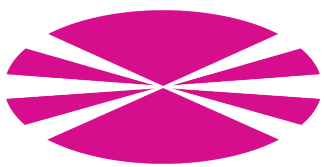
Unit 5: Design Principles

Software Design (614G01015)

David Alonso Ríos
Eduardo Mosqueira Rey (Coordinator)

Department of Computer Science, Faculty of Computer Science



UNIVERSIDADE DA CORUÑA

Table of Contents

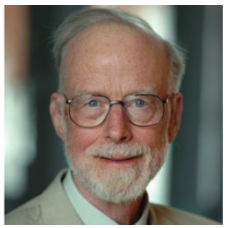
Table of Contents

Quality in design

Simple software vs. complex software

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

C.A.R. Hoare, Turing Award Lecture 1980



Rotten software

"...But then something goes wrong. The software starts to rot like a piece of bad meat. As time goes by, the rotting continues. Eventually, the sheer effort required to make even the simplest of changes becomes so onerous that the developers cry for a redesign"

Robert C. Martin

Code smells

Code smells

"A code smell is a surface indication that usually corresponds to a deeper problem in the system."

Martin Fowler

Some code smells...

- **Duplicated code**
  - Identical or very similar code exists in more than one location.
- **Large class**
  - A class that has grown too large (God object).
- **Lazy class**
  - A class that does too little.
- **Data class**
  - Classes with all data and no behavior.
- **Feature envy**
  - A class that uses methods of another class excessively.
- **Inappropriate intimacy**
  - A class that has dependencies on implementation details of another class.

Some code smells...

- **Refused bequest**
  - A class overrides a method in such a way that the contract of the base class is not honored by the derived class.
- **Data clumps**
  - Bunches of data that hang around together really ought to be made into their own object.
- **Long method**
  - A method that has grown too large.
- **Long parameter list**
  - Parameter list that is hard to read, and makes calling and testing the function complicated.
- **Switch statements**
  - Most times you see a switch statement you should consider polymorphism.

Design smells

Design smells

"Design smells are symptoms of poor design.
These symptoms are similar in nature to code smells, but they are at a higher level. They are smells that pervade the overall structure of the software rather than a small section of code."

Robert C. Martin

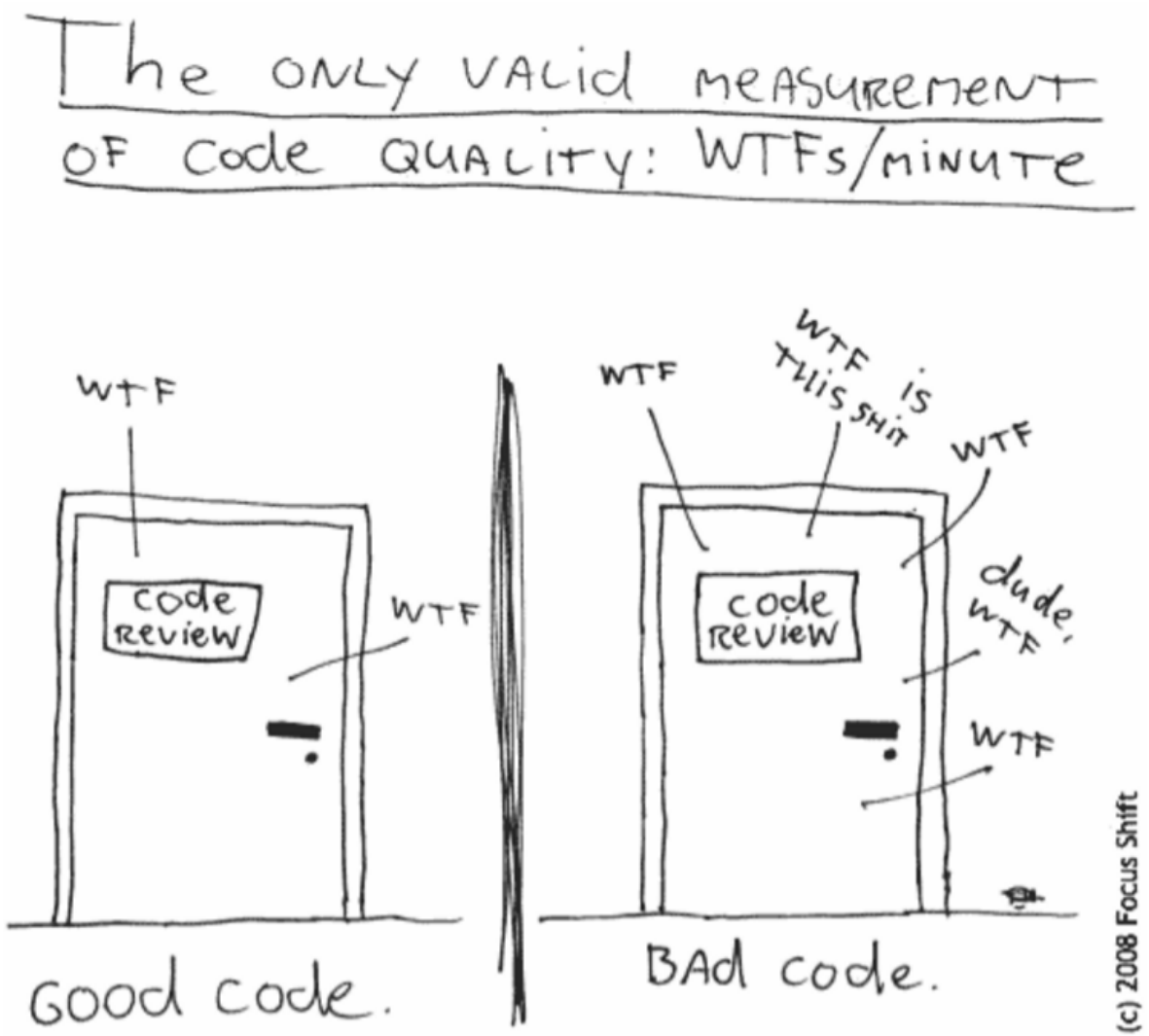
Design smells: The odors of rotting software

- **Rigidity**
  - The system is hard to change because every change forces many other changes to other parts of the system.
- **Fragility**
  - Changes cause the system to break in places that have no conceptual relationship to the part that was changed.
- **Immobility**
  - It is hard to disentangle the system into components that can be reused in other systems.
- **Viscosity**
  - Doing things right is harder than doing things wrong. Design-preserving methods are harder to employ than “hacks”.

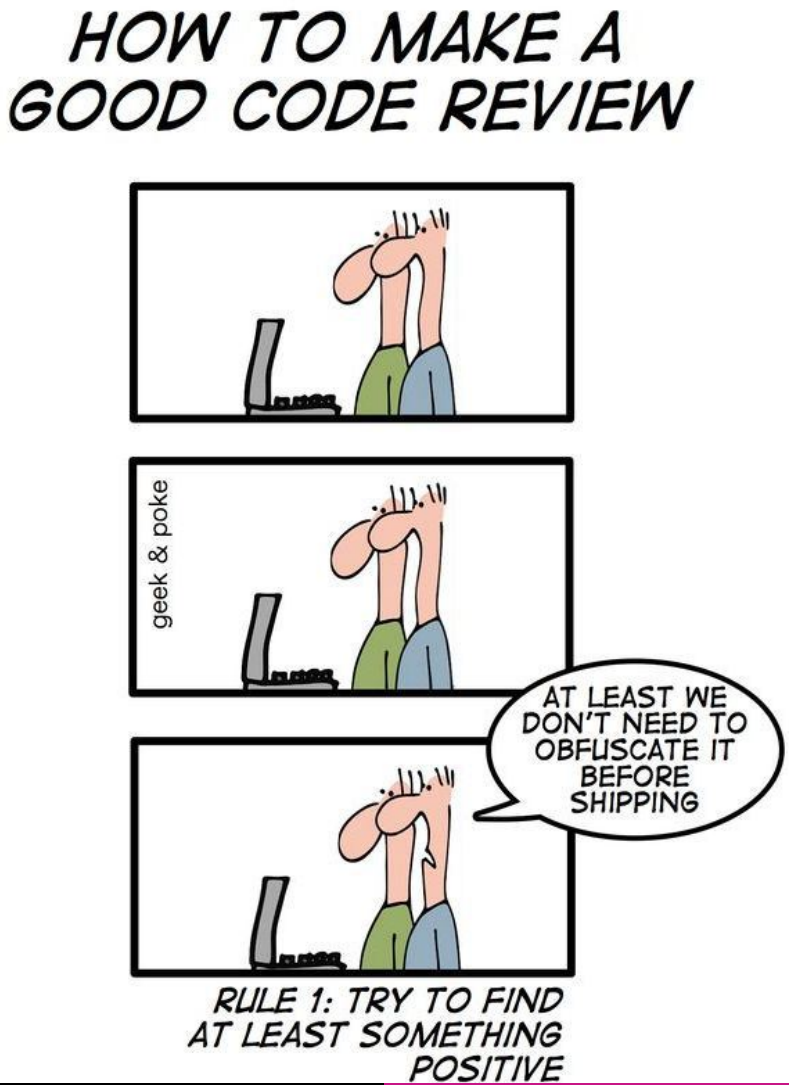
Design smells: The odors of rotting software

- **Needless Complexity**
  - Overdesign. The design contains infrastructure that adds no direct benefit.
- **Needless Repetition**
  - The design contains repeating structures that could be unified under a single abstraction. Cut and paste may be useful text-editing operations, but they can be disastrous code-editing operations.
- **Opacity**
  - It is hard to read and understand. It does not express its intent well.

