OO Design Principles SHORT

Design Principles

Design Principles

- Law of Demeter
- SOLID

Law of Demeter

Law of Demeter

- Goal - reduce coupling to an understandable and maintainable level.
- Violation - link multiple/many method calls together using . notation as follows:

```
getPreferencesService().getDimension(MAIN_WINDOW_ID).setWidth(700);
getPreferencesService().getColorScheme(OCEAN).getTextColor().setColor(BLUE);

if (getCommandProcessor().getPool().getSize() >= MAX_POOL_SIZE)
{
    // warning
}
else
{
    // process command
}
```

Law of Demeter

- various assumptions - for example, that the components are all accessible and correctly initialized.
- Fragility / consequential changes - changes to the details of used classes almost inevitably lead to changes in the own class as well.

```
getPreferencesService().getDimension(MAIN_WINDOW_ID).setWidth(700);
getPreferencesService().getColorScheme(OCEAN).getTextColor().setColor(BLUE);

if (getCommandProcessor().getPool().getSize() >= MAX_POOL_SIZE)
{
    // warning
}
else
{
    // process command
}
```

Law of Demeter

- **Therefore, .-chaining is potentially "evil"**

```
car.getOwner().getAddress().getStreet();
```

- **Extraction of individual objects does not make it better**

```
Owner owner = car.getOwner();
Address ownerAddress = owner.getAddress();
Street ownerStreet = ownerAddress.getStreet();
```

- **Idea: control over the objects involved**

Law of Demeter

- »*Don’t talk to strangers*«

- «Don't call us, we'll call you»

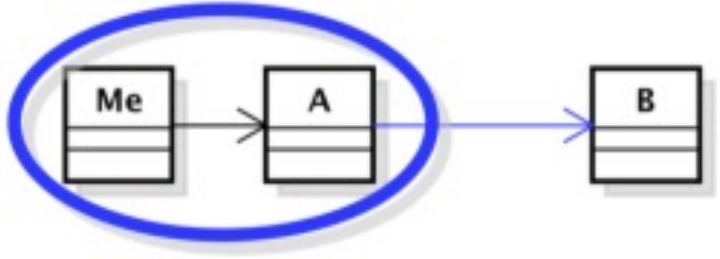
Law of Demeter

- »*Don't talk to strangers*« ...

```
myObject.Never().Talks().To().Strangers();
```

- Hint: »***Only talk to your immediate friends.***«

- How to achieve it? Rules for the design of calls:
  1. methods of your own class,
  2. methods of objects passed as parameters,
  3. methods of objects created by the own object itself, or
  4. methods of associated classes.



Rating: Law of Demeter

- ✓ a rather loosely coupled system with few dependencies is the result, which facilitates reusability.
- ✓ Individual classes are easier to change and there is less propagation of consequential changes in the system
- ✓ Easier testability due to smaller number of dependencies
- ✓ More methods in the interface of the own class, because some methods cannot be addressed now possibly over indirection.

Law of Demeter

What about Fluent Interfaces?
Don't they violate the Law of Demeter?

Law of Demeter – Fluent Interfaces

- **NO!**
- **They have a completely different direction of design**
- **Especially they operate on one and the same object (thus allowed by rule 4)**

```
Report report = new ReportBuilder()
        .withHeader("Law of Demeter Report")
        .withBorder(2, Color.BLACK)
        .titled("Fluent Interfaces are fine with Law of Demeter")
        .writtenBy("Michael Inden")
        .outputAs(OutputType.PDF)
        .build();
```

Direct Mail Informatics

SOLID

- **S** R P – Single Responsibility Principle
- **O** C P – Open Closed Principle
- **L** S P – Liskov Substitution Principle
- **I** S P – Interface Segregation Principle
- **D** I P – Dependency Inversion Principle

Quelle: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

S R P – Single Responsiblity Principle

- *fulfill exactly one clearly defined task*
- ***those who do many things at once rarely succeed in all of them.***
- *high cohesion, i.e. a high degree of cohesion*
- *orthogonality = combine functionalities easily without (major) side effects*

