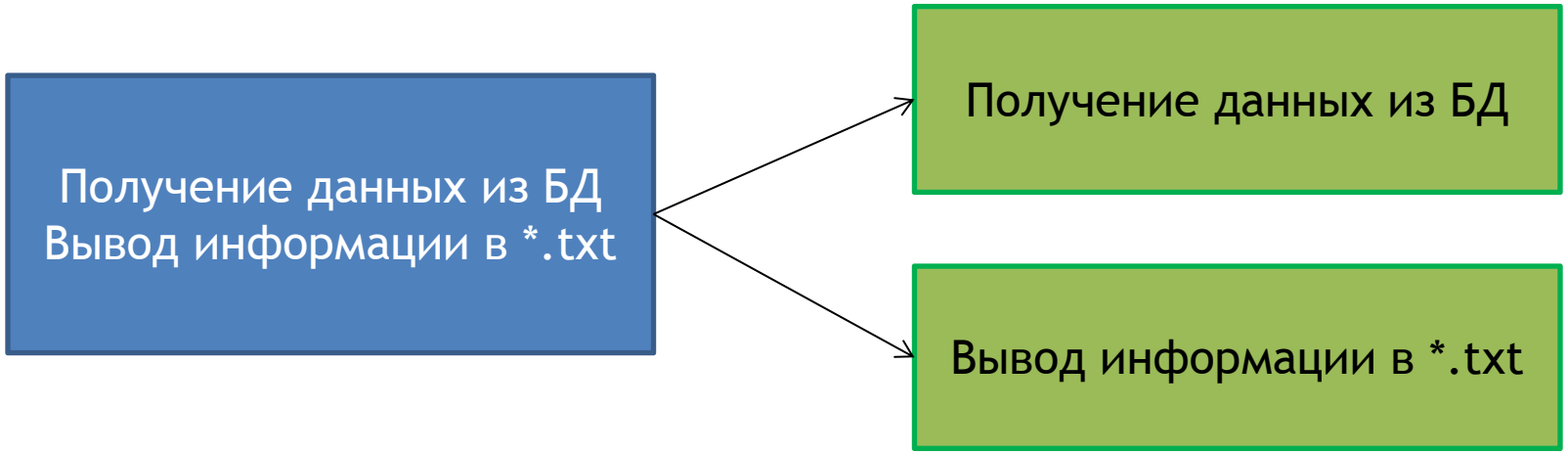
| Инициал | Представляет[1] | Название[4], понятие |
|---|---|---|
| **S** | SRP[5] | **Принцип единственной ответственности** (***The Single Responsibility Principle***)<br>Каждый класс должен иметь одну и только одну причину для изменений. |
| **O** | OCP[6] | **Принцип открытости/закрытости** (***The Open Closed Principle***)<br>«программные сущности … должны быть открыты для расширения, но закрыты для модификации.» |
| **L** | LSP[7] | **Принцип подстановки Барбары Лисков** (***The Liskov Substitution Principle***)<br>«объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы.» См. также контрактное программирование.<br>Наследующий класс должен дополнять, а не изменять базовый. |
| **I** | ISP[8] | **Принцип разделения интерфейса** (***The Interface Segregation Principle***)<br>«много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения.»[9] |
| **D** | DIP[10] | **Принцип инверсии зависимостей** (***The Dependency Inversion Principle***)<br>«Зависимость на Абстракциях. Нет зависимости на что-то конкретное.»[9] |

# Single Responsibility Principle

У программной сущности должна быть только одна причина для изменений.

# Single Responsibility Principle

У программной сущности должна быть только одна причина для изменений.



# The Open-Closed Principle

Программные сущности должны быть открыты для расширения (*наследование, полиморфизм, композиция*) и закрыты для модификации (*инкапсуляция*).

# The Open-Closed Principle

Программные сущности должны быть открыты для расширения (*наследование, полиморфизм, композиция*) и закрыты для модификации (*инкапсуляция*).

# Liskov Substitution Principle

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.
Наследующий класс должен дополнять, а не изменять родительский.

# Interface Segregation Principle

Программные сущности не должны зависеть от частей интерфейса, которые они не используют (и знать о них тоже не должны).

# The Dependency Inversion Principle

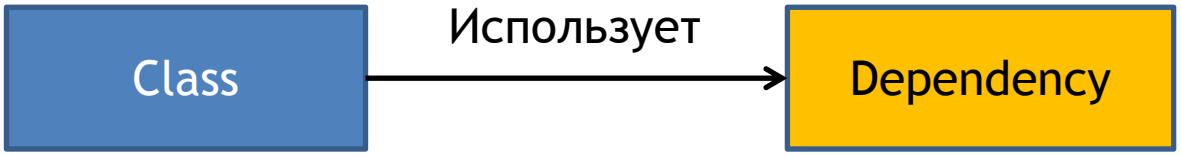
Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.