WTFs / Minute


The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF
WTF IS THIS SHIT
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift

Code review


HOW TO MAKE A GOOD CODE REVIEW
geek & poke
AT LEAST WE DON'T NEED TO OBFUSCATE IT BEFORE SHIPPING
RULE 1: TRY TO FIND AT LEAST SOMETHING POSITIVE

Design principles

Design principles

Guidelines that, when applied together, will make it easy for a programmer to develop software that is easy to maintain and extend.

- They are basic and generic recommendations that help developers eliminate design smells and build better designs.
- Robert C. Martin gathered them under the acronym SOLID (but there are many other principles $\Rightarrow$ U6: Design Patterns).
- They are not rules or laws, they are heuristics that represent common-sense practices that can help you stay out of trouble.
- They have been observed to work in many cases, but there is no proof that they always work. Over-conformance to the principles leads to the design smell of *needless complexity*.

Table of Contents

SOLID principles

Source: http://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

SOLID principles

- **S**ingle Responsibility Principle (SRP)
- **O**pen Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

The Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP)

Every object should have a single responsibility entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.

- A responsibility is a change factor, if it changes it should be necessary to change the code that implements it.
- If a class implements a single responsibility then the class **has only one reason to change**.
- If a class implements more than one responsibility then the responsibilities are *coupled*. A change in one of them can affect the others, which lead us to a fragile design.
- If we separate responsibilities we limit the propagation of changes.

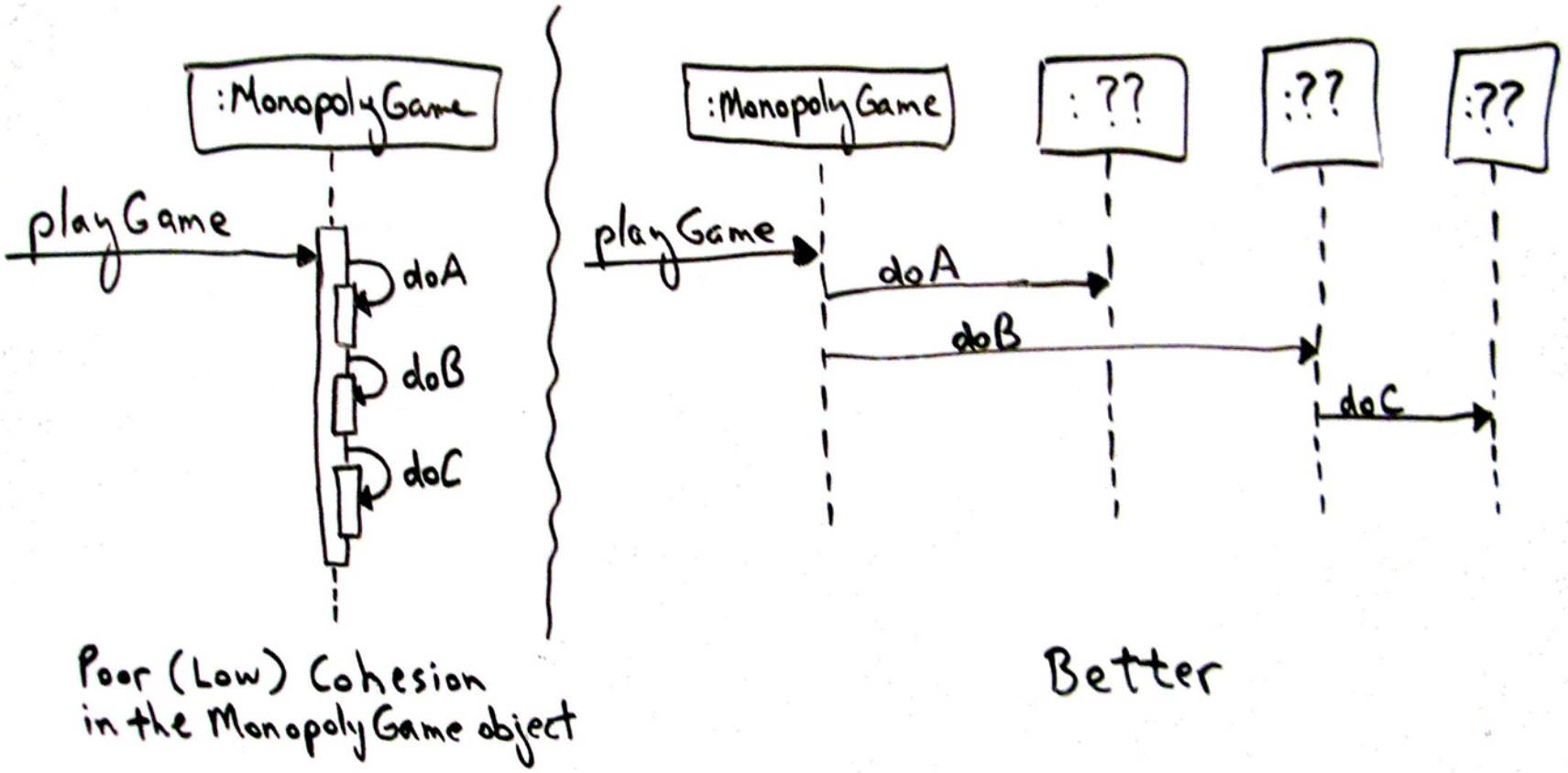
The Single Responsibility Principle (SRP)

Cohesion

It is a measure of the functional relatedness of the elements of an object.

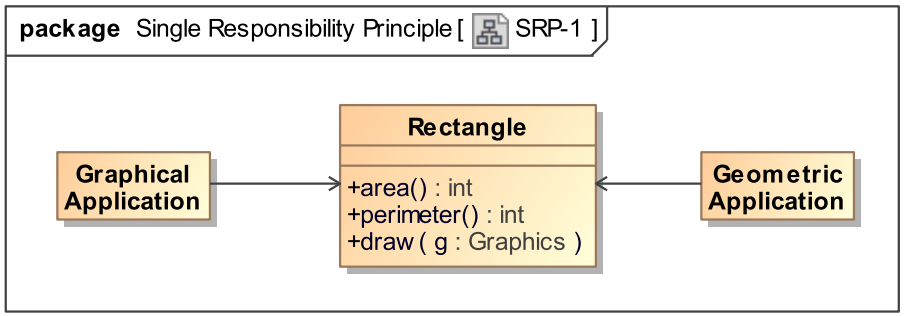
- Measures the amount of work an object is capable of performing.
- It is not easy to measure cohesion, a good estimator can be the number of code lines or methods inside an object.
- Objects with high cohesion are more focused on the objective to solve (single responsibility), are more readable, more reusable and, as a side-effect, normally present low coupling.
- Avoid **God Classes** (i.e., those that monopolize almost all the responsibilities of a program).

Low cohesion vs. high cohesion



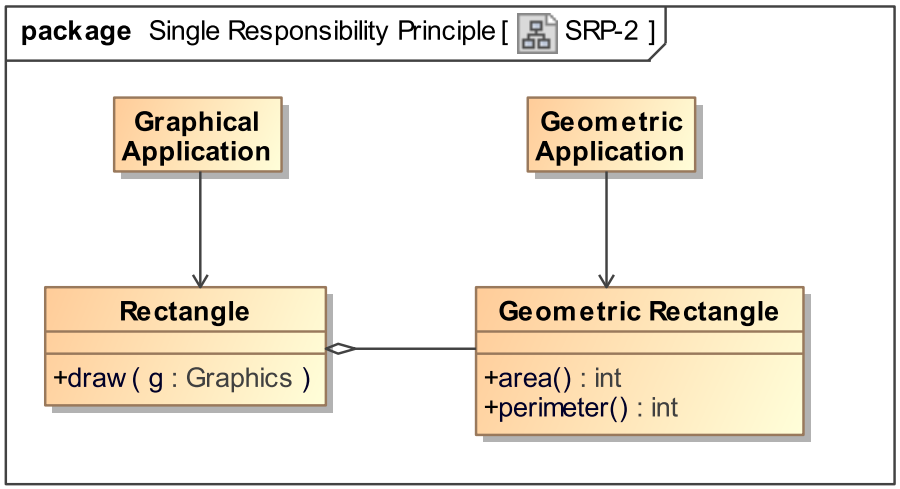
Example

- Two different applications use the `Rectangle` class.
- One application does computational geometry. It uses `Rectangle` to help it with the mathematics of geometric shapes. It never draws the rectangle on the screen.
- The other application is graphical in nature. It may also do some computational geometry, but it definitely draws the rectangle on the screen.
- The `Rectangle` class has two responsibilities. The first responsibility is to provide a mathematical model of the geometry of a rectangle. The second responsibility is to render the rectangle on a graphical user interface.



Example

- If a change to the `GraphicalApplication` causes the `Rectangle` to change for some reason, that change may force us to rebuild, retest, and redeploy the geometry application
- A better design is to separate the two responsibilities into two completely different classes
- This design moves the computational portions of `Rectangle` into the `GeometricRectangle` class. Now changes made to the way rectangles are rendered cannot affect the `GeometricApplication`.



Conclusions

- As all the principles, the SRP shows a path to follow, but it is not a law to obey blindly.
- For example, an object can have several responsibilities if they are related and change in a coordinate way.
- The concepts of *Single Responsibility* and *Cohesion* are very related to the concept of *Coupling* that we will see in more detail in the following unit about design patterns.

The Open Closed Principle (OCP)

The Open-Closed Principle (OCP)

The Open Closed Principle (OCP)

A module (class) should be open for extension but closed for modification.

Bertrand Meyer (1988). Object-oriented software construction

"A module will be said to be open if it is still available for extension. *For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.*
A module will be said to be closed if it is available for use by other modules. *This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding)"*

Bertrand Meyer

The Open-Closed Principle (OCP)

- In other words, we want to be able to change what the modules do, without changing the source code of the modules.
- In object oriented programming this can be done easily through inheritance mechanisms.
- **Example**:
  - A company has several types of employees.
  - The salary of regular employees (type C) is calculated as an annual base quantity plus a quantity that depends on the number of years staying in the company.
  - Type A employees are paid according to the number of projects they carry out throughout a year.
  - Type B employees are like type C but they receive an extra bonus at the end of the year (as they are managers).

