CSCC01 Summer 2017

Week 09 - Design Principles. More Design Patterns.

Ilir Dema

University of Toronto Scarborough

July 10, 2017

The factory pattern

Consider this code fragment

```
Duck duck;

if (picnic) {
duck = new MallardDuck();
} else if (hunting) {
duck = new DecoyDuck();
} else if (inBathTub) {
duck = new RubberDuck();
```

Attention!

Many instantiations makes updates and hence maintenance difficult and error-prone

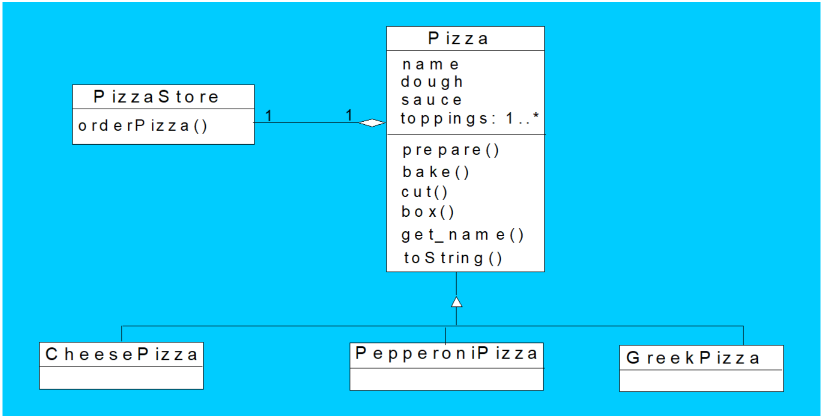
Instantiating concrete classes

- Using new instantiates a concrete class.
- This is programming to implementation instead of interface.
- Concrete classes are often instantiated in more than one place.
- Thus, when changes or extensions are made all the instantiations will have to be changed.
- Such extensions can result in updates being more difficult and error-prone.

Example

Suppose that you are required to develop a system that accepts orders for pizzas. There are three types of pizzas, namely, cheese, Greek, and pepperoni. The pizzas differ according to the dough used, the sauce used and the toppings.
Draw a class diagram for the system.

EXAMPLE: CLASS DIAGRAM



Clearly, we have a design problem here!

Example

What is wrong with new?

- `new` operator is fundamental in Java
- The problem how change impacts `new`.
- When we have code that makes use of lots of concrete classes, that code may have changes as new concrete classes are added.
- So, the code is not *closed for modification*.

Identifying aspects that may vary

Consider this code fragment

```
Pizza orderPizza() {
  Pizza pizza = new Pizza();
  pizza.prepare();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza;
}
```

Attention!

For flexibility we want Pizza to be abstract class or interface but we cannot instantiate those.

Identifying aspects that may vary

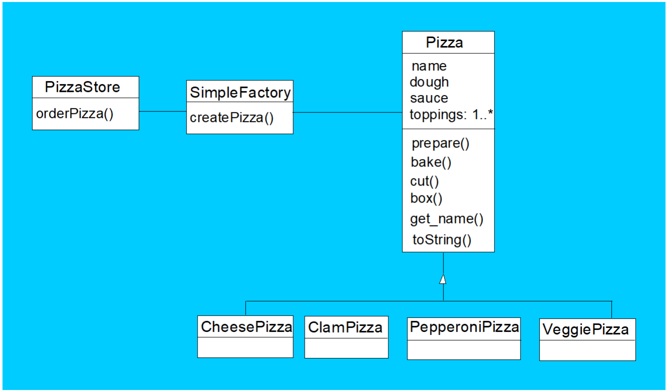
Consider this code fragment

```
Pizza orderPizza(String type) {
  Pizza pizza;
  if(type.equals(``cheese'')) {
    pizza = new CheesePizza();
    // add code for each type of pizza ...
    // this is the part that changes
 }
  // the part below does not change ...
  pizza.prepare();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza;
}
```

Example: Revised class diagram

What should be done?

Encapsulate object creation!