Модули верхних уровней содержат бизнес-логику и интерфейс
Модули нижних уровней содержат технические детали (например, обращение к БД или к стороннему сервису).

Под "не должны зависеть" подразумеваются детали. Т.е. модуль бизнес-логики естественно использует модуль доступа к БД, но при этом в него не должны просачиваться детали имплементации (например, что это за БД и какой индекс надо использовать для выполнения запроса)

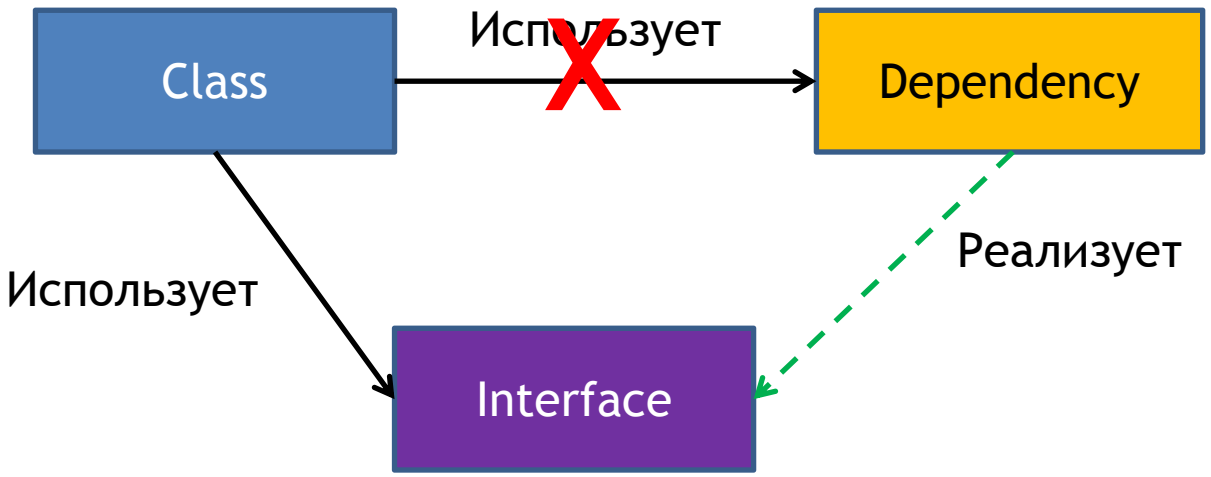
# The Dependency Inversion Principle

Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.