The Open-Closed Principle (OCP)

Code that does NOT meet the open closed principle

```
class Personnel {
    enum Type{TYPE_A, TYPE_B, TYPEE_C}
    int base;
    int years;
    int extra;
    int numProjects;
    Type type;

    public int salary() {
        if (type == Type.TYPE_A)
            return (10000 * numProjects);
        else if (type == Type.TYPE_B)
            return base + (1000 * years) + extra;
        else // Type C
            return base + (1000 * years);
    }
}
```

Adding a new type of employee (`TYPE_D`) with a new way of calculating the salary implies changing the previous code.

This code is difficult to maintain, prone to errors and not very encapsulated.

The Open-Closed Principle (OCP)

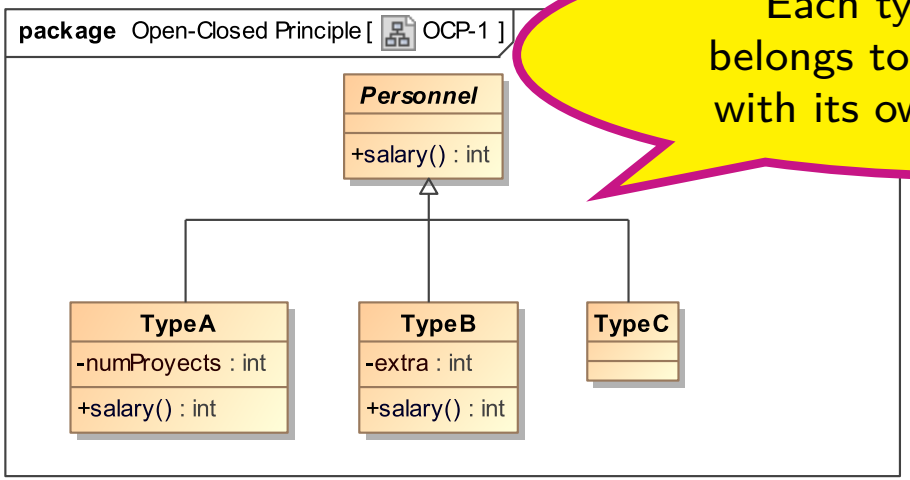
Code that meets the open-closed principle

```
abstract class Personnel {
    protected int base;
    protected int years;

    public int salary() {
        return base + (1000 * years);
    }
}
```

`Personnel` is defined as an abstract class.

`salary()` has a default implementation.



Each type of employee belongs to a class of its own with its own characteristics.

The Open-Closed Principle (OCP)

Code that meets the open-closed principle

```
class TypeA extends Personnel {
    private int numProjects;

    @Override
    public int salary() { return (10000 * numProjects); }
}

class TypeB extends Personnel {
    private int extra;

    @Override
    public int salary() { return super.salary() + extra; }
}

class TypeC extends Personnel { }
```

Each type of employee redefines `salary()` if necessary.

The Open-Closed Principle (OCP)

Client code that uses `Personnel`

```
public int calculateSalary(Personnel p)
    int salary = p.salary();
    int socialSecurity = p.socialSecurit
    int withholding = p.withholding();

    return salary - socialSecurity - withholding;
}
```

The call to `salary()` uses dynamic binding to call the method `salary()` of the dynamic class that is into `p` (the same in the other methods).

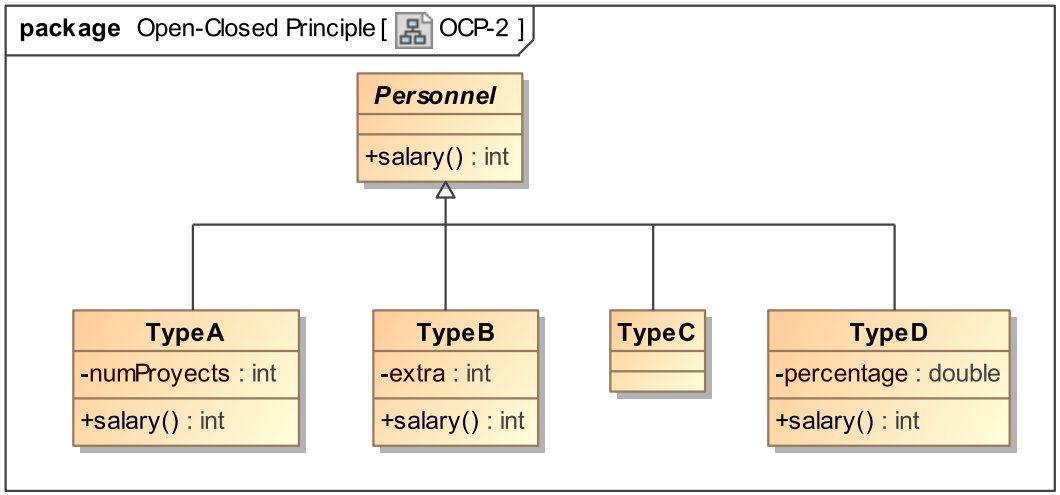
The Open-Closed Principle (OCP)

Code that meets the open-closed prin

```
class TypeD extends Personnel {
    private double percentage;

    @Override
    public int salary() { return (int)(super.salary() * percentage); }
}
```

Adding a new type of employee (`TypeD`) means only adding a new subclass to the system (without modifying the existing code).



The Open-Closed Principle (OCP)

Client code that uses `Personnel`

```
TypeD typeD = new TypeD();
calculateSalary(typeD)

//...

public int calculateSalary(Personnel p) {
    int salary = p.salary();
    int socialSecurity = p.socialSecurity();
    int withholding = p.withholding();

    return salary - socialSecurity - withholding;
}
```

`calculateSalary()` works flawlessly with objects of type D that did not exist when the method was created.

The Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP)

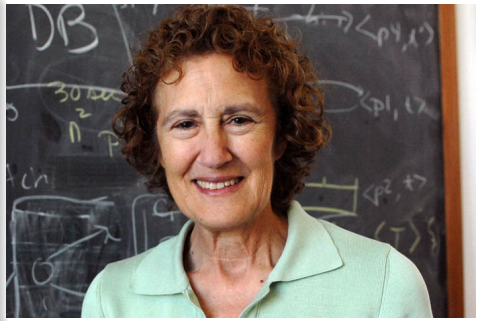
Subclasses should be substitutable for their base classes.

- Subclasses can always be passed when a base class is required, but that does not guarantee that the subclass is a *behavioral subtype* of the base class (a subclass that preserves the behavior of the base class).
- This principle was coined by Barbara Liskov (the first woman to be granted a doctorate in computer science in the United States) in her work regarding data abstraction and type theory.
- It also derives from the concept of Design by Contract (DBC) by Bertrand Meyer.

The Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP) - Liskov statement

"If for each object `o1` of type `S` there is an object `o2` of type `T` such that for all programs `P` defined in terms of `T`, the behavior of `P` is unchanged when `o1` is substituted for `o2` then `S` is a subtype of `T`".



The Liskov Substitution Principle (LSP) - Object oriented statement

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

- **What does "without knowing it" mean?**

The Liskov Substitution Principle (LSP)

- If a subclass does not use overriding $\Rightarrow$ an instance of the subclass and an instance of the superclass will have the same behavior.

Subclass without overriding

```
class Superclass {
    public int value;
    public int sum(int x) { return value + x; }
}
class Subclass extends Superclass {
    public int multiply(int x) { return value * x; }
}

class Client {
    public static void method(Superclass s) {
        s.value = 5;
        System.out.println(s.sum(6));
    }
}
// ...
Client.method(new Superclass()); // Prints 11
Client.method(new Subclass());   // Prints 11
```

The Liskov Substitution Principle (LSP)

- If a subclass DOES use overriding we can find surprises if it is not done properly.

Subclass with wrong overriding

```
class Superclass {
    int value;
    public int sum(int x) { return value + x; }
}
class Subclass extends Superclass {
    @Override
    public int sum(int x) { return value + x + 1; }
}

class Client {
    public static void method(Superclass s) {
        s.value = 5;
        System.out.println(s.sum(6));
    }
}
// ...
Client.method(new Superclass()); // Prints 11
Client.method(new Subclass());   // PRINTS 12 !!
```

The Liskov Substitution Principle (LSP) - Example

Superclass `Rectangle`

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

    public int getHeight() {
        return height;
    }

    public void setHeight(int value) {
        height = value;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int value){
        width = value;
    }
}
```

A square is a rectangle with two equal sides.

Subclass `Square`

```
public class Square extends Rectangle {
    @Override
    public void setHeight(int value) {
        super.setHeight(value);
        super.setWidth(value);
    }

    @Override
    public void setWidth(int value) {
        super.setHeight(value);
        super.setWidth(value);
    }
}
```

The Liskov Substitution Principle (LSP) - Example

`Square` is not a behavioral subtype of `Rectangle`

```
public class ClientRectangle {
    public static void clientMethod(Rectangle r) {
        r.setWidth(5);
        r.setHeight(6);
        if ((r.getWidth() * r.getHeight()) == 30) {
            System.out.println("Correct multiplication...");
        } else {
            System.out.println("THIS IS IMPOSSIBLE !!!");
        }
    }

