Software Design & Programming Techniques

Class Design Principles

Prof. Dr-Ing. Klaus Ostermann

Based on slides by Prof. Dr. Mira Mezini

Class Design Principles

- 2.1 About Class Design Principles (CDPs)
- 2.2 Single Responsibility Principle (SRP)
- 2.3 The Open-Closed Principle (OCP)
- 2.4 Liskov Substitution Principle (LSP)
- 2.5 Interface Segregation Principle (ISP)
- 2.6 Dependency Inversion Principle (DIP)

2.1 About Class Design Principles (CDPs)

CDPs are Heuristics

- **CDPs state desired properties of class designs.**
  E.g. “a class should have only one responsibility”

- **CDPs are heuristics.**
  They serve as guides to good designs, not as absolute criteria to judge the quality of designs.

- **CDPs are somewhat vague and ambiguous.**
  This does not make them useless.

CDPs are About Ease of Use and Change

- **CDPs help making a class design usable for clients.**
  We think about how our classes are used by other classes.

- During its lifetime of a software its class design changes constantly.
  This is a consequence of requirement changes which is the rationale for conducting an iterative design process.

- CDPs do not only judge the current state of the code
- **They give an understanding of how well the code will work under the effect of change.**
  Especially whether and how changes will affect client classes.

Class Design Principles do not aim for code that works, but for code that can efficiently be worked on!

S.O.L.I.D. Principles

In this course, we will examine the S.O.L.I.D Principles:

- Single Responsibility Principle (SRP)
- Open-closed Principle (OCP)
- Liskov Substitution Principles (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

2.2 Single Responsibility Principle (SRP)

A class (*) should have only one reason to change.

(*)More generally, every abstraction such as method, function, datatype,module

2.2 Single Responsibility Principle (SRP)

- 2.2.1 Responsibility and Cohesion
- 2.2.2 Introduction to SRP by Example
- 2.2.3 The Employee Example
- 2.2.4 The Modem Example
- 2.2.5 To Apply or Not to Apply
- 2.2.6 SRP, more generally
- 2.2.7 The Smart Home Example
- 2.2.8 Takeaway

2.2.1 Responsibility and Cohesion

- A class is assigned the responsibility to know or do something
  Class `PersonData` is responsible for knowing the data of a person.
  Class `CarFactory` is responsible for creating `Car` objects.

- **A responsibility is an axis of change.**
  If new functionality must be achieved, or existing functionality needs to be changed, the responsibilities of classes must be changed.

- A class with only one responsibility will have only one reason to change!

Responsibility and Cohesion

- **Cohesion measures the degree of togetherness among the elements of a class.**
  It measures the extent to which operations and data within a class belong to a common concept this class is representing.

- **Cohesiveness is not an absolute predicate** on classes/designs (measured by binary values).

- In a class with very high cohesion every element is part of the implementation of one concept.

  The elements of the class work together to achieve one common functionality.

- A class with high cohesion implements only one responsibility (only few responsibilities)!

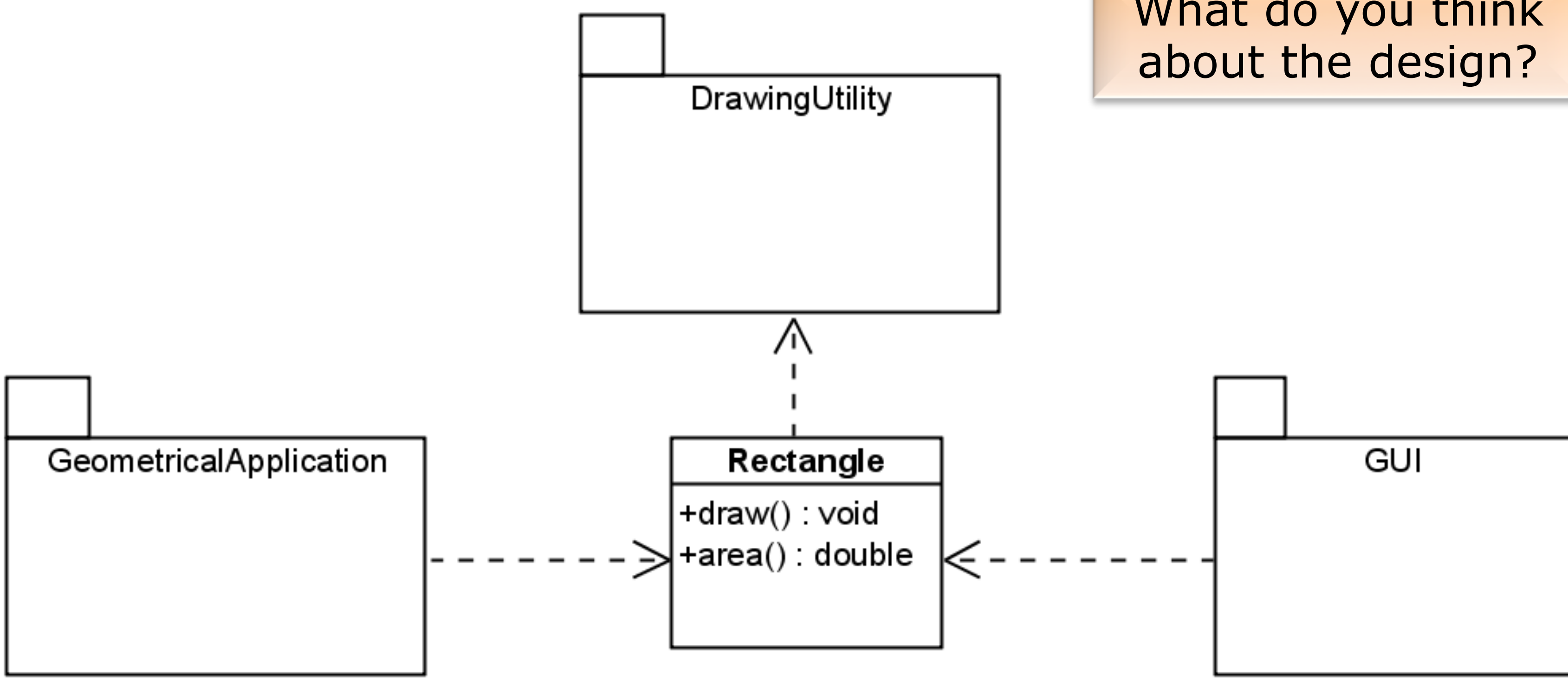
  Therefore, **a class with low cohesion violates SRP**.