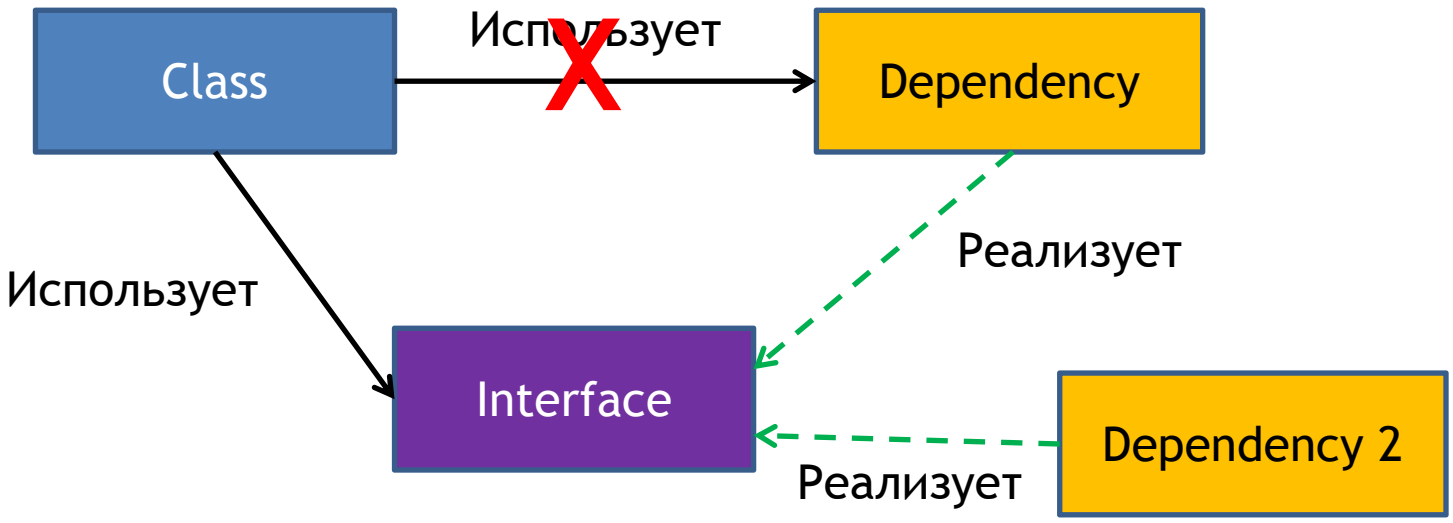
# The Dependency Inversion Principle

Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.



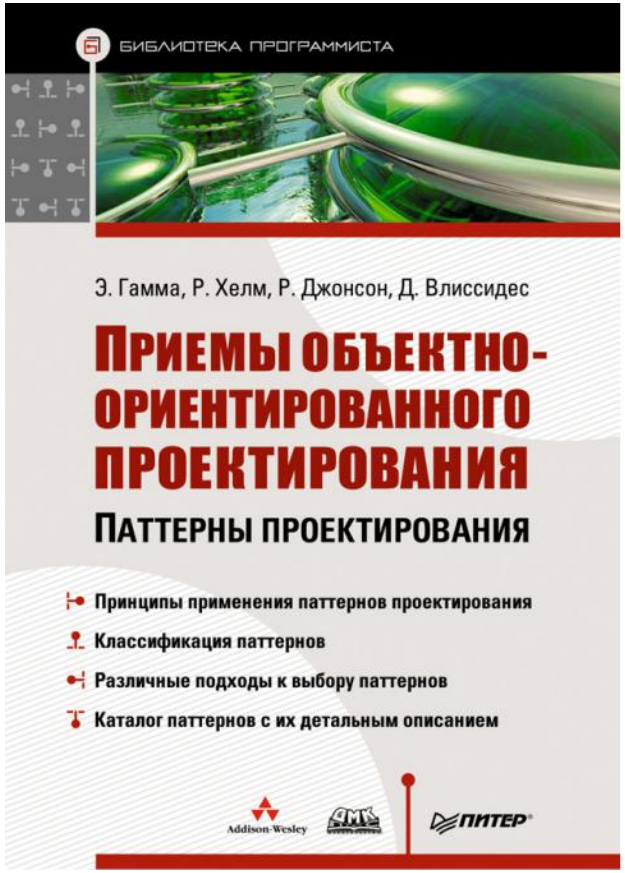
# The Dependency Inversion Principle

Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.



# Паттерны (шаблоны) проектирования


Christofer Alexander. «A Pattern Language. Towns. Buildings. Constructions» 1977




Erich Gamma, Richard Helm, Ralph Johnson,John Vlissides «Design Patterns. Elements of Reusable Object-Oriented Softwart»
1995

**Зачем знать паттерны?**
Шаблоны упрощают проектирование и поддержку программ.

- **Проверенные решения.**

Ваш код более предсказуем когда вы используете готовые решения, вместо повторного изобретения велосипеда.

- **Стандартизация кода.**

Использование типовых унифицированных решений — должно быть меньше ошибок.

- **Общий язык.**

По названию шаблона понятно, какой подход вы придумали и какие классы для этого нужны.

# Классификация шаблонов

- **Порождающие шаблоны** — для гибкого создания объектов без внесения в программу лишних зависимостей.

- **Структурные шаблоны** — показывают различные способы построения связей между объектами.

- **Поведенческие шаблоны** — для эффективной коммуникации между объектами.

**Порождающие шаблоны**

- Factory method
- Abstract factory
- Singleton
- ...

**Структурные шаблоны**

- Adapter
- Decorator
- ...