    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        ClientRectangle.clientMethod(r); //
        Correct multiplication...
        Square s = new Square();
        ClientRectangle.clientMethod(s); // THIS IS IMPOSSIBLE !!!
    }
}
```

The Liskov Substitution Principle (LSP) - Example

- The `Square` class does not meet the LSP because it can legally replace a `Rectangle` but the overriding that it performs causes an observable change in its behavior (**clients will know it**).
- Most of the times subclasses modify the behavior of superclasses without contradicting the LSP (e.g., a client will expect that the calculation of area in figures varies depending on their type).
- How do we know if an instance of overriding meets or violates the substitution principle? The solution may be in the **design by contract** and the **principle of subcontracting**.

Principle of subcontracting

Design by contract

"*If the preconditions are true when a method is called, then the method guarantees that the postconditions will be true when it does return.*"

- Using this scheme, methods declare preconditions and postconditions that are understood as a contract between that method and the code that uses it.
- **Precondition**: A condition that must be true in order for the method to be executed.
- **Postcondition**: A condition that will be true upon completion of a method.

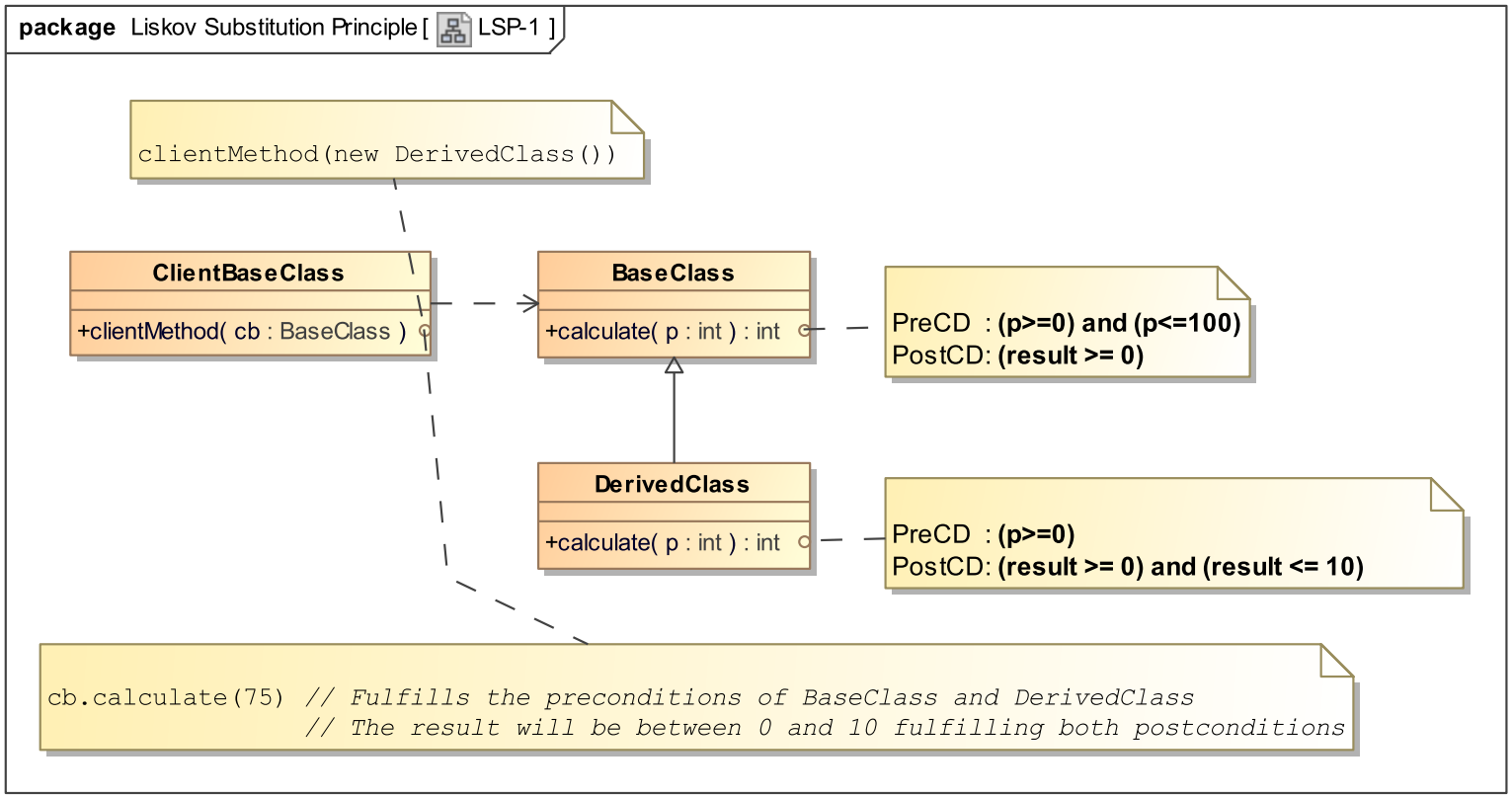
Principle of subcontracting

Principle of subcontracting

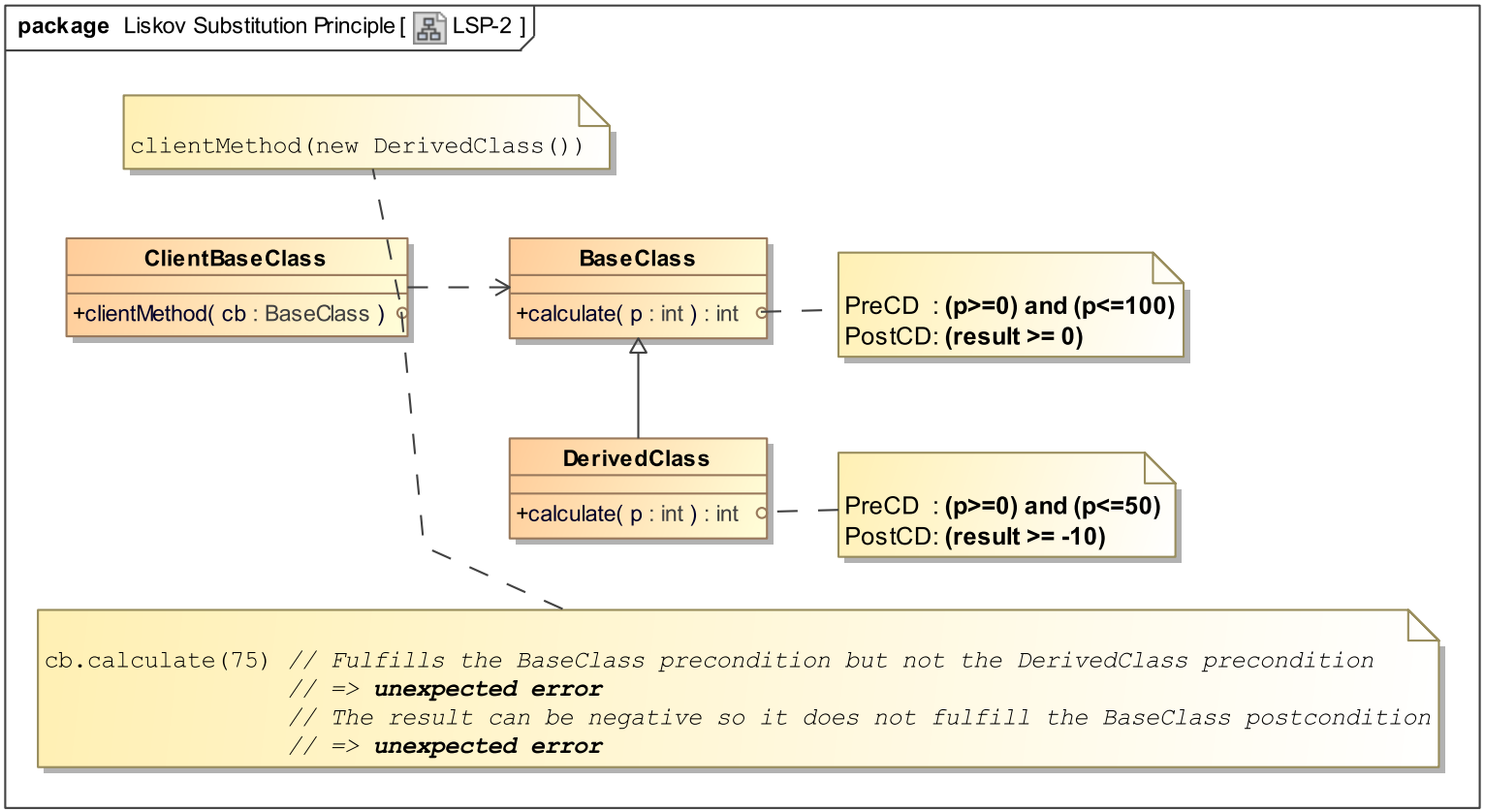
When redefining a routine in a derivative class, you may only replace its precondition by a weaker one, and its postcondition by a stronger one.

- When a class redefines (overwrites) a superclass method we can say that it is subcontracting that method.
- Like all subcontracts, it must guarantee that the contract established in the base class is met.
- That doesn't mean that the contract is the same but that it must follow the principle of subcontracting.

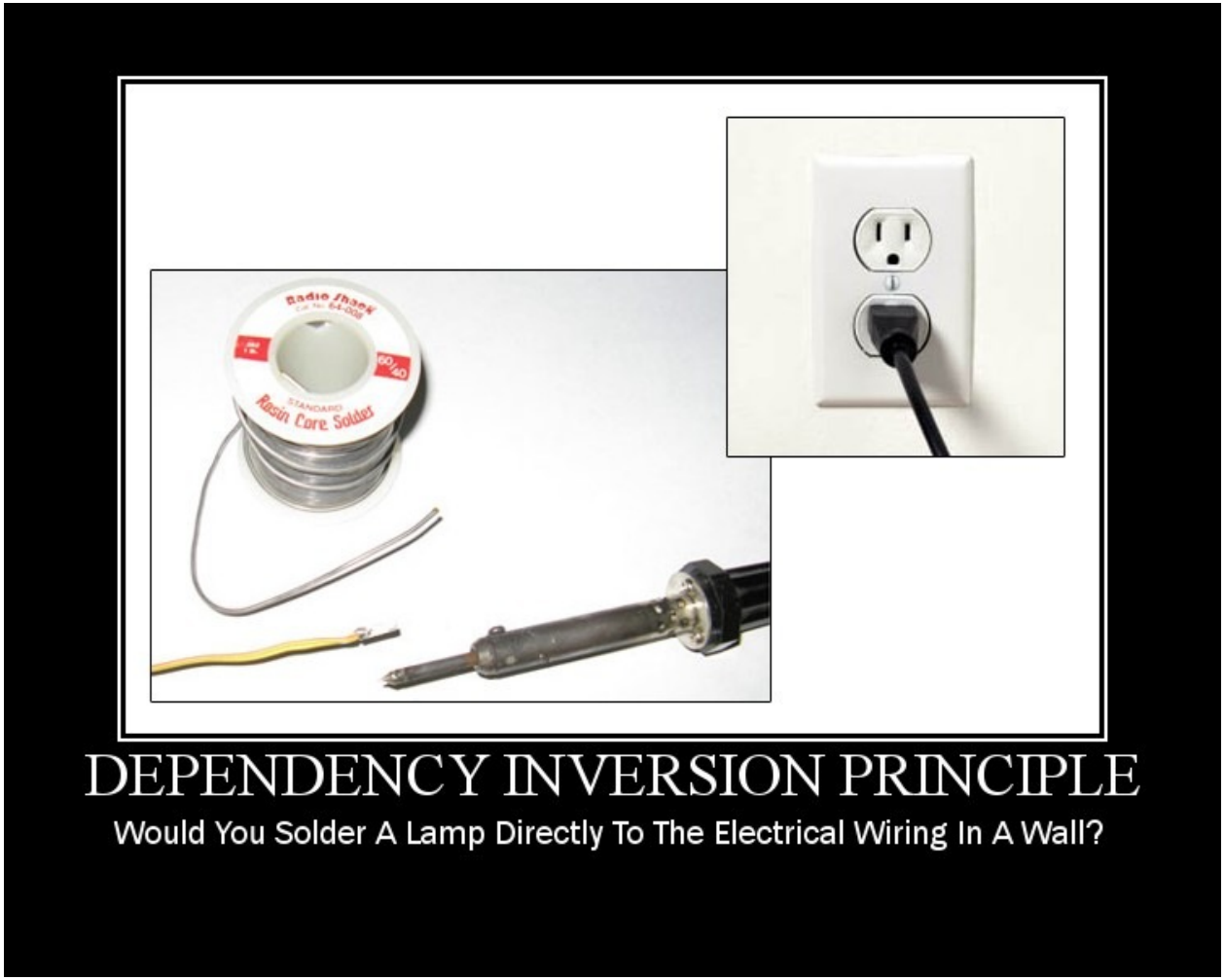
Meeting the principle of subcontracting



Not meeting the principle of subcontracting



The Dependency Inversion Principle (DIP)



The Dependency Inversion Principle (DIP)

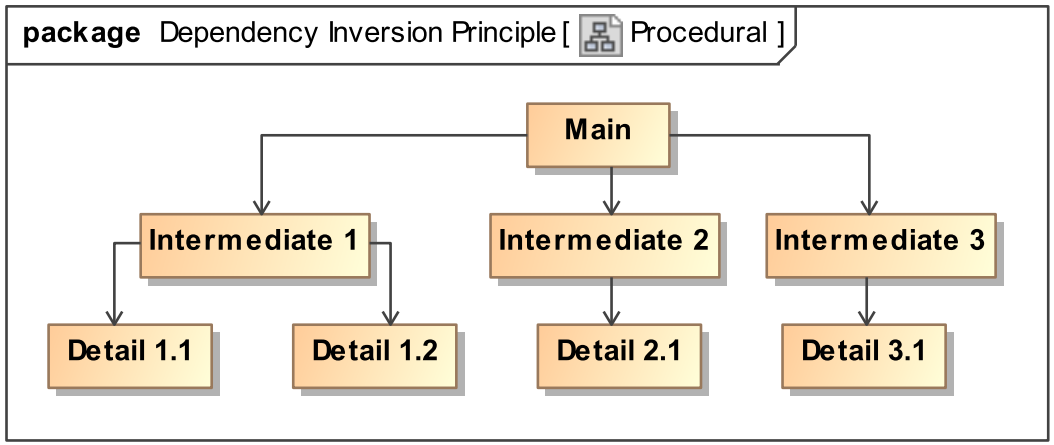
The Dependency Inversion Principle (DIP)

Depend upon abstractions. Do not depend upon concretions.

- Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes.

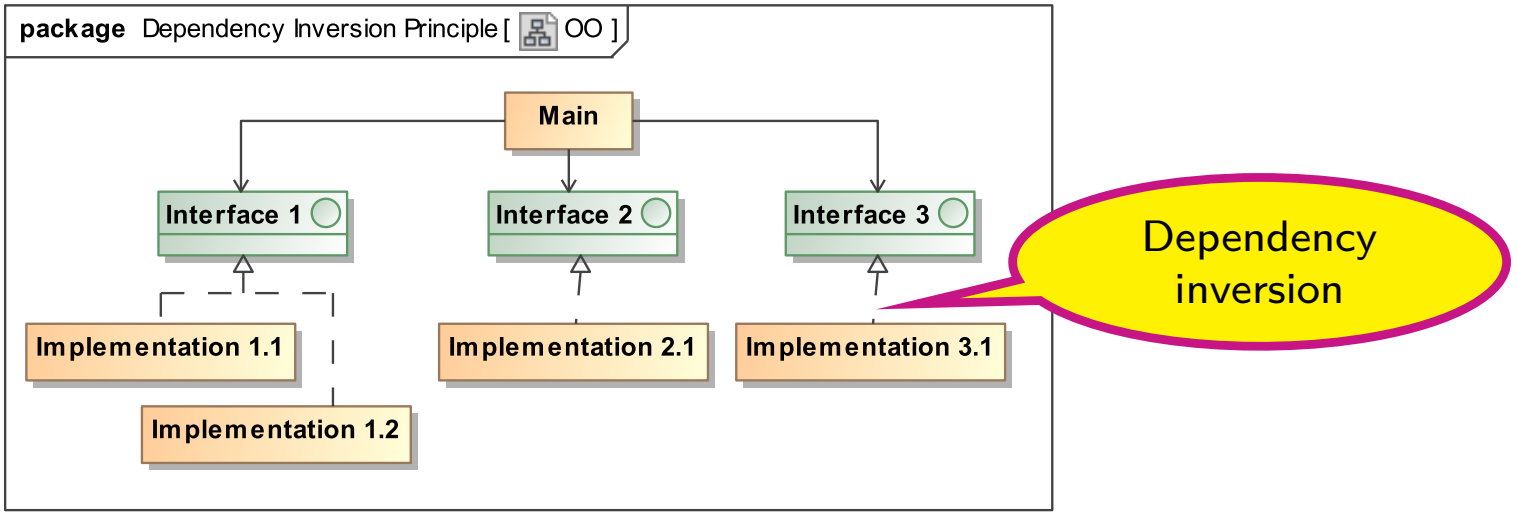
The Dependency Inversion Principle (DIP)

- **Procedural designs** exhibit a particular kind of dependency structure in which high level modules depend upon lower level modules, which depend upon yet lower level modules, etc..



The Dependency Inversion Principle (DIP)

- An **object oriented architecture** shows a very different dependency structure in which the majority of dependencies point towards abstractions
- High level modules should not depend upon low level modules, both should depend upon abstractions. Thus the dependency has been inverted.



The Dependency Inversion Principle (DIP)

Code that does NOT meet the Dependency Inversion Principle