2.2.2 Introduction to SRP by Example

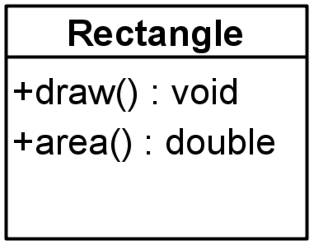
- Consider the following design, depicted in UML.

- `GUI` package uses `Rectangle` to draw rectangle shapes in the screen. `Rectangle` uses `DrawingUtility` to implement `draw`.
- `GeometricApplication` is a package for geometrical computations which also uses `Rectangle (area())`.

What do you think about the design?



Problems of `Rectangle`



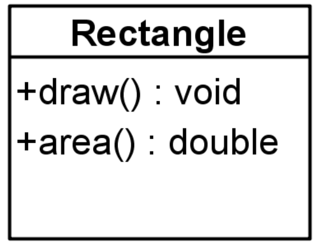
- `Rectangle` has multiple responsibilities!
  1) Geometrics of rectangles represented by the method `area()`
  2) Drawing of rectangles represented by the method `draw()`

- `Rectangle` has low cohesion!

  Geometrics and drawing do not naturally belong together.

OK, but why is this a problem?

Problems of `Rectangle`

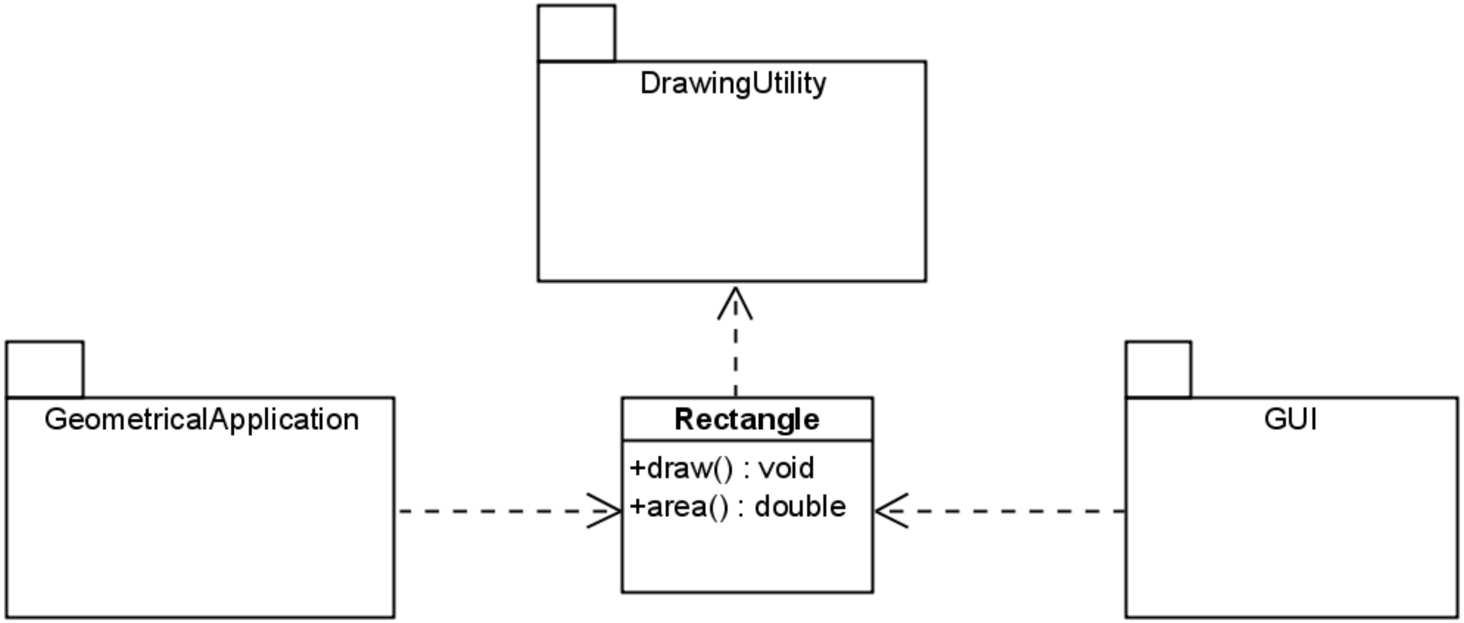


- **`Rectangle` is hard to use!** It has multiple reasons to change.
  Even if we want to use only one of its responsibilities, we must depend on both of them.
  We inherit the effects of changes along every possible axis of change (= responsibility)!

- **`Rectangle` is easily misunderstood!**
  It is not only a representation of a rectangle shape, but also part of a process concerned with drawing rectangle shapes in the screen.
  It was not created as a representation of a certain concept, but as a bundle of needed functionality without careful consideration of their cohesion.

Undesired Effects of Change

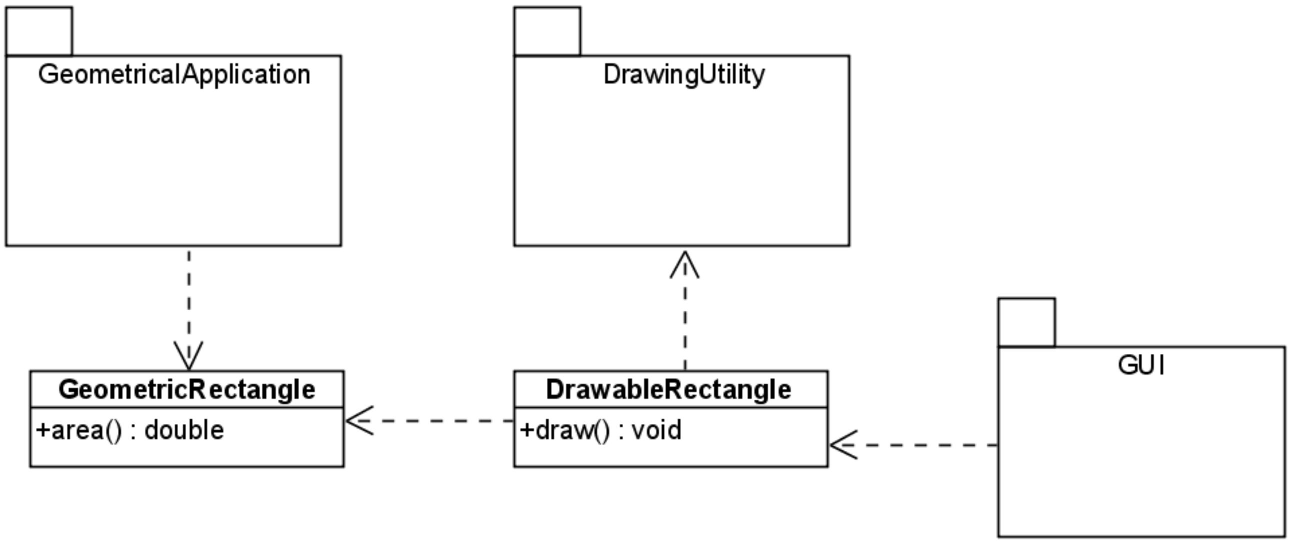
- Unnecessary dependency between `GeometricApplication` and `DrawingUtility` (`DrawingUtility` classes have to be deployed along with `Rectangle`) even if we only want to use the geometrical functions of rectangles.