**Поведенческие шаблоны**

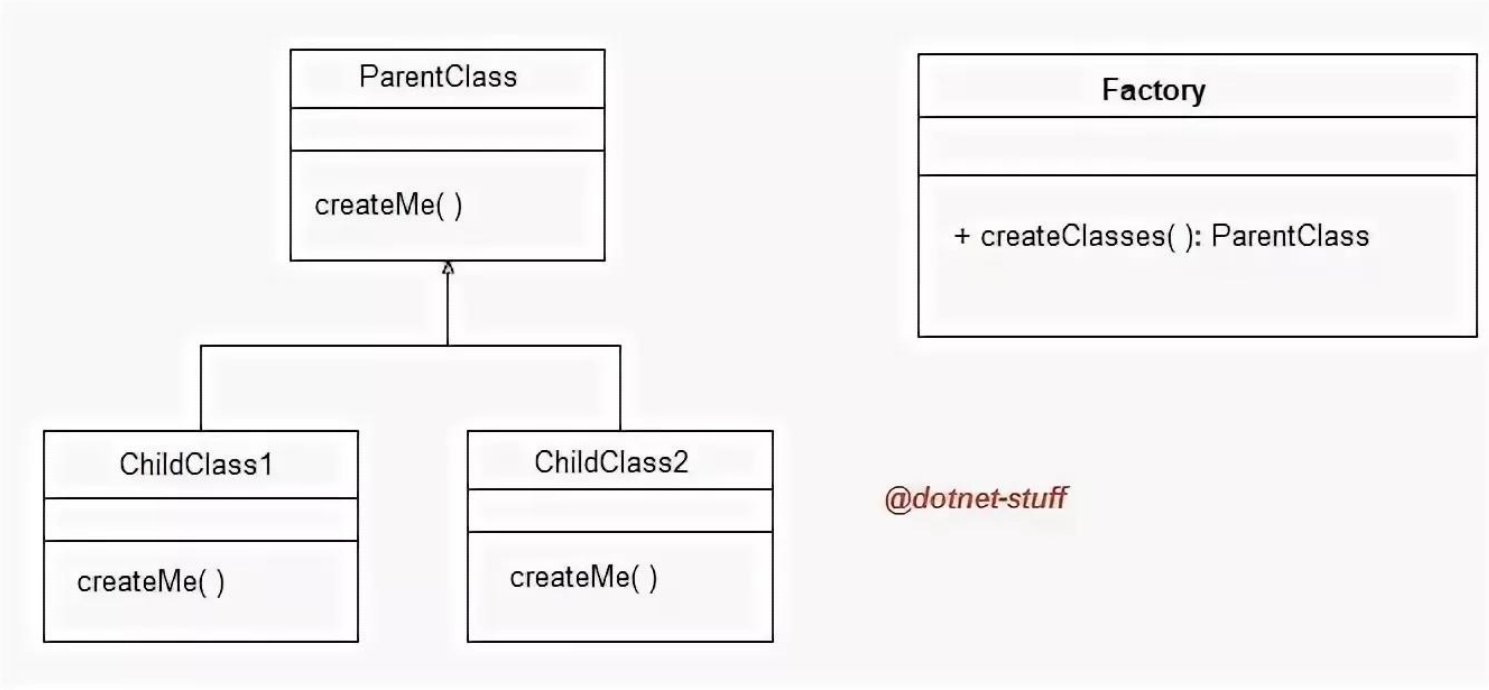
- Observer
- Strategy

# Simple Factory

Фабрика — объект, создающий другие объекты.
Простая фабрика генерирует экземпляр класса для клиента.

**Когда использовать?**

Когда создание объекта подразумевает какую-то логику, а не просто несколько присваиваний, то имеет смысл делегировать задачу выделенной фабрике, а не повторять повсюду один и тот же код.

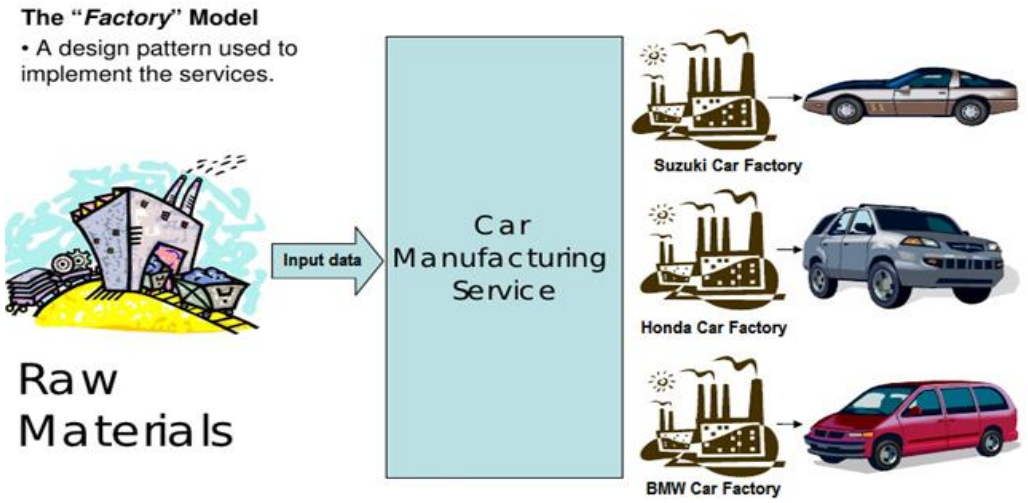


# Factory method

Способ делегирования логики создания объектов дочерним классам.