Fix the code

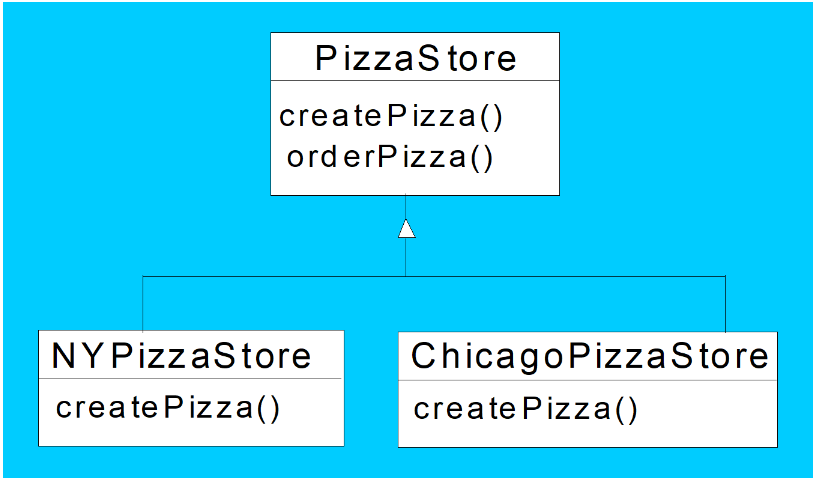
```
public class PizzaStore {
  SimplePizzaFactory factory;
  public PizzaStore(SimplePizzaFactory factory) {
     this.factory = factory;
  }

  public Pizza orderPizza(String type) {
  Pizza pizza;
  pizza = factory.createPizza(type);
  pizza.prepare();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza;
}
```

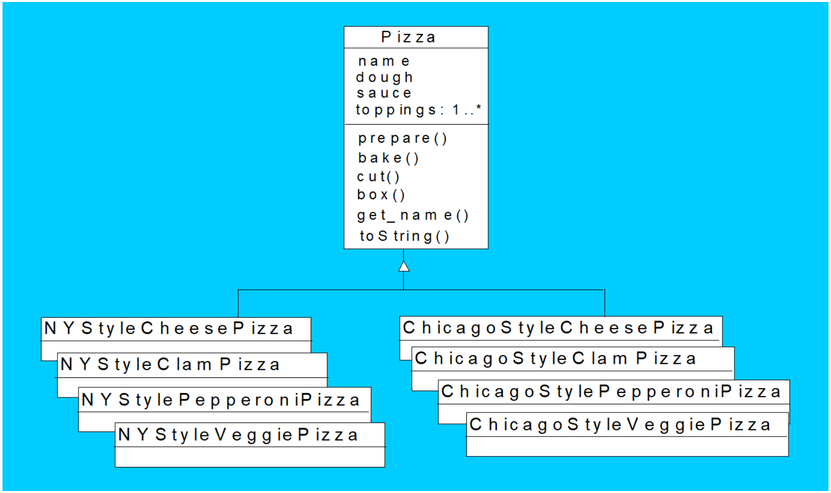
More changes are needed ...

Franchises in different parts of the country are now adding their own special touches to the pizza. For example, customers at the franchise in New York like a thin base, with tasty sauce and little cheese. However, customers in Chicago prefer a thick base, rich sauce and a lot of cheese. Some franchises also cut the pizza slices differently (e.g. square)
You need to extend the system to cater for this.

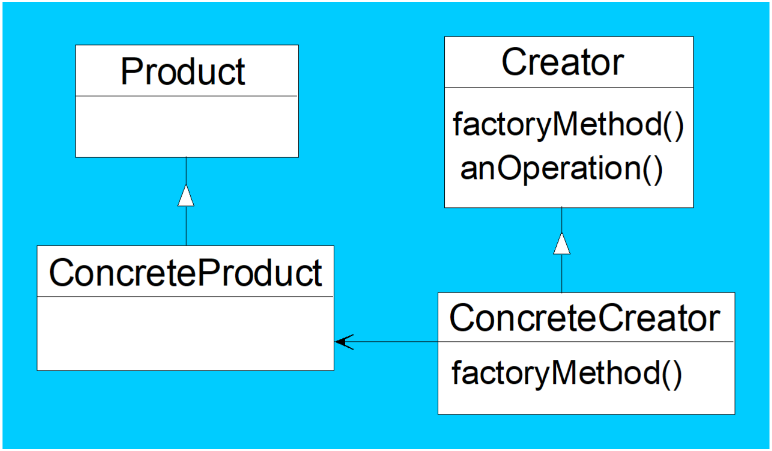
Creator Class



Product class



Structure


Product
ConcreteProduct
Creator
factoryMethod()
anOperation()
ConcreteCreator
factoryMethod()

In summary

- The creator gives you an interface for writing objects.
- The creator specifies the “factory method”, e.g. the createPizza method.
- Other methods generally included in the creator are methods that operate on the product produced by the creator, e.g. orderPizza.
- Only subclasses implement the factory method.