- **Problem**: If drawing functionality changes in the future, we need to retest `Rectangle` also in the context of `GeometricalApplication`!

A SRP-Compliant Design

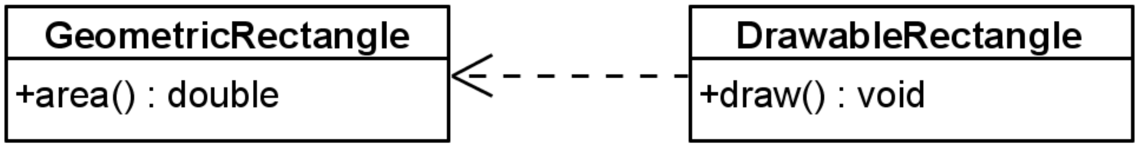
- Split `Rectangle` according to its responsibilities.
  `GeometricRectangle` models a rectangle by its geometric properties.
  `DrawableRectangle` models a graphical rectangle by its visual properties.



- `GeometricalApplication` uses only `GeometricRectangle`.
  It only depends on the geometrical aspects.

- `GUI` uses `DrawableRectangle` and indirectly `GeometricRectangle`.
  It needs both aspects and therefore has to depend on both.

Two Classes with High Cohesion



- **Both classes can be (re)used easily!**
  Only changes to the responsibilities we use will affect us.

- **Both classes are easily understood!**
  Each implements one concept.
  `GeometricRectangle` represents a rectangle shape by his size.
  `DrawableRectangle` encapsulates a rectangle with visual properties.

Takeaway so Far

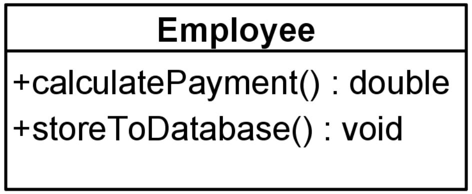
A class should have only one reason to change.

- Applying SRP maximizes the cohesion of classes.
- Classes with high cohesion:
  - can be reused easily,
  - are easily understood,
  - protect clients from changes, that should not affect them.

What’s Next?

- Next, more scenarios are discussed, where we might want to apply SRP.
- Goal:
  - Get a better feeling as when to apply SRP and when not.
  - Get to know some issues related to applying SRP in terms of the mechanisms available for doing so.

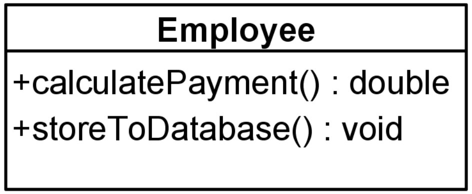
2.2.3 The Employee Example



- Consider the class `Employee` which has two responsibilities:
  1) Calculating the employees pay.
  2) Storing the employee data to the database.

Should we split the responsibilities of this class?

`Employee` Represents a Typical SRP-Violation

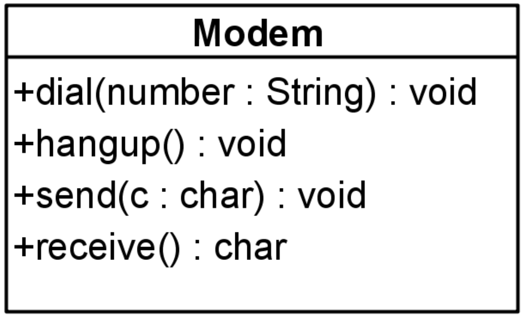


- Calculating the payment of an employee is part of the **business rules**.
  It corresponds to a real-world concept the application shall implement.
- Storing the employee information in the database is a **technical aspect**.
  It is a necessity of the IT architecture that we have selected; does not correspond to a real-world concept.

- **Mixing business rules and technical aspects is calling for trouble!**
  From experience we know that both aspects are extremely volatile.

Most probably we should split in this case.

2.2.4 The Modem Example



- The class `Modem` has also two responsibilities:
  1) Connection management (`dial` and `hangup`)
  2) Data communication (`send` and `receive`)

Should we split the responsibilities of this class?

It depends...

To Split or Not to Split `Modem`?

Break down the question to:

- Do we expect connection management and data communication to constitute **different axes of change?**

  Do we expect them to change together, or independently.

- Will these responsibilities be **used by different clients?**

- Do we plan to provide **different configurations** of modems **to different customers?**

To Split or Not to Split `Modem`?

- **Split if**:
  - Responsibilities will change separately.
  - Responsibilities are used / will probably be used by different clients.
  - We plan to provide different configurations of modems with varying combinations of responsibilities (features).
- **Do not split if**:
  - Responsibilities will only change together, e.g. if they both implement one common protocol.
  - Responsibilities are used together by the same clients.
  - Both correspond to non optional features.

The `Modem` Example

Modem
+dial(number : String) : void +hangup() : void +send(c : char) : void +receive() : char

- The class `Modem` has also two responsibilities:
  1) Connection management (`dial` and `hangup`)
  2) Data communication (`send` and `receive`)

Should we split the responsibilities of this class?

Probably not ...

2.2.5 To Apply or Not to Apply

- Decide based on the nature of responsibilities:
  changed together / not used together
  used together / not used together
  optional / non optional

- **Only apply a principle, if there is a symptom!**
  An axis of change is an axis of change only, if the change actually occurs.

2.2.6 SRP, more generally

- More generally, the SRP applies to any kind of programming abstraction
  - Classes, methods, functions, packages, data types, …

- Important: The SRP must be applied with respect to the right level of abstraction
  - High-level abstractions → high-level responsibilites
  - Otherwise it seems contradictory that, say, a package (a collection of classes designed according to SRP) can have only a single responsibiltiy

Strategic Application

- Choose the kinds of changes to guide SRP application.
  - Guess the most likely kinds of changes.
  - Separate to protect from those changes.