**Когда использовать?**

- Когда заранее неизвестно, объекты каких типов необходимо создавать;
- Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать;
- Когда создание новых объектов необходимо делегировать из базового класса классам наследникам.



# Builder

Позволяет создавать разные свойства объекта, избегая загрязнения конструктора.
Это полезно, когда у объекта может быть несколько свойств. Или когда создание объекта состоит из большого количества этапов.

**Когда использовать?**

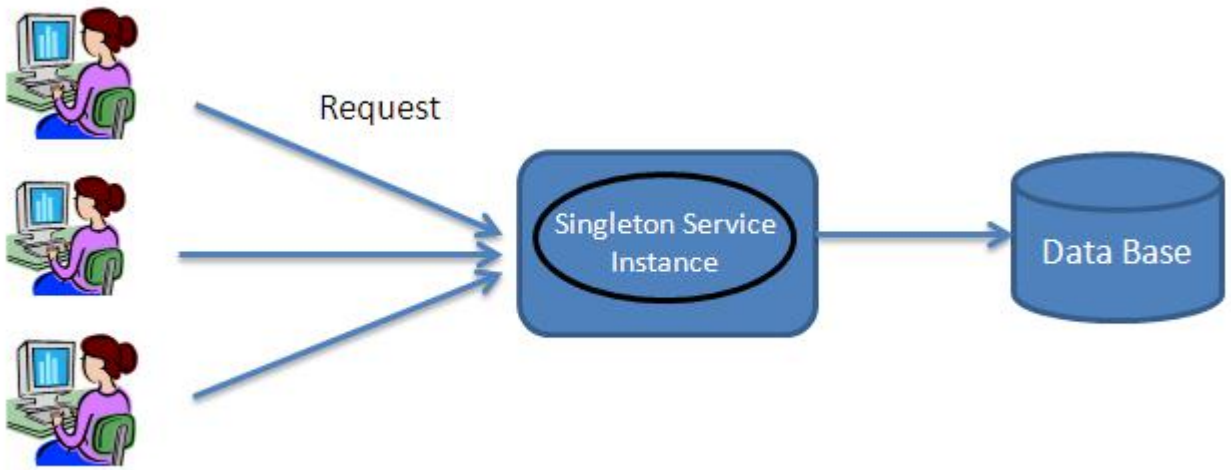
- Когда у объекта много свойств;
- Когда создание объекта состоит из большого количества этапов.

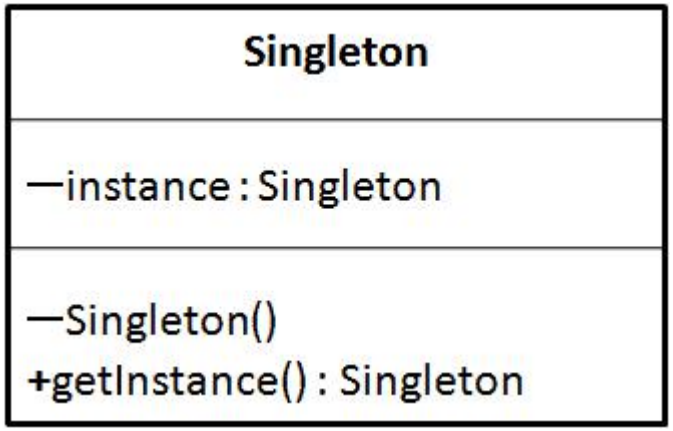
Шаблон Builder решает проблему антишаблона Telescoping constructor:

```
public function __construct($size, $cheese = true, $pepperoni = true, $tomato = false, $lettuce = true)
{
}
```