Final thoughts

- When you have code that instantiates concrete classes that may change, encapsulate the code that changes.
- The factory pattern allows you to encapsulate the behaviour of the instantiation.
- Duplication of code is prevented.
- Client code is decoupled from actual implementations.
- Programming to interface not implementation.

Builder Pattern

- Separate the construction of a complex object from its representation so that the same construction process can create different representations
- Use the Builder pattern when:
    - the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
    - the construction process must allow different representations for the object that is constructed

The address example

- Telescoping Constructor
- A constructor with many arguments, possibly of the same type
- Example: `new CanadianAddress(“Apt 101”, “301”, “College St”, “Toronto”, “ON”, “B4D C4T”)`

Problems with telescoping constructor

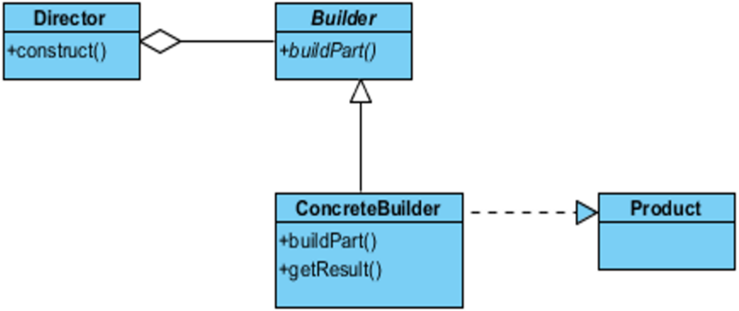
- Error prone: correctness might depend on argument order
- Developers might get it wrong (or waste time checking)
- Compiler cannot save you
- Harder to test: more arguments implies even more combinations to test
- Results in ugly code

The builder pattern

- Break construction into separate tasks:
- Collecting arguments
- Creating the instance
- Solve the telescoping constructor problem by collecting arguments one at a time
- Improve code quality

THE UML STRUCTURE

UML Structure



The correct Canadian Address

- We "solved" the telescoping constructor problem by:
    - Providing a default constructor (i.e. a constructor with 0 arguments).
    - Providing setters for all properties.
- The problem is that we made it way too easy for other developers/classes to (accidently) create invalid/illegal instances (e.g. an address that is missing the street name).
- We want to ensure that it is impossible to create invalid/illegal CanadianAddress instances.
- Notice that this requirement is easy to satisfy with our previous solution - If the arguments to the constructor are invalid, we throw an exception.

Introduce a middleman

- We will solve our problem like we usually do - Introduce a middleman.
- Our middleman is the CanadianAddressBuilder class, which breaks the task (of instantiating a CanadianAddress) into two parts:
- Collect all the arguments.
- Create an instance of CanadianAddress.
- Essentially, the builder is just a "container" for the CanadianAddress constructor arguments.

Finally

- Here is what we did:
- We made Builder a public static class inside CanadianAddress.
    - The same way we define classes inside packages, we can define static classes inside other classes.
    - We refer to the class as CanadianAddress.Builder.
    - CanadianAddress and CanadianAddress.Builder can access each other's private methods, constructors, and instance variables.
- We change the constructor of CanadianAddress to be private, and take a single argument of type CanadianAddress.Builder.
- You can see how the builder is used in Main.java (almost the same as before, we only changed the name of the class to be CanadianAddress.Builder).

SOLID

- S Single responsibility principle
- O Open/closed principle
- L Liskov substitution principle
- I Interface segregation principle
- D Dependency inversion principle

S: Single Responsibility Principle

- every class should have a single responsibility
- responsibility should be entirely encapsulated by the class
- all class services should be aligned with that responsibility

Why?