- Prescience (*Voraussicht*) derived from experience:
  - Experienced designer hopes to know the user and an industry well enough to judge the probability of different kinds of changes.
  - Invoke SRP against the most probable changes.

- After all: **Be agile**.

  Predictions will often be wrong.

  Wait for changes to happen and modify the design when needed.
  Simulate change.

Simulate Change

- **Write tests first**.
  - Testing is one kind of usage of the system
  - Force the system to be testable; changes in testability will not be surprising later.
  - Force developers to build the abstractions needed for testability; protect from other kind of changes as well.

- Use **short development (iteration) cycles**
- **Develop features before infrastructure**; show them to stakeholders
- Develop the **most important features first**
- **Release software early and often**; get it in front of users and customers as soon as possible

2.2.7 The Smart Home Example

- Consider the case of a smart home provider.
- A smart home has many features that are controlled electronically.
- The provider wants to sell several configurations of a smart home, each with a specific selection of features.
- Let us judge a typical OO design of a smart home...

Typical OO Design

```
abstract class Location {
   abstract List<Shutter> shutters();
   abstract List<Light> lights(); ...
}
class Room extends Location {
   List<Light> lights;
   List<Light> lights() { return lights; }
   List<Shutter> shutters; ...
}
abstract class CompositeLocation extends Location {
   abstract List<? extends Location> locations();
   List<Light> lights() { ... }
   List<Shutter> shutters() { ... } ...
}
class Floor extends CompositeLocation {
    List<Room> rooms;
    List<? extends Location> locations() { return rooms; } ...
}
class House extends CompositeLocation {
    List<Floor> floors;
    List<? extends Location> locations() { return floors; } ...
}
```

Shutter control feature

Lighting control feature

To Split or not to Split

Should we split the responsibilities in the smart home scenario?

Yes, if we want to be able to make functional packages - heating control, lightening control, security, etc. - optional

The question is how?

How to Split Responsibilities?

Ideally would like to have several versions of class definitions - one per responsibility - which can be mixed and matched on-demand.

Base configuration

```
abstract class Location {
}
abstract class CompositeLocation extends Location {
   abstract List<? extends Location> locations();
}
class Room extends Location {
}
class Floor extends CompositeLocation {
   List<Room> rooms;
   List<? extends Location> locations() { return rooms; }
}
class House extends CompositeLocation {
   List<Floor> floors;
   List<? extends Location> locations() { return floors; }
}

...
House house = new House();
House house() { return house; }
...
```

How to Split Responsibilities?

Ideally would like to have several versions of class definitions - one per responsibility - which can be mixed and matched on-demand.

Base configuration

```
abstract class Location {
}
abs                    ion extends Location {

}
cla
}
cla



}
cla



}

...
House hous
House hous
...
```

Shutter control

```
abstract class Location {
  a                    > shutters();
}

abst
  L
}

clas
  L
  L
}
```

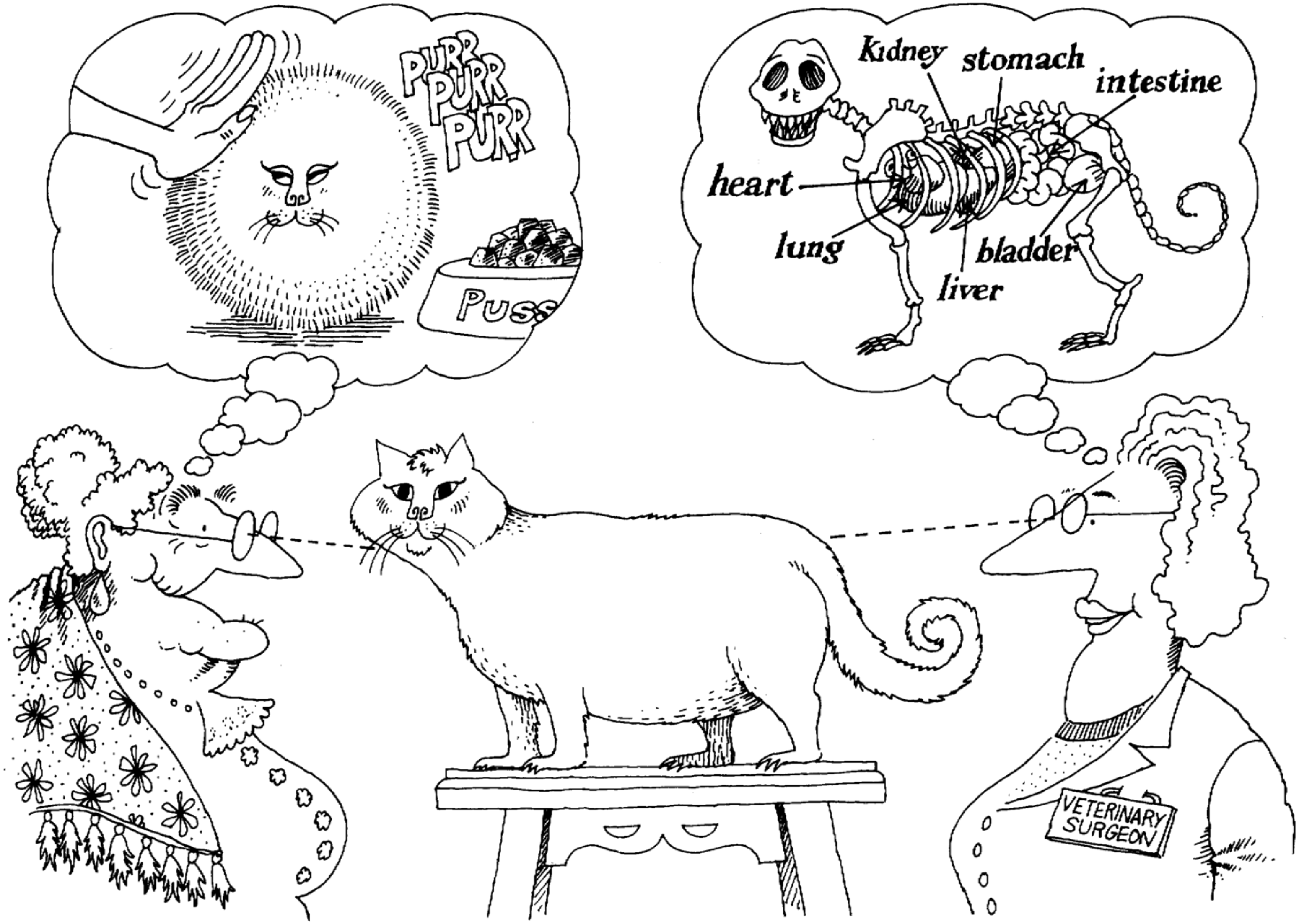
Lighting control

```
abstract class Location {
  abstract List<Light> lights();
}

abstract class CompositeLocation {
  List<Light> lights() { ... }
}

class Room {
  List<Light> lights;
  List<Light> lights() { return lights; }
}
```