# Singleton

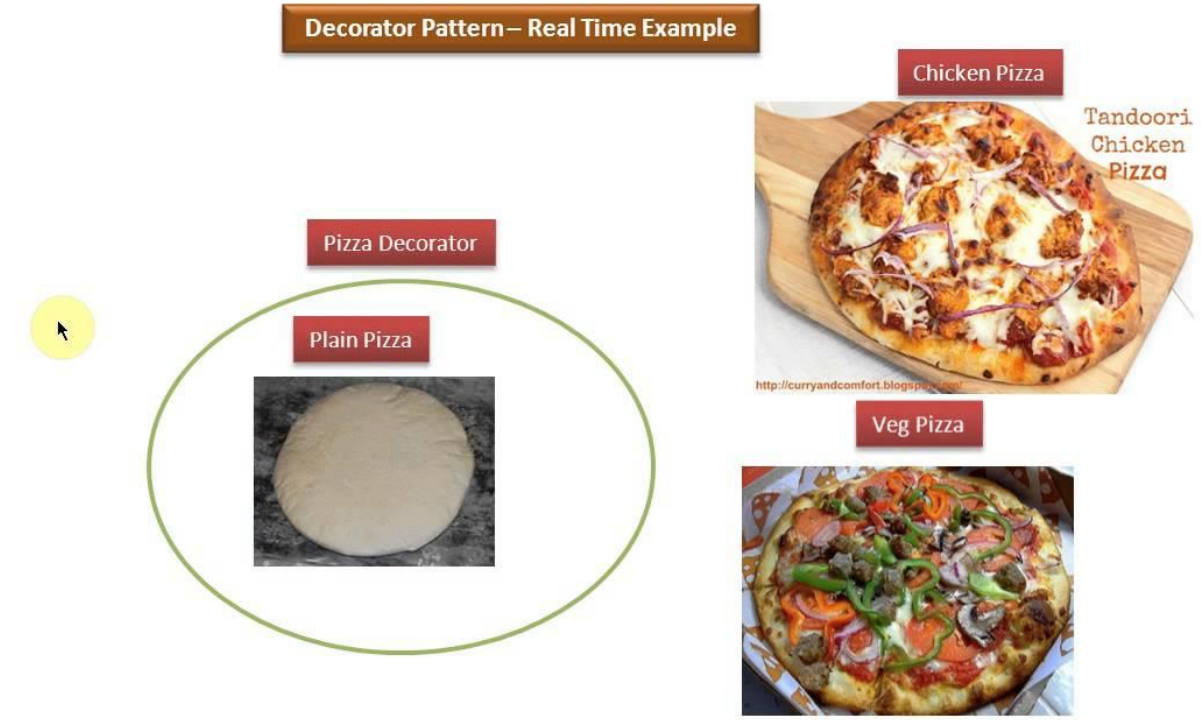
Гарантирует, что класс имеет только один экземпляр и предоставляет глобальную точку доступа к нему.



# Decorator

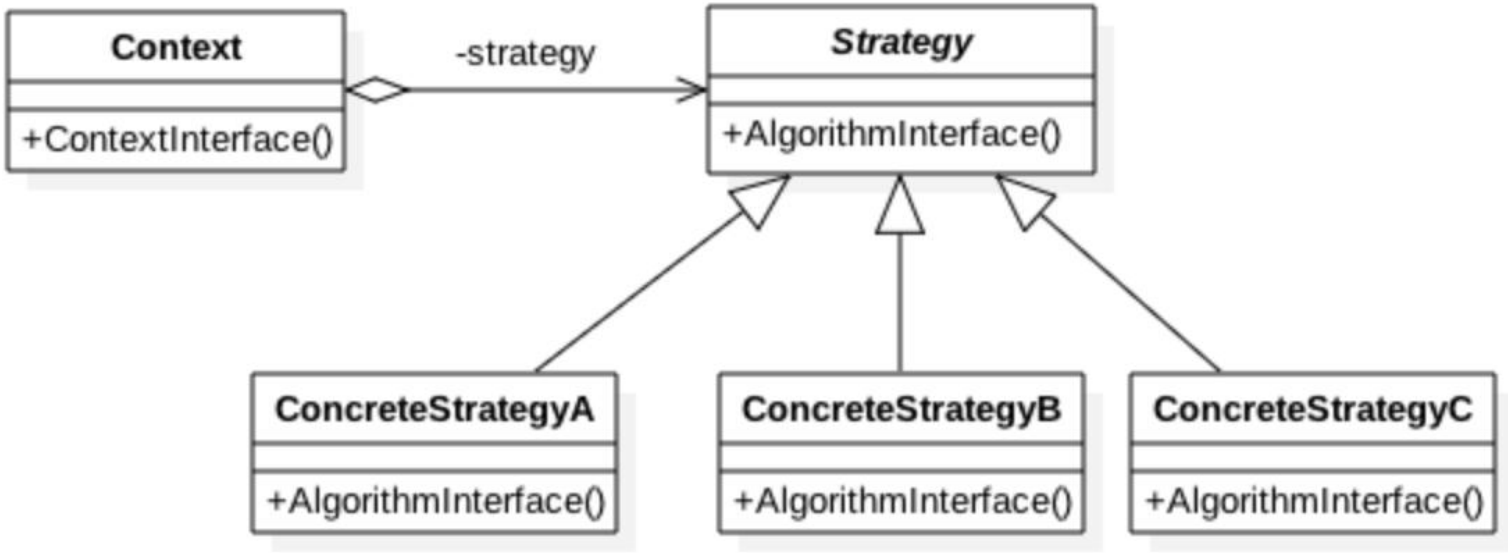
Позволяет подключать к объекту дополнительное поведение (статически или динамически), не влияя на поведение других объектов того же класса.
Является гибкой альтернативой порождению подклассов с целью расширения функциональности.