```
ArrayList myList = new ArrayList();
```

- `ArrayList` is a concrete class, not an abstract one.
  - In fact we are being very strict, the `ArrayList` class is a Java API class so it shouldn't be very problematic to rely on it.
  - The dependency inversion principle refers to avoiding relying on concrete *volatile* classes (whose probability of change is high).
- In any case, we can get the advantages of using an interface `List` instead of an abstract class like `ArrayList`.

The Dependency Inversion Principle (DIP)

Using interfaces and not concrete classes

```
List myList = new ArrayList();
```

- Our class declares that it wants to use a list (`List`).
- `ArrayList` is only a concrete implementation of `List` that we are using at this point.

The Dependency Inversion Principle (DIP)

Using interfaces and not concrete classes

```
List myList = new ArrayList();  // We can change the implementation class
                                // List myList = new LinkedList();

// ...

myList._______(); // => Only methods defined in List are allowed here
                  // Changing the implementation does not affect this code
```

- It would be very easy to switch to another implementation of `List`.
- By defining the variable as type `List` we are restricting ourselves to using only the methods declared in `List`, so changing an implementation for another has no effect on our code (we do not depend on the implementing class).

The Dependency Inversion Principle (DIP)

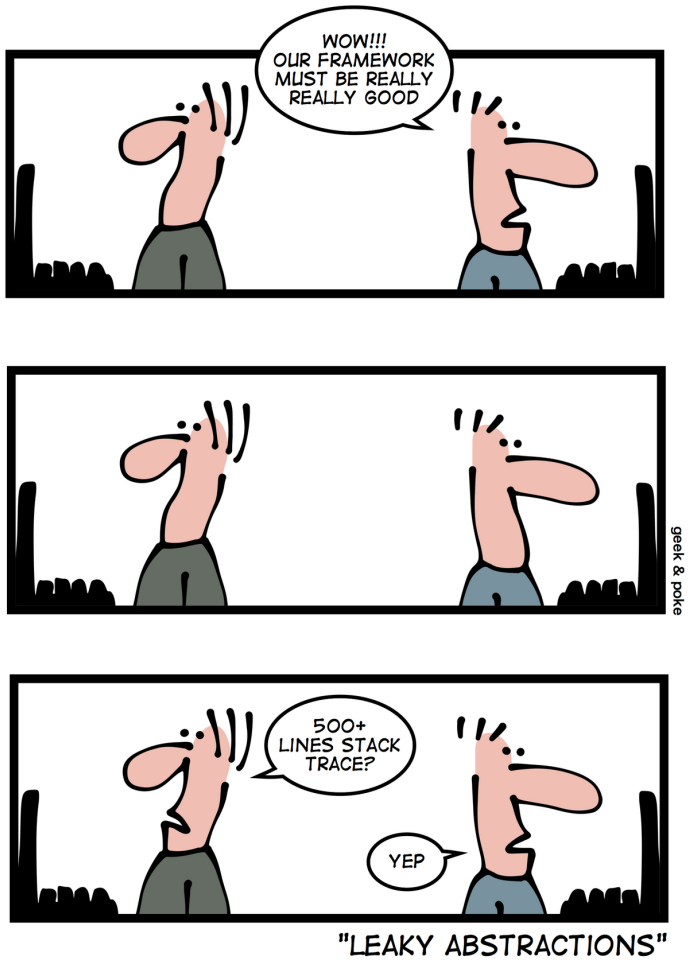
Problems with dependency inversion

```
ArrayList myList1 = new ArrayList();
myList1.ensureCapacity(100); // It works...