- makes the class more robust
- makes the class more reusable

A class should have only one reason to change

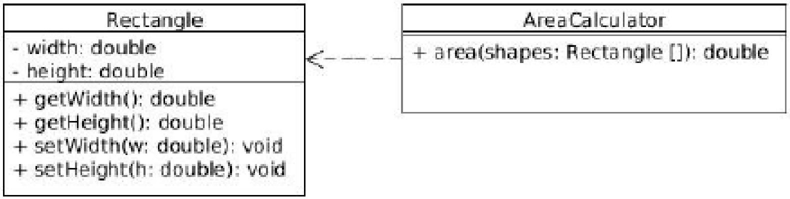
WENGER
Single Responsibility Principle
Just because you *can* doesn't mean you *should*.

O: Open/Closed Principle

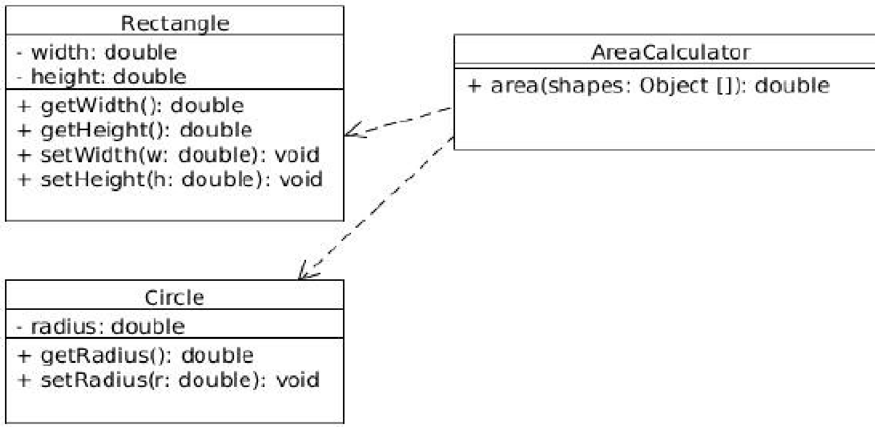
- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
- add new features not by modifying the original class, but rather by extending it and adding new behaviours
- the derived class may or may not have the same interface as the original class

O: OPEN/CLOSED PRINCIPLE

- Example:
  - area calculates the area of all Rectangles in the input.
- What if we need to add more shapes?



Open/Closed Principle



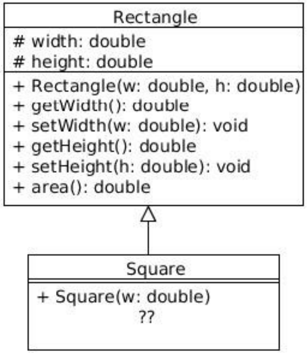
A class should be open for extensibility, but closed for modification.

L: Liskov substitution principle

- If S is a subtype of T, then objects of type S may be substituted for objects of type T, without altering any of the desired properties of the program.
- “S is a subtype of T”? In Java, S is a child class of T, or S implements interface T.
- For example, if C is a child class of P, then we should be able to substitute C for P in our code without breaking it.

Liskov substitution principle

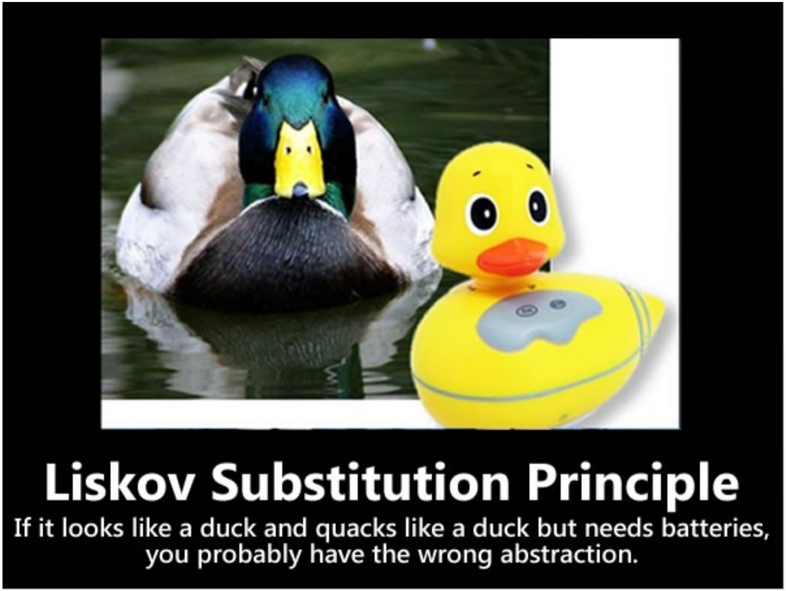
- A classic example of breaking this principle:



L: Liskov substitution principle

- In OO programming and design, unlike in math, it is not the case that a Square is a Rectangle!
- This is because a Rectangle has more behaviours than a Square, not less.
- The LSP is related to the Open/Close principle: the sub classes should only extend (add behaviours), not modify or remove them.