# Strategy

Позволяет переключаться между алгоритмами или стратегиями в зависимости от ситуации (независимо от клиентов, их использующих).

Альтернатива наследованию (вместо расширения абстрактного класса).

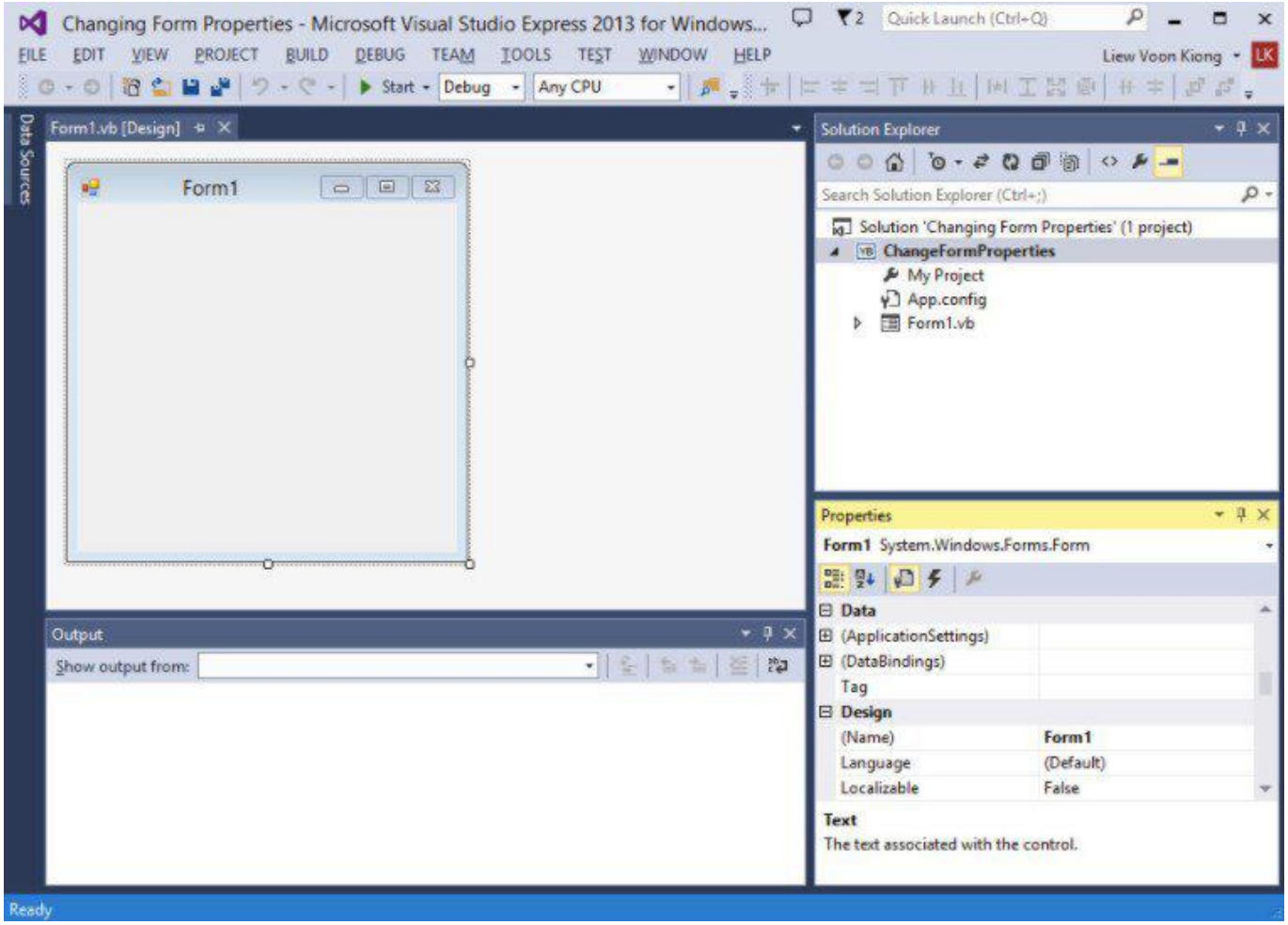


# Не усложняйте!

- **KISS (Keep It Simple, Stupid).** Пишите проще.
- **YAGNI (You Aren't Gonna Need It).** Вам это не понадобится.
- **Магическое число 7, плюс или минус 2.** Максимальная длина функций - 7 строк. Отступы углубляются не более, чем на 5 уровней. Классы с не более, чем 5 методами. В шаблонах проектирования от 5 до 9 классов.