List myList2 = new ArrayList();
myList2.ensureCapacity(100); // Compilation error !!!
```

- Since we do not know what implementation of list we are using we are limiting ourselves: we cannot use those specific `ArrayList` features that are not in the `List` interface, such as `ensureCapacity`.
- There is a trade-off between not knowing what the implementation is and trying to be efficient with it.
- **Leaky abstractions**: No matter how hard we try to hide the implementation, it always finds a way to filter through the abstraction.

The Dependency Inversion Principle (DIP)



- "*The Law of Leaky Abstractions*", Joel Spolsky:

  `http://www.joelonsoftware.com/articles/LeakyAbstractions.html`

The Dependency Inversion Principle (DIP)

Using factories to get rid of any dependency on implementations

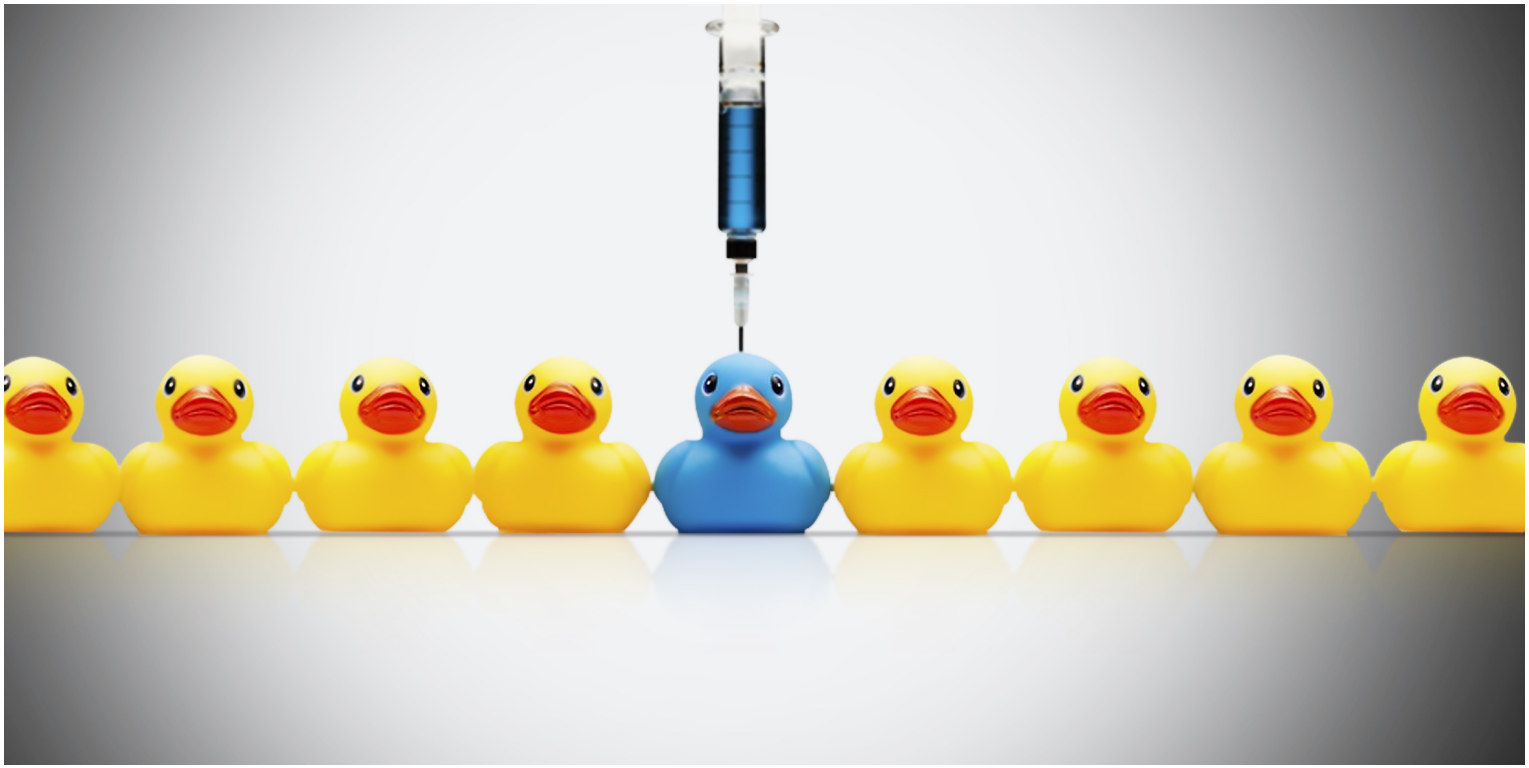
```
List myList = listFactory.createList();
```

- Since we have removed the reference to `ArrayList` in the definition of the variable, why not get rid of it in the `new`?
- The fact is that after a `new` operator you can only specify a concrete class, never an abstract one or an interface.
- The only way to eliminate this dependency is through factory methods $\Rightarrow$ see **Factory Method pattern**

The Dependency Inversion Principle (DIP)

Dependency injection

An injection is the passing of a dependency (a service) to a dependent object (a client) that would use it, not allowing the client to build or find that service.



The Dependency Inversion Principle (DIP)

Example WITHOUT dependency injection

```
public class Client {
    // Internal reference to the service used by this client
    private Service service;

    // Constructor injection
    Client() {
        // Specify a specific implementation in the constructor
        // instead of using dependency injection
        service = new ServiceExample();
    }

    // Method within this client that uses the services
    public String greet() {
        return "Hello " + service.getName();
    }
}
```

The Dependency Inversion Principle (DIP)

Dependency injection with constructors or setter methods

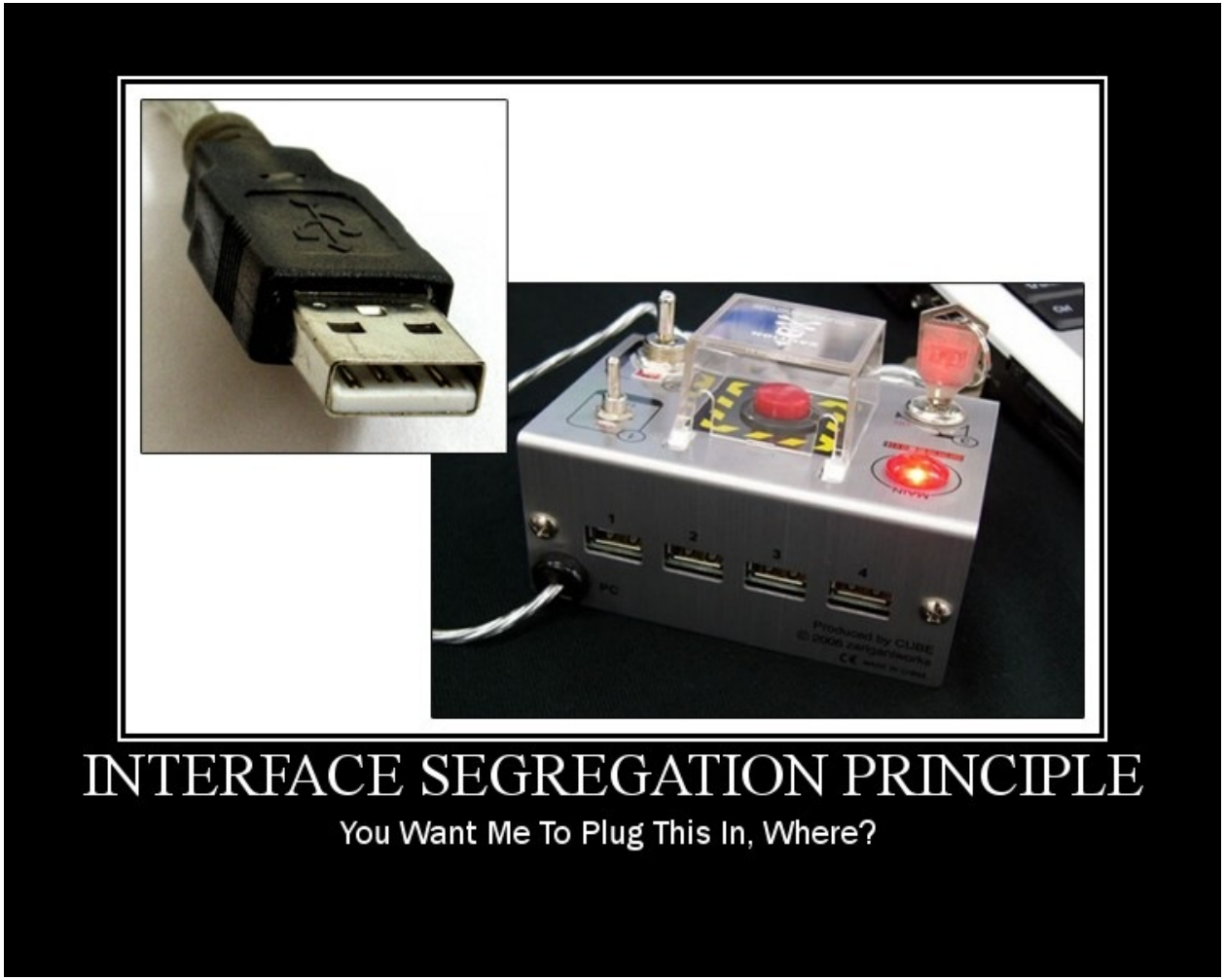
```
public class Client {
    // Internal reference to the service used by this client
    private Service service;