View-Specific Responsibilities



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Splitting by Inheritance

```
abstract class LocationWithShutters extends Location {

}

ab
Co

}

..
```

```
abstract class LightedLocation extends Location {
   abstract List<Light> lights();
   ...
}
abstract class LightedCompositeLocation extends CompositeLocation {
   List<Light> lights() {
      List<Light> lights = new ArrayList<Light>();
      for (Location child : locations()) {
                 lights.addAll(child.lights())}
      return lights;
   }
}
class LightedRoom extends Room {
   List<Light> lights;
   List<Light> lights() { return lights; }
}
class LightedFloor extends ...
class LightedHouse extends ...

...
House house = new LightedHouse();
...
```

What do you think about the design?

Splitting by Inheritance: Problems

```
abstract class LightedLocation extends Location {
   abstract List<Light> lights();
   ...
}
abstract class LightedCompositeLocation extends CompositeLocation {
   List<Light> lights() {
      List<Light> lights = new ArrayList<Light>();
      for (Location child : locations()) {
                 lights.addAll(child.lights())}
      return lights;
   }
}
class LightedRoom extends Room {
   List<Light> lights;
   List<Light> lights() { return lig
}
class LightedFloor extends ...
class LightedHouse extends ...

...
House house = new LightedHouse();
...
```

Classes are not replaced in type references.

`child` is of type `Location`. Call is invalid. Need a cast. This is unsafe because the design cannot guarantee that only `LightedLocations` are added as children to a `LightedCompositeLocation`.

Splitting by Inheritance: Problems

```
abstract class LightedLocation extends Location {
    abstract List<Light> lights();
    ...
}
abstract class LightedCompositeLocat                                         {
    List<Light> lights() {
        List<Light> lights = new Array
        for (Location child : locatior
                    lights.addAll(chilc
        return lights;
    }
}
class LightedRoom extends Room {
    List<Light> lights;
    List<Light> lights() { return lig
}
class LightedFloor extends ...
class LightedHouse extends ...

...
House house = new LightedHouse();
...
```

Classes are not replaced in inheritance relationships.

What should `LightedFloor`, `LightedHouse` inherit from?

Inherit from `Floor/House` and duplicate lightening functionality.

Alternatively, inherit form `LightedCompositeLocation` and duplicate `Floor/House` functionality.

None is satisfactory.

Splitting by Inheritance: Problems

```
abstract class LightedLocation extends Location {
   abstract List<Light> lights();
   ...
}
abstract class LightedCompositeLocation extends CompositeLocation {
   List<Light> lights() {
      List<Light> lights = new ArrayList<Light>();
      for (Location child : locations()) {
                 lights.addAll(child.lights())}
      return lights;
   }
}
class LightedRoom extends Room {
   List<Light> lights;
   List<Light> lights() { return lig
}
class LightedFloor extends ...
class LightedHouse extends ...

...
House house = new LightedHouse();
...
```

We must ensure that the new classes are instantiated whenever the old ones were instantiated in the base configuration.

Splitting by Inheritance: Problems

Moreover, the composition is not easy even with multiple inheritance.

Later for a more elaborated discussion.

2.2.8 Takeaway

A class should have only one reason to change.

- Applying SRP maximizes the cohesion of classes.
- Classes with high cohesion
  - can be reused easily,
  - are easily understood,
  - protect clients from changes, that should not affect them.

- But be strategic in applying SRP.
- Carefully study the context and make informed trade-offs.
- Guess at most likely axes of change and separate along them.
- Be agile: Simulate changes as much as possible; apply SRP when changes actually occur.
- Separation may not be straightforward with typical OO mechanisms.

Class Design Principles

- 2.1 About Class Design Principles (CDPs)
- 2.2 Single Responsibility Principle (SRP)
- 2.3 The Open-Closed Principle (OCP)
- 2.4 Liskov Substitution Principle (LSP)
- 2.5 Interface Segregation Principle (ISP)
- 2.6 Dependency Inversion Principle (DIP)

2.3 The Open-Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modifications.
*(Robert C. Martin, 1996)**

*Martin claims this paraphrases the open-closed principle by Bertrand Meyer, but Meyer's definition is different.

2.3 The Open-Closed Principle (OCP)

- 2.3.1 Extension and Modification
- 2.3.2 Abstraction is the Key
- 2.3.3 OCP by Example
- 2.3.4 Abstractions May Support or Hinder Change
- 2.3.5 Strategic and Agile Opening
- 2.3.6 Takeaway

2.3.1 Extension and Modification

- **Extension**: Extending the *behavior* of an module.
- **Modification***:* Changing the *code* of an module.

- **Open for extension**:
  As requirements of the application change, we can extend the module with new behaviors that satisfy those changes. We change what the module does.
- **Closed for modification**:
  Changes in behavior do not result in changes in the modules source or binary code.

Why Closed for Modifications?

Question: Why not simply change the code if I needed?

- Module was **already delivered to customers**, a change will not be accepted.

  If you need to change something, hopefully you opened your module for extension!