- *indication for violation: that it is difficult to find concise names for a method or a class or that this name contains several verbs or nouns.*
- ***sheer method length a pretty good indicator***

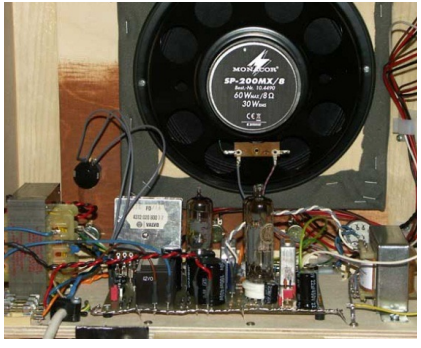
OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

Quelle: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Open Closed Principle

Open for Modifications



Closed for Extensions

Open for Extensions. Closed for Modifications

Open Closed Principle

Modules conforming to OCP …

… are open for extension:
behaviour of module can be extended to meet new requirements.

… are closed for modification:
extending the module does not result in changes to the source or binary code of the module.

Open Closed Principle

- easy extensibility
- correct encapsulation and separation of responsibilities.
- a class should only have to change after its completion if completely new requirements or functionalities have to be integrated or if errors have to be corrected.

Open Closed Principle - Example

The following code does **not conform** to OCP:

```
public class Drawer {
  public void drawAll(List<Shape> shapes) {
     for (Shape shape : shapes) {
          if (shape instanceof Circle) {
                  drawCircle((Circle) shape);
          }
          if (shape instanceof Square) {
                  drawSquare((Square) shape);
          }
     }
  }
  ...
```

Open Closed Principle - Example

The following code does **conform** to OCP:

```
public class Drawer {
  public void drawAll(List<Shape> shapes) {
     for (Shape shape : shapes) {
          shape.draw()
     }
  }
  ...
```

Open Closed Principle

- Example game application with different bonus elements, such as extra lives or additional equipment, as an incentive.
- Task: design a new level and integrate new types of bonus elements there.
- Desire: as simple and extensible as possible, ideally only create new classes for the new, special bonus elements.
- If the base design already follows the OCP: then all bonus elements have a common interface or an (abstract) base class
- The rest of the application is hardly affected, or in the best case not affected at all, by changes or extensions to the bonus elements.

Liskov Substitution Principle

Quelle: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Liskov Substitution Principle Objectives

- Use inheritance the right way
- Know the possible traps of Inheritance and Polymorphism

More than IS-A

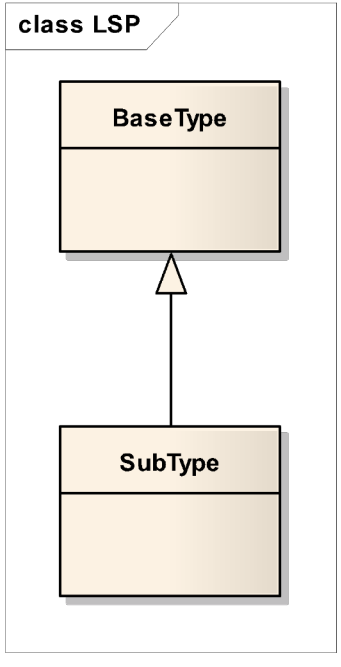
- Know about "Design by Contract" to help making your design and code less fragile
- Take care of important Preconditions, Postconditions and Invariants in your Design
- Apply this knowledge in your designs, documentations and unit tests

Design by Contract

Liskov Substitution Principle

"Subtypes must be substitutable for their base type"

(Barbara Liskov, 1988)



Wherever an object of BaseType is used,

an instance of SubType could be used instead.

```
BaseType b = new SubType();
b.doSomething();
```

A simple example of LSP Violation

```
public class B {

   public String getName() {
      return "Base";
   }

}
```

```
public class E extends B {

   public String getName() {
      throw new UnsupportedOperationException();
   }

}
```

```
public class SomeClient {

   @Override

   public String lowerName(B b) {
      return b.getName().toLowerCase();

   }

}
```

SomeClient makes an assumption about B that does not hold for E

Pay attention when you inherit …

Your Subclass has to ensure that the following holds:

The "Contract" of the Superclass must still be fulfilled.