    // Constructor injection
    Client(Service service) {
        // Save the reference to the passed-in service inside this client
        this.service = service;
    }

    // Setter injection
    public void setService(Service service) {
        this.service = service;
    }

    // Method within this client that uses the services
    public String greet() {
        return "Hello " + service.getName();
    }
}
```

The Interface Segregation Principle (ISP)



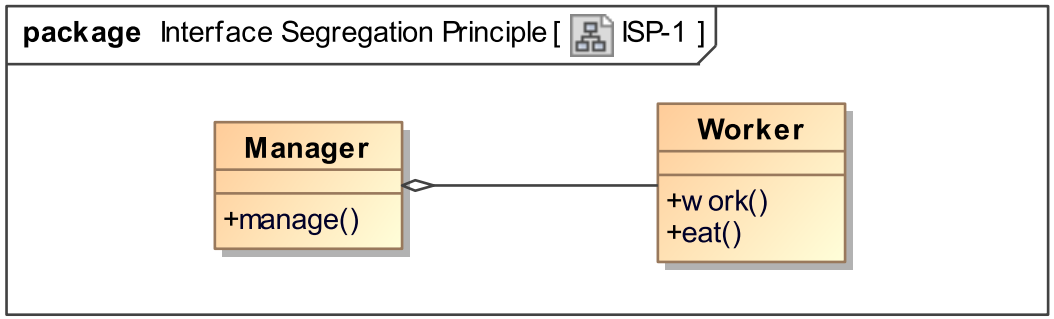
The Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP)

Having many client-specific interfaces is better than having one general-purpose interface.

- Dependency inversion principle said that we have to depend upon abstractions and not upon concretions.
- If we have a *server* class, we can create an interface with all the services offered, but it is more correct to segregate it into several specific interfaces for each particular type of service.

The Interface Segregation Principle (ISP)



Manager uses a service

```
public class Manager {
    List<Worker> workers;

    public void work() {
        for(Worker w : workers)
            w.work();
    }
}
```

Worker offer several services

```
public class Worker {
  public void work() {
      System.out.println("Working...");
  }

  public void eat() {
      System.out.println("Eating...");
  }
}
```

The Interface Segregation Principle (ISP)

`Workable` Interface

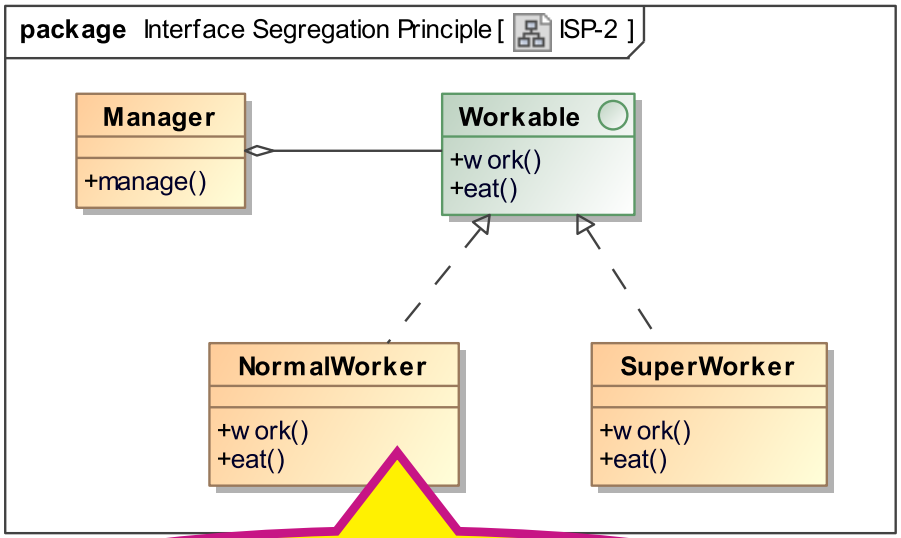
```
public interface Workable {
    void eat();
    void work();
}
```

We extract from `Worker` an interface with the services offered by it

`NormalWorker` Class

```
public class NormalWorker
                implements Workable {
  @Override
  public void work() {
    System.out.println("Working...");
  }

  @Override
  public void eat() {
      System.out.println("Eating...");
  }
}
```



Different classes of workers will implement that interface

The Interface Segregation Principle (ISP)

- **Problem**
  - `Workable` is too general an interface, it performs tasks that are too heterogeneous.
  - `Manager` is only interested in the working capacity of workers, not in their eating habits.
  - There can be workers that do not eat (e.g. robots) but they would be obliged to develop a convenience implementation of the `eat()` method.
- **Solution**
  - Segregate the `Workable` interface separating those services that are not related.
  - The interface Segregation Principle indicates that **interfaces should be designed taking into account how they will be used and not how they will be implemented**.

The Interface Segregation Principle (ISP)

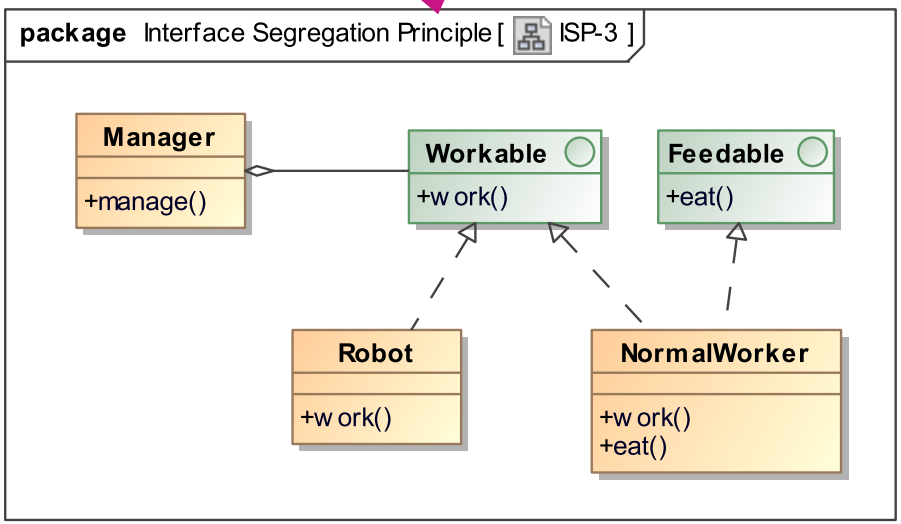
The interface has been split in two: `Workable` and `Feedable`

`Worker` Interface

```
public interface Workable {
    void work();
}
```

`Eater` Interface

```
public interface Feedable {
    void eat();
}
```



The Interface Segregation Principle (ISP)

Classes will implement only those interfaces in which they are interested.

`NormalWorker` class

```
public class NormalWorker
        implements Workable, Feedable {
  @Override
  public void work() {
    System.out.println("Working...");
  }

  @Override
  public void eat() {
      System.out.println("Eating...");
  }
}
```

`Robot` class

```
public class Robot implements Workable {
  @Override
  public void work() {
      System.out.println
          ("Working CEASELESSLY...");
  }
}
```

Problems with interfaces

- **Interface cannot grow**
  - If we add new methods we will break the code of the implementing classes because they are not implementing all the methods of the interface.
- **Example:**
  - An enterprise can allow their workers to dedicate a 20 % of their time to personal projects, but you cannot add a `personalProject()` method in the interface without forcing the creation of new versions of `NormalWorker`, `SuperWorker` and `Robot`.
  - We can try to include in advance all possible methods in an interface to avoid this problem, but we would be going **against the Interface Segregation Principle**, and also it is very difficult to predict future needs.

Problems with interfaces

- **We cannot define optional methods in the interfaces**
  - All the methods defined in the interfaces must be implemented by the implementing classes.
- **Example:**
  - The Java API collection allows to define optional methods in the documentation (e.g., the `remove()` method in the `Iterator` interface).
  - This means that it is acceptable to develop an implementation that throws an exception (such as `UnsupportedOperationException`)

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`hasNext()` Returns `true` if the iteration has more elements.
`E`	`next()` Returns the next element in the iteration.
`void`	`remove()` Removes from the underlying collection the last element returned by this iterator (optional operation).

Interfaces and default methods in Java

Default methods in interfaces (since Java 8)

Methods in interfaces that allow the inclusion of a default implementation for them.

- They are defined using the keyword `default`.
- They solve the problem of growing interfaces and let you define truly optional methods (as long as they have a default implementation).

Adding the `personalProject()` method to `Workable`

```
public interface Workable {
    void work();

    default void personalProject() {
        System.out.println("I have no personal project");
    }
}
```

Interfaces and default methods in Java

`NormalWorker` does not implement `personalProject()`...

```
public class NormalWorker implements Workable, Feedable {
    @Override
    public void work() {
        System.out.println("Working...");
    }

    @Override
    public void eat() {
        System.out.println("Eating...");
    }
}
```

... but it inherits its default implementation.