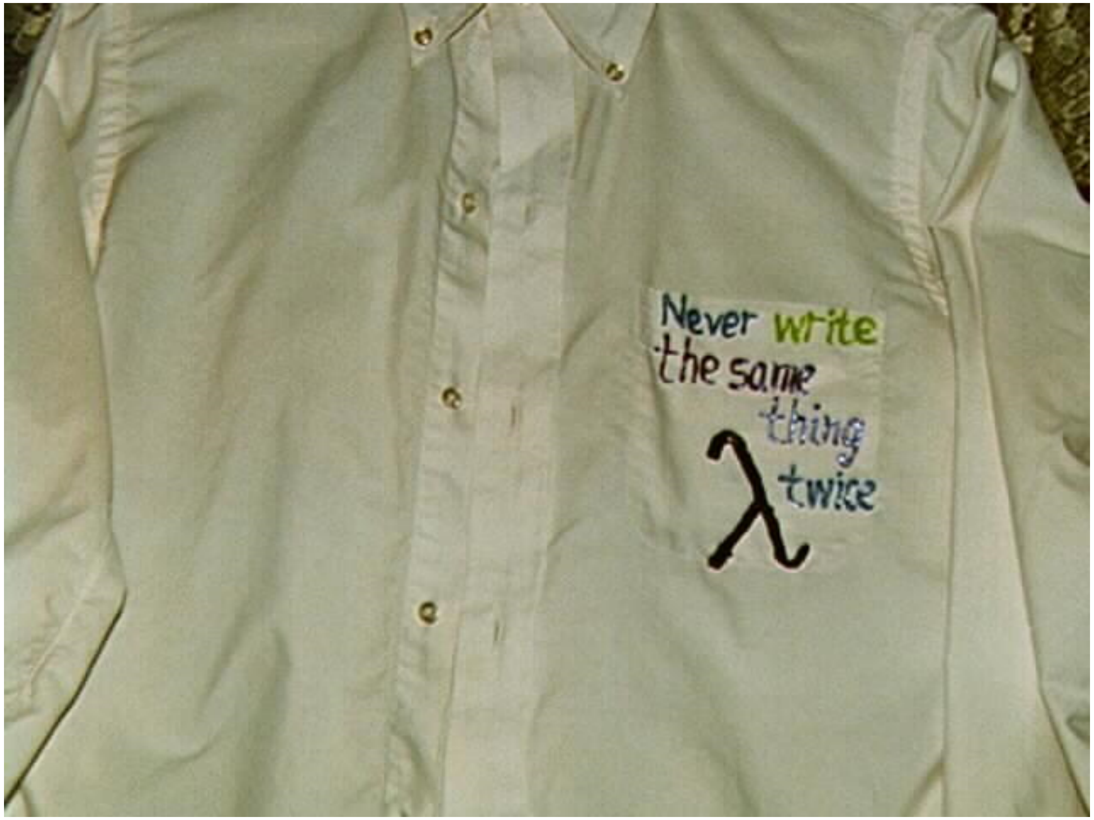
- Module is a **third-party library only available as binary code**.

  If you need to change something, hopefully the third-party opened the module for extension!

- **Most importantly**: not changing existing code for the sake of implementing extensions enables incremental compilation, testing, debugging.

2.3.2 Abstraction is the Key

To enable extending a software entity without modifying it, its implementation must **abstract over variable subparts of behavior**.



Abstraction in Programming Languages

Many programming languages allow to create abstractions that are fixed and yet represent an unbound group of possible behaviors!

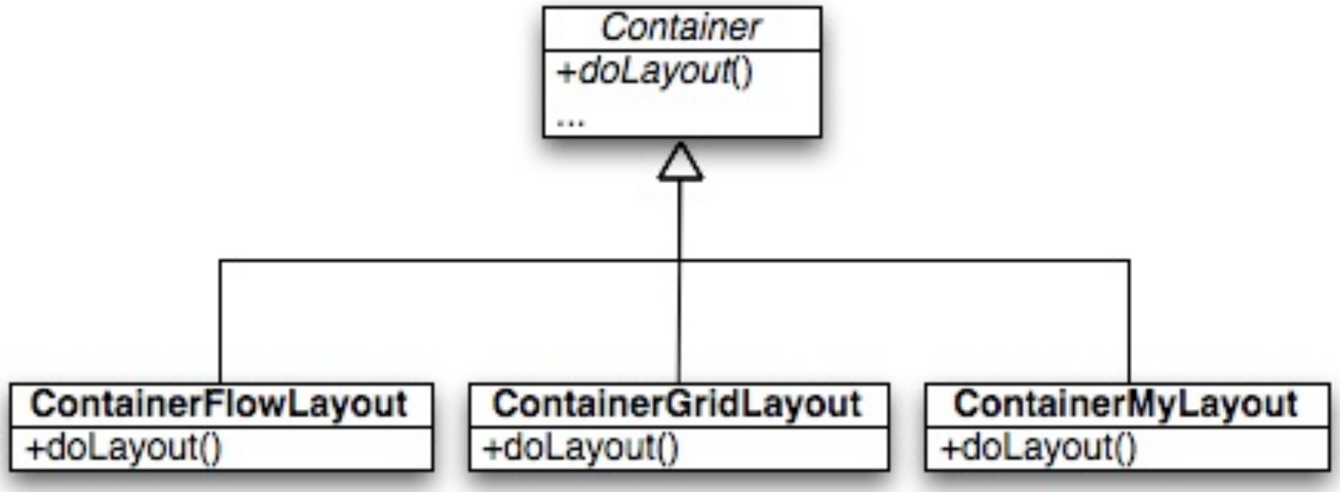
Object-oriented languages:

- abstractions are encoded in abstract base classes, interfaces, generics (type parameters), methods, …
- the unbounded group of possible behaviors is represented by all the possible derivative classes of an abstract class, the implementations of an interface, the instances of a type parameter, all possible arguments to a method, …

Functional languages:

- abstractions are encoded in functions, generic data types/functions, …
- the unbounded group of possible behaviors is represented by all the possible calls of the functions, all possible type instantiations of generic data types/functions, …

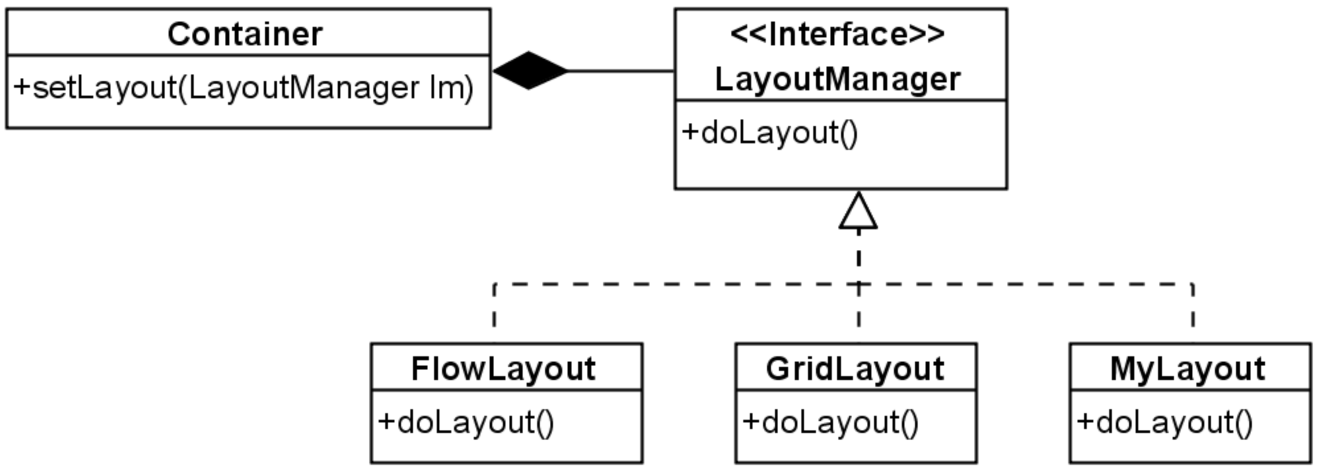
Abstracting over Variations in OO (I)