Contract:

What users expect of the operation …

- Given Preconditions are fulfilled before execution
- The Postconditions must be fulfilled after execution

```
// Precondition: Enough money on my account
public void transferMoney(Account other, int amount);
// Postcondition: money moved to other account
```

Contracts

Which letterbox would you trust??

→ Contracts belong to the interface of a class

→ Document any pre- and postconditions.

Precondition	Post until 18:00, Porto CHF 1.-- (A-Post)	What the Client needs to ensure
Postcondition	Delivery next day until 12:00	What the Supplier needs to ensure

Contract of a Class

Precondition: What users must ensure before calling an operation

Postcondition: The least that users can expect to be fulfilled after execution
(when precondition was fulfilled)

Invariant: Special condition on the state of an object that is always true
(before and after each public operation execution)

Belongs to the interface of a class:

- either express it as conditions (if supported by language)
- or document it in text style at least

Inheriting a Contract

Your Subclass has to ensure that the following holds:

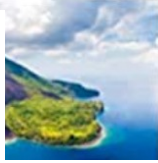
The "Contract" of the Superclass must still be fulfilled.

What happens with the Contract in the Subclass?

- It is inherited too
- Preconditions may be relaxed (but not restricted!)
- Postconditions may be restricted (but not relaxed!)

In other words … you are allowed to:

- Expect less from the user (caller) but not more
- Provide more than the user expects but not less



```
class CalculationException extends Exception
{
    // ...
}
class SpecialCalculationException extends CalculationException
{
    // ...
}

class BaseFigure
{
    Number calcArea() throws CalculationException
    {
        return new Double(getWidth() * getHeight());
    }
}

class Polygon extends BaseFigure
{
    // Speziellere Rückgabe (Double extends Number) und speziellere Exception
    Double calcArea() throws SpecialCalculationException
    {
        // Bewusst, um gleich ein Problem zu zeigen
        return null;
    }
}
```

L S P – Liskov Substituion Principle

```
class Client
{
    void doSomethingWithFigure(final BaseFigure figure)
    {
        final Number result = figure.calcArea();
        // NullPointerException für Polygon
        final double area = result.doubleValue();
        // ...
    }
}
```

Conclusion / Pragmatics

- Inheritance is more than just an "is a" relationship
- Inheritance means "substitutable“
- Therefore you have to think hard if an inheritance-relation in your design is appropriate

- Document the expectations of clients:
  Specify the contract: Pre- & Post-conditions, Invariants
  (most important: contracts in interface classes!)

Liskov Substituion Principle

Specialcase even if „is-a“ holds

Liskov Substituion Principle – Example

```
public class Rectangle
{
    private int height;
    private int width;

    public Rectangle(int height, int width)
    {
        this.height = height;
        this.width = width;
    }

    public int computeArea()
    {
        return this.height * this.width;
    }

    public void setHeight(int height) { this.height = height; }
    public void setWidth(int width) { this.width = width; }

    public int getHeight() { return this.height; }
    public int getWidth()  { return this.width;  }
}
```

Liskov Substituion Principle

```
public class Square extends Rectangle
{
    public Square(int sideLength)
    {
        super(sideLength, sideLength);
    }

    protected void setSideLength(final int sideLength)
    {
        // gleiche Seitenlänge sicherstellen
        super.setHeight(sideLength);
        super.setWidth(sideLength);
    }

    // Hints and Problems LATER
    @Override
    public void setHeight(int height) { setSideLength(height); }
    @Override
    public void setWidth(int width) { setSideLength(width); }
}
```

Liskov Substituion Principle

```
public class RectangleTest
{
    @Test
    void testAreaCalulaction()
    {
        Rectangle rect = new Rectangle(7, 6);

        rect.setWidth(5);
        rect.setHeight(10);

        assertEquals(50, rect.computeArea());
    }
}
```

For Rectangle this works well, but according to LSP you should also be able to use Square

Liskov Substituion Principle

```
public class SquareTest
{
    @Test
    void testAreaCalulaction()
    {
        Rectangle rect = new Square(7);

        rect.setWidth(5);
        rect.setHeight(10);

        assertEquals(50, rect.computeArea());

        // for Square either 25 or 100 depending on the order ...
    }
}
```

- For Square the order is decisive, but in any case the result calculated for Rectangle does not fit!
- actually one would like to forbid the set() methods ...