```
// ...
NormalWorker nw = new NormalWorker();
nw.personalProject();  // shows "I have no personal project"
// ...
```

Interfaces and default methods in Java

`SuperWorker` redefines `personalProject()`

```
public class SuperWorker implements Workable, Feedable {

    @Override
    public void work() {
        System.out.println("Working A LOT...");
    }

    @Override
    public void come() {
        System.out.println("Eating VERY LITTLE...");
    }

    @Override
    public void personalProject() {
        System.out.println("Achieve world peace...");
    }
}
```

Interfaces and default methods in Java

- Since Java 8, the `remove()` method of the `Iterator` interface has a default implementation that throws an exception if called.
- So now it is really an **optional** method in the interface. The only abstract methods whose implementation is required are `hasNext()` and `next()`.

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
`boolean`	**`hasNext`**`()` Returns `true` if the iteration has more elements.
`E`	**`next`**`()` Returns the next element in the iteration.

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
`default void`	**`forEachRemaining`**`(`**`Consumer`**`<? super E> action)` Performs the given action for each remaining element until all elements have been processed or the action throws an exception.
`default void`	**`remove`**`()` Removes from the underlying collection the last element returned by this iterator (optional operation).

Interfaces and default methods in Java

- **What happens with code conflicts?**
  - If we inherit from two interfaces that have different default implementations for the same methods we will obtain a compiler error: "`interface X inherits unrelated defaults for Y from types M and N`".
  - This can be solved by forcing the conflicting class to implement again the problematic method.
  - It can be done by using `super` to choose one of the default implementations from one interface: e.g. `SecondInterface.super.doSomething();`

Interfaces and static methods in Java

Static methods in interfaces (since Java 8)

Static methods that reside in an interface instead of in a class.

- Defined using the `static` keyword.
- They represent *helper methods* related to the interface.
- They are invoked just like any other static method, i.e., using the name of the class (in this case the interface) in which they are defined.
- As they are in an interface they are implicitly public, so the `public` specifier can be omitted.

Interfaces and static methods in Java

`Workable` includes a static method `describeWorker`...

```
public interface Workable {
    void work();
    default void personalProject() { /* ... */ }

    static void describeWorker(Workable w) {
        System.out.print("I work in: ");
        w.work();
        System.out.print("And my personal project is: ");
        w.personalProject();
    }
}
```

... which acts as a regular static method.

```
NormalWorker nw = new NormalWorker();
SuperWorker sw = new SuperWorker();

Workable.describeWorker(nw);
Workable.describeWorker(sw);
```

Interfaces and static methods in Java

- Simple interfaces such as `Comparator` with only two abstract methods...

All Methods	Static Methods	Instance Methods	Abstract Methods	Default Methods

Modifier and Type	Method and Description
int	**compare**(T o1, T o2) Compares its two arguments for order.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this comparator.

- ...are now riddled with static methods as shown in the following snapshot of its *javadoc*:

Interfaces and static methods in Java

All Methods | **Static Methods** | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
`static <T,U extends Comparable<? super U>> Comparator<T>`	**`comparing`**`(Function<? super T,? extends U> keyExtractor)` Accepts a function that extracts a **`Comparable`** sort key from a type `T`, and returns a `Comparator<T>` that compares by that sort key.
`static <T,U> Comparator<T>`	**`comparing`**`(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)` Accepts a function that extracts a sort key from a type `T`, and returns a `Comparator<T>` that compares by that sort key using the specified **`Comparator`**.
`static <T> Comparator<T>`	**`comparingDouble`**`(ToDoubleFunction<? super T> keyExtractor)` Accepts a function that extracts a `double` sort key from a type `T`, and returns a `Comparator<T>` that compares by that sort key.
`static <T> Comparator<T>`	**`comparingInt`**`(ToIntFunction<? super T> keyExtractor)` Accepts a function that extracts an `int` sort key from a type `T`, and returns a `Comparator<T>` that compares by that sort key.
`static <T> Comparator<T>`	**`comparingLong`**`(ToLongFunction<? super T> keyExtractor)` Accepts a function that extracts a `long` sort key from a type `T`, and returns a `Comparator<T>` that compares by that sort key.
`static <T extends Comparable<? super T>> Comparator<T>`	**`naturalOrder`**`()` Returns a comparator that compares **`Comparable`** objects in natural order.
`static <T> Comparator<T>`	**`nullsFirst`**`(Comparator<? super T> comparator)` Returns a null-friendly comparator that considers `null` to be less than non-null.
`static <T> Comparator<T>`	**`nullsLast`**`(Comparator<? super T> comparator)` Returns a null-friendly comparator that considers `null` to be greater than non-null.
`static <T extends Comparable<? super T>> Comparator<T>`	**`reverseOrder`**`()` Returns a comparator that imposes the reverse of the *natural ordering*.

Interfaces and private methods in Java

Private methods in interfaces (since Java 9)

Private methods are now allowed in interfaces. They can be called by the code of default or static methods that reside in said interface.

- Once the taboo of including code in the interfaces was broken, it did not make sense to forbid the inclusion of private methods in those interfaces.
- Private methods can be instance methods or static methods.

Interfaces and private methods in Java

`Workable` with private methods:

```
public interface Workable {
    void work();

    private String noProject() {
        return "I have no personal project";
    }
    default void personalProject() {
        System.out.println(noProject());
    }

    private static void print(String s) {
        System.out.print(s);
    }
    static void describeWorker(Workable w) {
        print("I work in: ");
        w.work();
        print("And my personal project is: ");
        w.personalProject();
    }
}
```

To sum up...

- **Since Java 9 in an interface you can have:**
  - Constant and static attributes.
  - Abstract methods.
  - Default methods.
  - Static methods.
  - Private methods.
  - Private Static methods.

Table of Contents

Forms of inheritance

They are not mutually exclusive and it is not intended to be an exhaustive classification.

- **Correct Forms of Inheritance**
  - Inheritance for specialization
  - Inheritance for specification
  - Inheritance for extension
- **Incorrect Forms of Inheritance**
  - Inheritance for construction
  - Inheritance for variance
  - Inheritance for limitation
  - Inheritance for combination

Inheritance for Specialization

Inheritance for Specialization

The new class is a specialized variety of the parent class but satisfies the specifications of the parent in all relevant respects.

- Probably the most common use of inheritance.
- The principle of substitutability is explicitly upheld.
- **Example**:
  - Class `Personnel` declared a `salary()` method that was common for all the employees.
  - Some employees overwrite that method to offer a specialized version of it.

Inheritance for Specification

Inheritance for Specification

Parent class does not implement actual behavior but merely defines the behavior that must be implemented in child classes.

- It is another frequent use for inheritance.
- In Java is implemented through interfaces and abstract classes.
- It is used to guarantee that classes maintain a certain common interface.
- **Example**:
  - Class `Figure` defined two abstract methods (`area` and `perimeter`) that must be implemented by all the subclasses of `Figure` (`Circle`, `Rectangle`, etc.)

Inheritance for Extension

Inheritance for Extension

It occurs when a child class only adds new behavior to the parent class and does not modify or alter any of the inherited methods.

- The subclass is a specialized version of the superclass that does not overwrite its methods but simply adds new ones.
- As the functionality of the parent remains available and untouched, subclassification for extension does not contravene the principle of substitutability.
- It could be considered whether using composition would be more appropriate.
- **Example**:
  - Class `Point` defines a point in a two-dimensional space.
  - Class `ColoredPoint` inherits from `Point` and adds a new `color` property with its associated methods.

Inheritance for construction

Inheritance for construction

A class inherits its desired functionality from a parent class but they fail to share an *is-a* relationship.

- It is used from a pragmatic point of view to reuse code and simplify the implementation of new classes.
- Composition is a better alternative $\Rightarrow$ **Favor composition over inheritance**.
- It often directly breaks the principle of substitutability.
- **Examples**:
  - In a pinball game we have a `Ball` class and a `Hole` class.
  - `Hole` inherits from `Ball` although it does not have an `is-a` relationship but the behavior needed for the `Hole` abstraction matches the behavior of the class `Ball` (e.g. detection of hits).
  - Another example: Class `Stack` inherits from `Vector`. It does not break the principle of substitutability because it is inheritance for extension.

Inheritance for variance

Inheritance for variance

Occurs when two or more classes have similar implementations but do not seem to possess any hierarchical relationships between them. Arbitrarily, one of the two classes is selected to be the parent, with the common code being inherited by the other.

- A better alternative is to factor out the common code into an abstract class, and have both classes inherit from this common ancestor.
- **Example**:
  - Class `Square` could be made a subclass of class `Rectangle`, or the other way around.
  - Probably, the two of them should be subclasses from a common `Figure` class.

Inheritance for limitation

Inheritance for limitation

The behavior of the subclass is smaller or more restrictive than the behavior of the parent class.

- For example, a subclass overrides a method of the superclass to throw an exception.
- It is an explicit contravention of the principle of substitutability and should be avoided whenever possible.
- **Example**:
  - Class `Stack` protecting its abstraction overriding the mutable methods of `Vector`.

Class `Stack` limiting the inheritance from `Vector`

```
public void removeElementAt (int index) {
    throw new java.lang.UnsupportedOperationException("removeElementAt");
}
```

Inheritance for combination

Inheritance for combination

Occurs when the child class inherits features from more than one parent class (multiple inheritance).

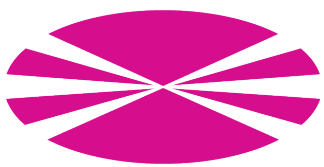
- Multiple inheritance allows more design possibilities but increases the complexity and reduces the understandability of code.
- Increases the probability of using inheritance for construction (inheritance without an `is-a` relationship).
- Multiple inheritance is not directly supported by Java, but it can be simulated using implementation of interfaces, which is a simpler and more adequate form of inheritance for combination.

Unit 5: Design Principles

Software Design (614G01015)

David Alonso Ríos
Eduardo Mosqueira Rey (Coordinator)

Department of Computer Science, Faculty of Computer Science



UNIVERSIDADE DA CORUÑA