- `Container` declares the layout functionality as abstract methods, but does not implement it. The rest of `Container` is implemented against the abstraction introduced by the abstract methods.

- Concrete subclasses fill in the details over which `Container's` implementation abstracts.

Abstracting over Variations in OO (II)

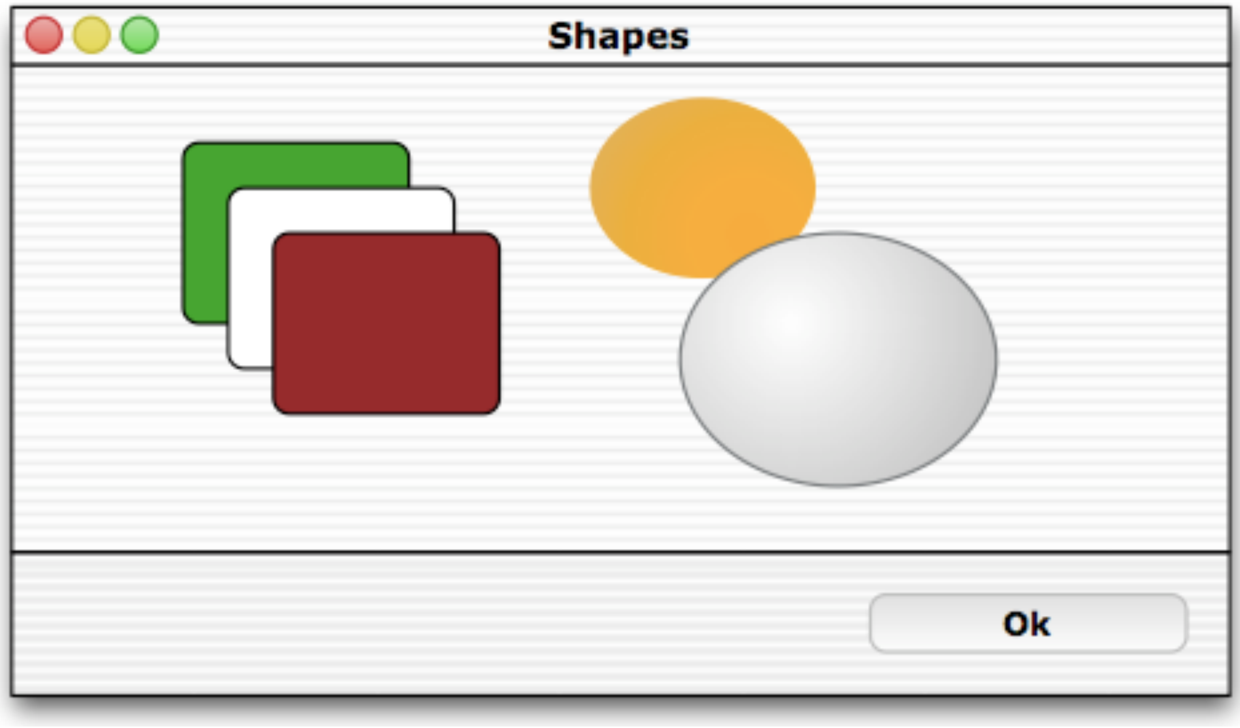


- `Container` delegates the layout functionality to an abstraction. The rest of its functionality is implemented against this abstraction.

- To change the behavior of an instance of `Container` we configure it with the `LayoutManager` of our choice.
- We can add new behavior by implementing our own `LayoutManager`.

2.3.3 OCP by Example

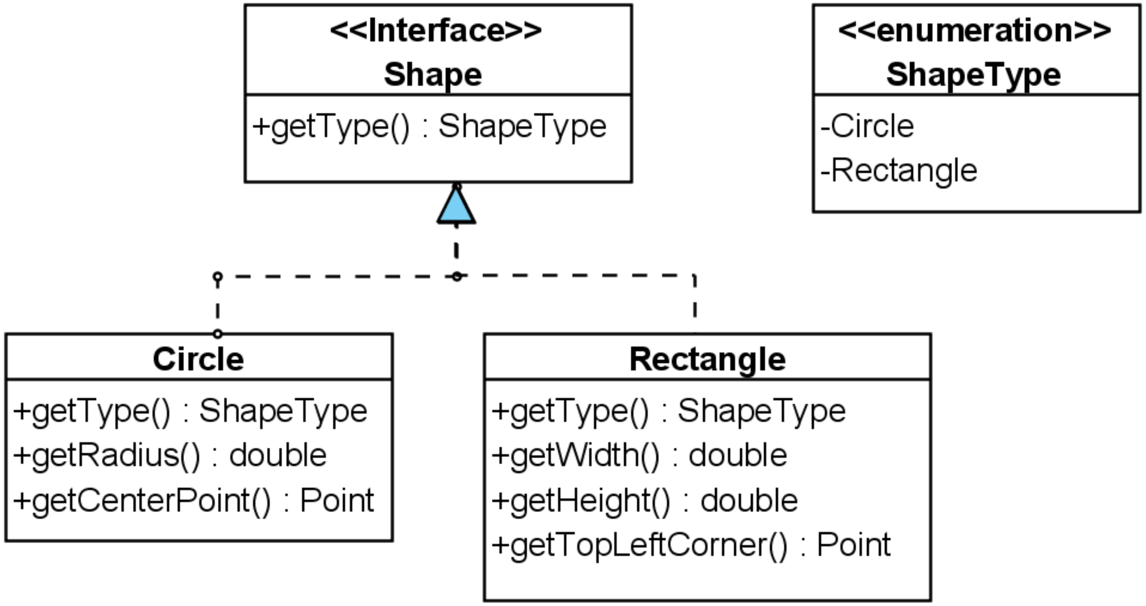
- Consider an application that draws shapes - circles and rectangles – on a standard GUI.