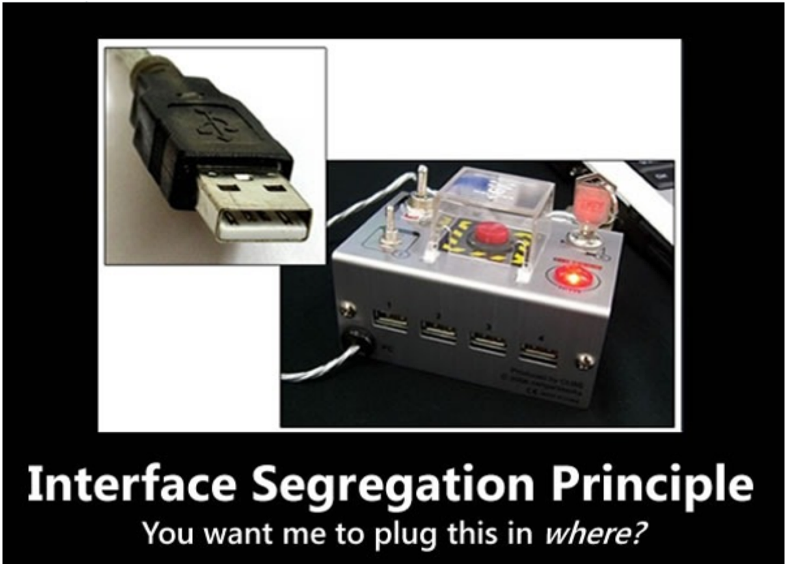
Liskov substitution principle


Liskov Substitution Principle
If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

I: Interface segregation principle

- No client should be forced to depend on methods it doesn't use.
- Better to have lots of small, specific interfaces than fewer larger ones.
- Easier to extend and modify the design.

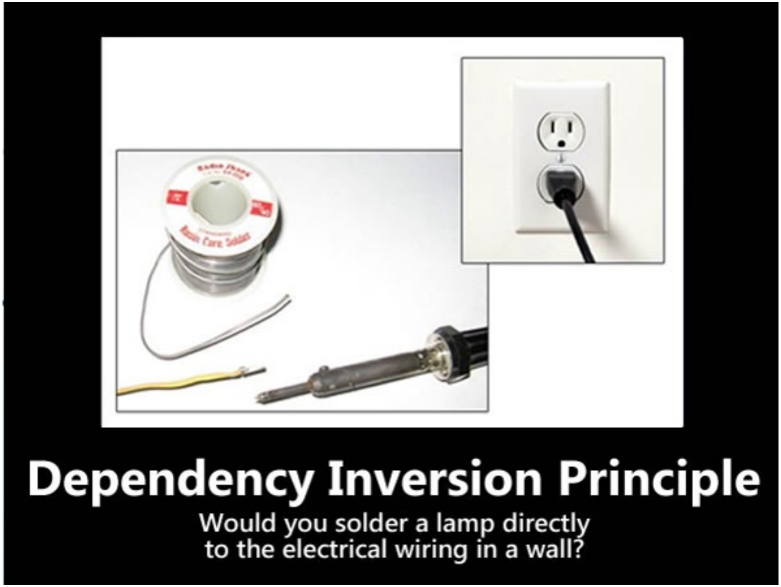
Clients should not be forced to depend on methods they do not use


Interface Segregation Principle
You want me to plug this in *where?*

D: Dependency inversion principle

- When building a complex system, we may be tempted to define the “low-level” classes first and then build the “higher-level” classes that use the low-level classes directly.
- (for simplicity, “low-level” - components, “high-level” - classes that use those components)
- But this approach is not flexible! What if we need to replace a low-level class? The logic in the high-level class will need to be replaced.
- To avoid such problems, we can introduce an abstraction layer between low-level classes and high -level classes.

Dependency inversion principle


Dependency Inversion Principle
Would you solder a lamp directly to the electrical wiring in a wall?