Liskov Substituion Principle

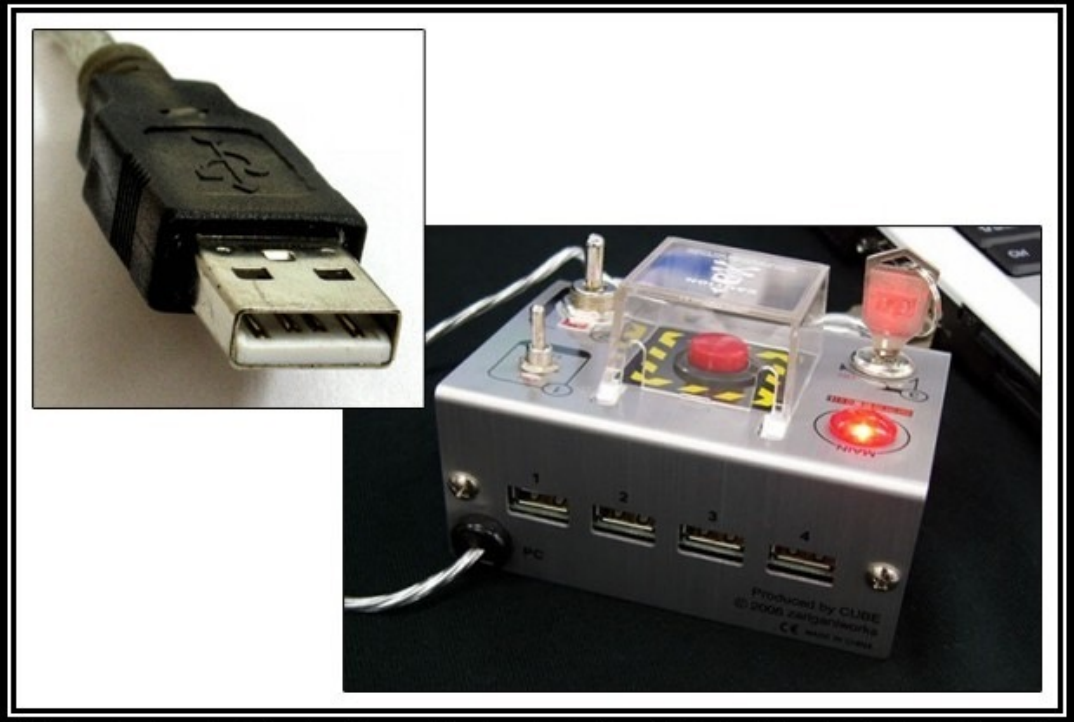
Everything clear so far? Let's see!

Liskov Substituion Principle

```
Rectangle rect1 = new Rectangle(7, 6);
rect1.setWidth(5)
rect1.setHeight(5)


Rectangle rect2 = new Rectangle(7, 7);
```

Hmmm ... so you can semantically make a square out of a rectangle, but it has the type Rectangle .... There is still some work to do ;-)



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

Quelle: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

I S P – Interface Segregation Principle

- *Create an interface that is as specific as possible, tailored to the task at hand or to the client.*
- *Often in practice one sees rather too broad or too unspecific interfaces, which consequently almost always also offer functionality that a client does not need, such as the following:*