# Визуальное проектирование приложений

Используются визуальные объекты-компоненты.
Результат – приложение с автоматически генерируемым кодом.



# Проблема повторного использования объектов

Я создал/нашел полезный объект на языке программирования X.
Как воспользоваться им из программы на языке Y?

# Проблема повторного использования объектов

Я создал/нашел полезный объект на языке программирования X.
Как воспользоваться им из программы на языке Y?

Связи между объектами-модулями должны быть **стандартными** и **не зависящими от языка программирования и платформы**, на которой работает программа.

# Стандартизация

Все камни разные – строить стены тяжело

# Стандартизация

Все камни разные – строить стены тяжело

1. На всей земле был один язык и одно наречие.
2. Двинувшись с востока, они нашли в земле Сеннаар равнину и поселились там.
3. И сказали друг другу: наделаем кирпичей и обожжем огнем. **И стали у них кирпичи вместо камней, а земляная смола вместо извести.**
(Книга Бытия 11:1-3)

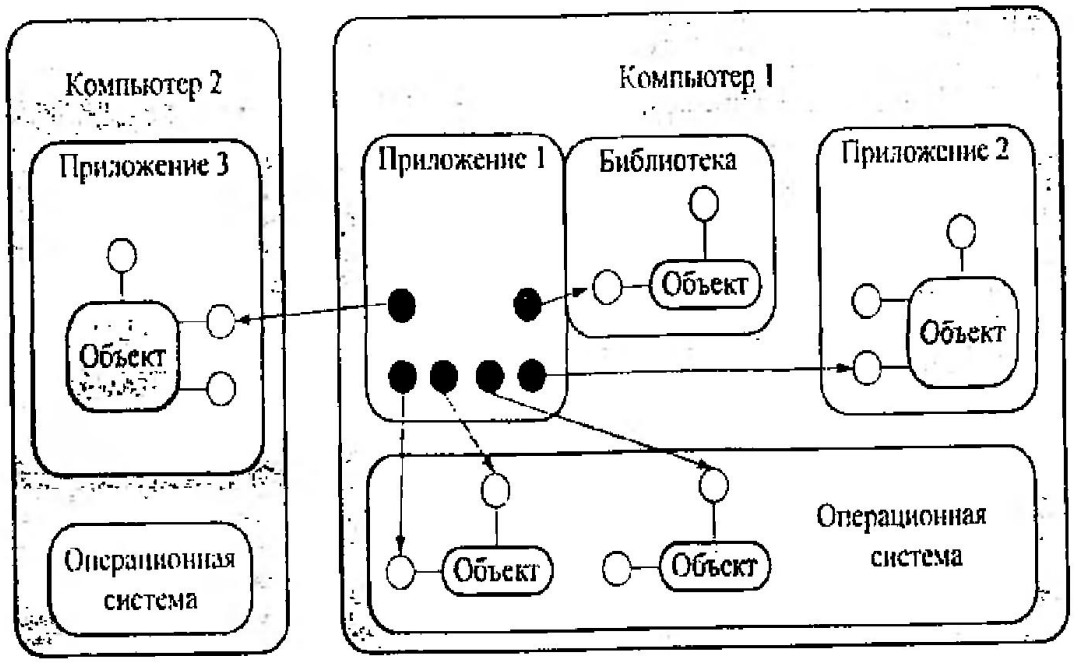
# Компонентный подход к программированию

Построение программ из физически отдельных компонентов, которые взаимодействуют между собой через стандартизированные интерфейсы:

- **Двоичные.** Технологии COM, ActiveX, CORBA.
- **Текстовые.** Протокол SOAP, объекты JSON.

Объекты-компоненты можно распространять без исходных кодов и использовать в любом языке программирования.

# Компонентный подход к программированию




Использование COM-объектов, зарегистрированных в Windows.

```
Set WA = CreateObject("Word.Application")
Set Shell = CreateObject("WScript.Shell")
Set FSO = CreateObject("Scripting.FileSystemObject")
```