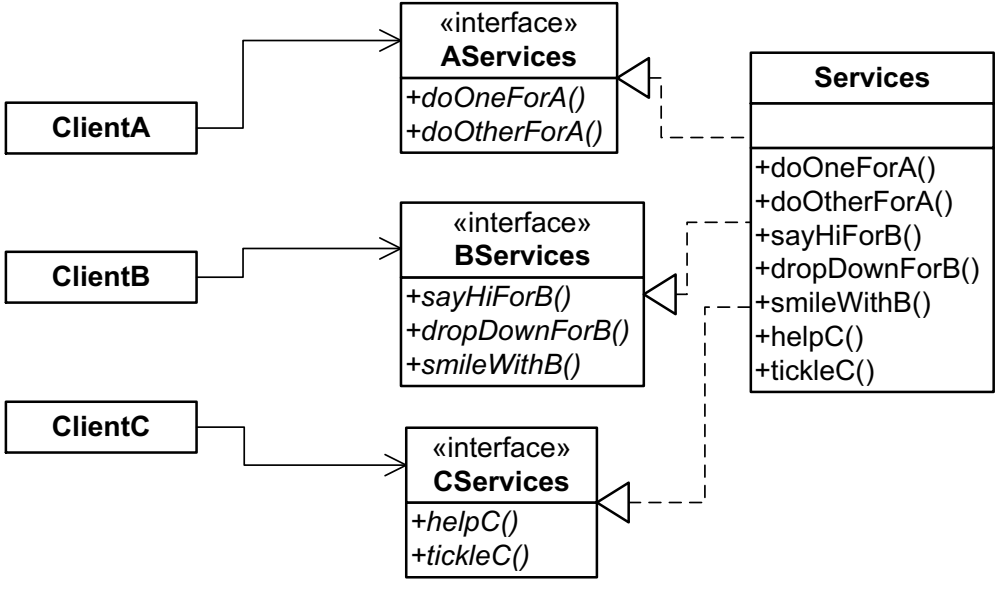
```
interface IUniversalFileCustomerAndPizzaService
{
    void scanDisk(final Drive drive) throws IOException;
    boolean rename(final File fileToRename,
                   final String newName) throws IOException;

    Customer findCustomerByName(final String name);
    Iterable<Customer> getAllCustomers(final FilterCondition filterCondition);

    boolean orderPizza(final long customerId, final Pizza pizza);
}
```

Interface Segregation Principle

Several client specific interfaces are better than one single, general interface



not only one interface for all clients

one interface per kind of client

Interface Segregation Principle

```
interface IFileService
{
    void scanDisk(final Drive drive) throws IOException;
    boolean rename(final File fileToRename,
                   final String newName) throws IOException;
}

interface ICustomerService
{
    Customer findCustomerByName(final String name);
    Iterable<Customer> getAllCustomers(final FilterCondition filterCondition);
}

interface IPizzaService
{
    boolean orderPizza(final long customerId, final Pizza pizza);
}
```

- the design of a successful interface is not so easy at all
- **It is a matter of finding the "right" granularity.**
- This requires a bit of experience, intuition, and also a little trial and error - especially a consideration from the point of view of possible users.

DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Quelle: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

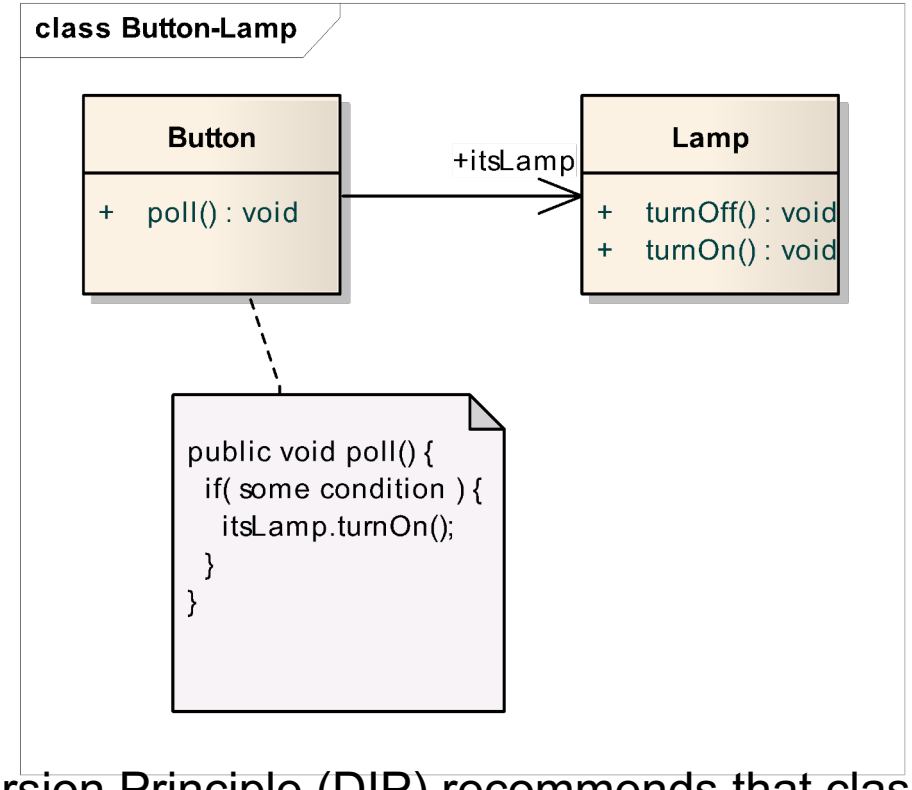
DIP – Dependency Inversion Principle

- “High-level modules should not depend on low-level modules. Both should depend on abstractions”
- “Abstractions should not depend on details. Details should depend on abstractions”

- Whaaaaat?

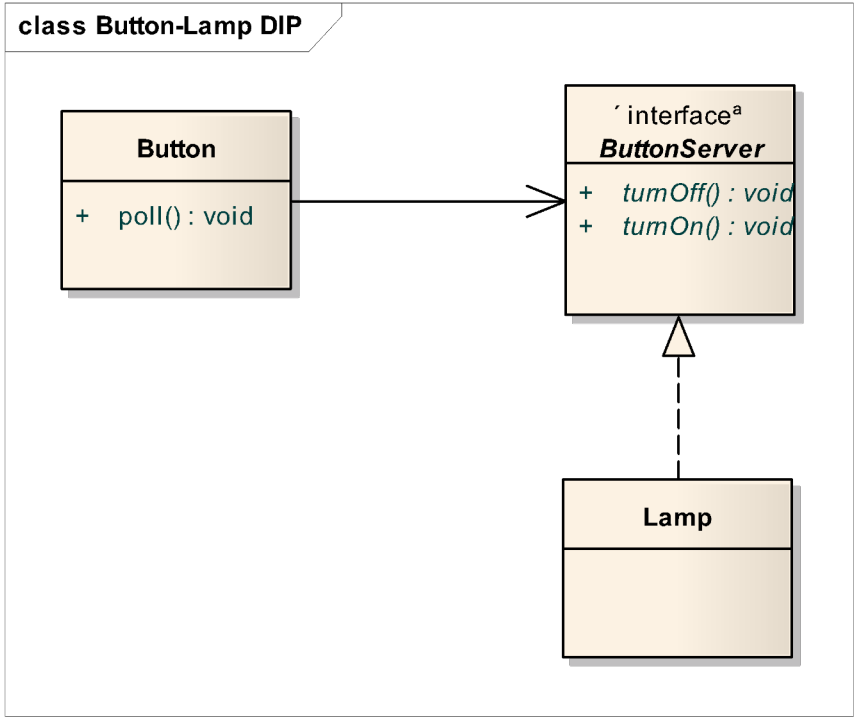
- **If possible, use interfaces (or abstract classes) to decouple (concrete) classes from each other.**

DIP What is wrong here?

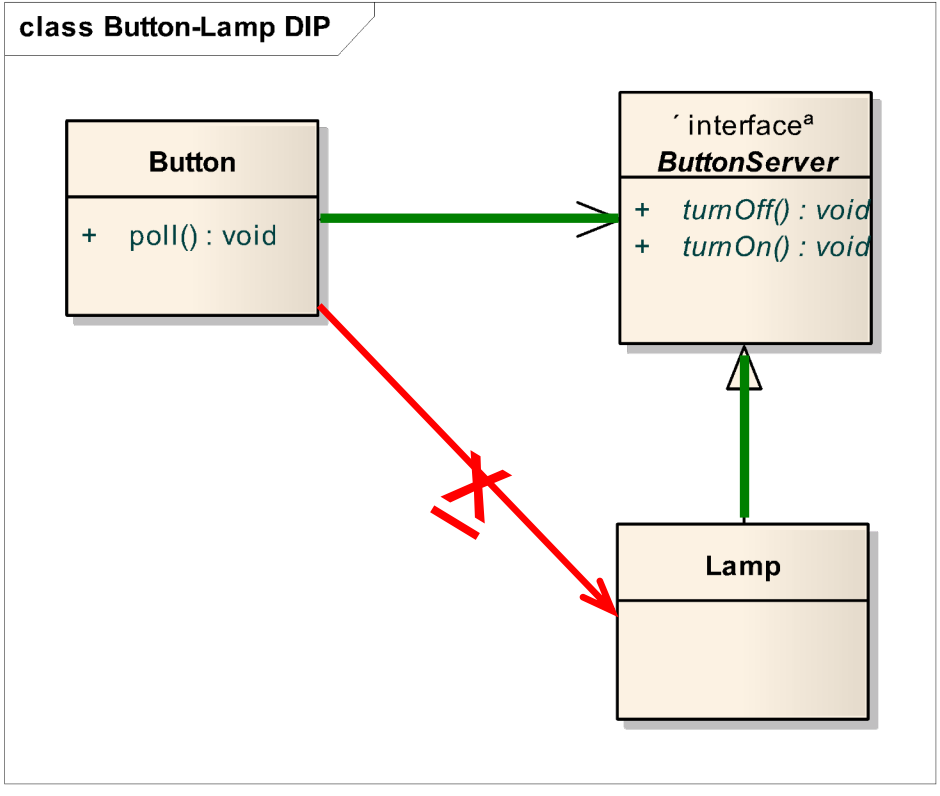


The Dependency Inversion Principle (DIP) recommends that classes should be as independent as possible from other concrete classes by resolving a dependency on a concrete class by a reference to its interface.

Introducing Button “Server” Interface



DIP Inverting the dependency



Button depends no more on Lamp

Instead Lamp has a dependency on the abstraction (=ButtonServer).

```
public class PizzaService
{
   private final Discount discount;
   private final CustomerDAO customerDAO;

   private final Map<Long, Receipt> customerToReceipt = new HashMap<>();

   public PizzaService()
   {
      // Direkte Abhängigkeiten
      discount = new Discount();
      customerDAO = new CustomerDAO();
   }

   public void orderPizza(final long customerId, final Pizza pizza)
   {
      final Customer customer = customerDAO.findById(customerId);
      customerToReceipt.putIfAbsent(customerId, new Receipt(customer));

      final Receipt receipt = customerToReceipt.get(customerId);
      final double price = discount.apply(pizza);
      receipt.addEntry(pizza, price);
   }

    ...

}
```

```
public class PizzaService
{
   private final ICustomerRepository customerRepository;

   private Map<Long, Receipt> customerToReceipt = new HashMap<>();

   // Konstruktor-Injektion
   public PizzaService(final ICustomerRepository customerRepository)
   {
      this.customerRepository = customerRepository;
   }

   // Method-Injektion
   public void orderPizza(final long customerId, final Pizza pizza,
                          final IDiscountStrategy discountStrategy)
   {
      final Customer customer = customerRepository.findById(customerId);
      customerToReceipt.putIfAbsent(customerId, new Receipt(customer));

      final Receipt receipt = customerToReceipt.get(customerId);
      final double price = discountStrategy.apply(pizza);
      receipt.addEntry(pizza, price);
   }

   ...

}
```

Thank you for participating and happy coding!