A Possible Design for Drawable Shapes

- Consider the following design of `Shapes`.

- Realizations of `Shape` identify themselves via the enumeration `ShapeType`.
- Realizations of `Shape` declare specialized methods for the shape type they represent; they mostly serve as containers for storing the geometric properties of shapes.



A Possible Design for Drawable Shapes

Drawing is implemented in separate methods (say of `Application` class)

```
public void drawAllShapes(List<Shape> shapes) {
        for(Shape shape : shapes) {
                switch(shape.getType()) {
                case Circle:
                        drawCircle((Circle)shape);
                        break;
                case Rectangle:
                        drawRectangle((Rectangle)shape);
                        break;
                }
        }
}
private void drawCircle(Circle circle) {
...
}
private void drawRectangle(Rectangle rectangle) {
...
}
```

What do you think about the design?

Evaluating the Design

- **Adding new shapes** (e.g., `Triangle`) **is hard**; we need to:
  - Implement a new realization of `Shape`.
  - Add a new member to `ShapeType`.
    This possibly leads to a recompile of all other realizations of `Shape`.
  - `drawAllShapes` (and every method that uses shapes in a similar way) must be changed.
    Hunt for every place that contains conditional logic to distinguish between the types of shapes and add code to it.

- **`drawAllShapes` is hard to reuse**!
  When we reuse it, we have to bring along `Rectangle` and `Circle`.

Rigid, Fragile, Immobile Designs

- **Rigid designs** are hard to change – every change causes many changes to other parts of the system.

  Our example design is rigid: Adding a new shape causes many existing classes to be changed.

- **Fragile designs** tend to break in many places when a single change is made.

  Our example design is fragile: Many switch/case (if/else) statements that are both hard to find and hard to decipher.

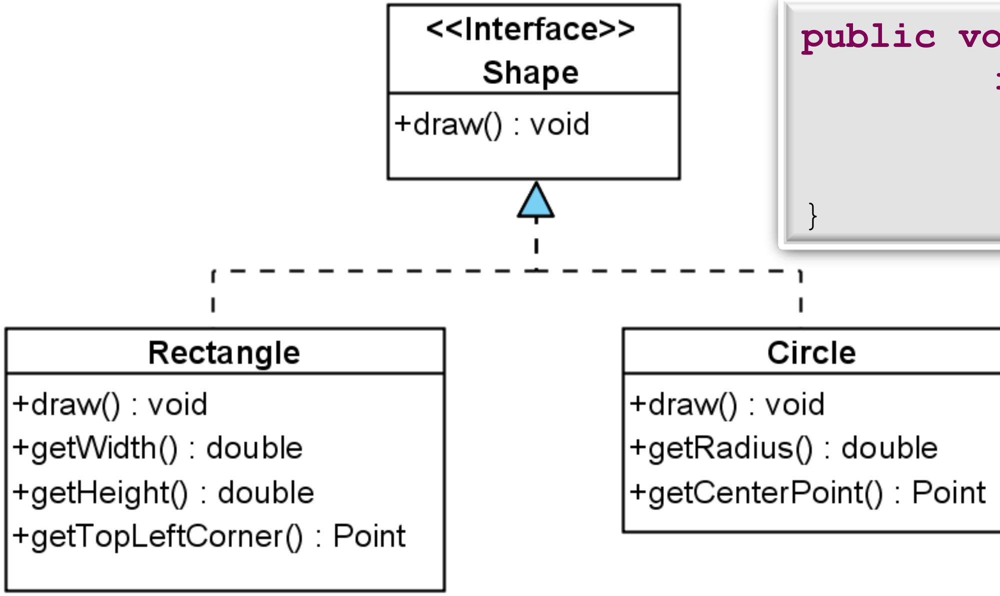
- **Immobile designs** contain parts that could be useful in other systems, but the effort and risk involved with separating those parts from the original system are too big.

  Our example design is immobile: `DrawAllShapes` is hard to reuse.

Evaluating the Design

- **The design violates OCP with respect to extensions with new kinds of shapes.**
- **We need to open our module for this kind of change by building appropriate abstractions.**

An Alternative Design




```java
public void drawAllShapes(List<Shape> shapes) {
        for(Shape shape : shapes) {
                shape.draw();
        }
}
```

This solution complies with OCP!

New abstraction: `Shape.draw()`

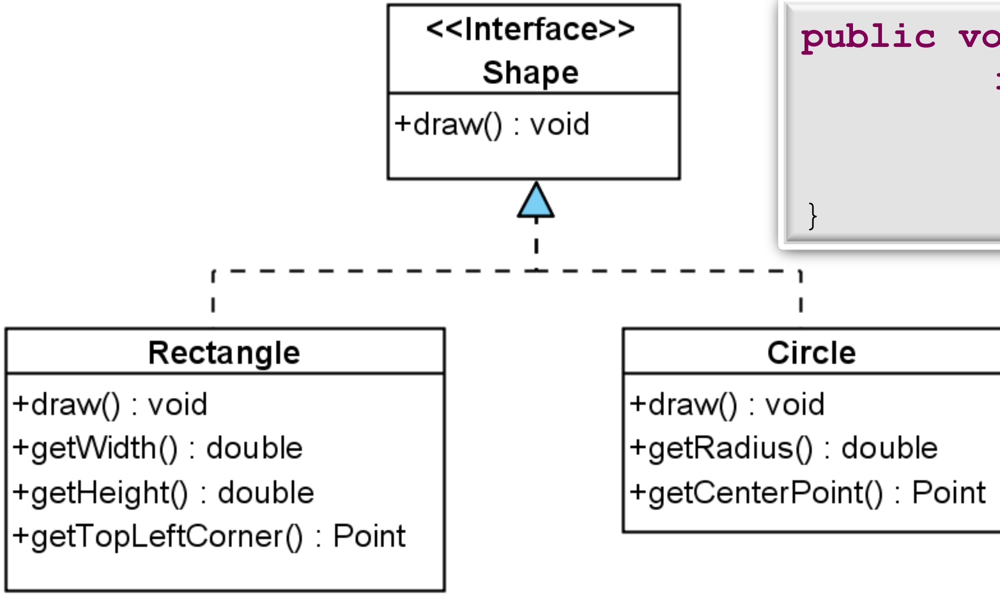
`ShapeType` is not necessary anymore.

Extensibility:

Adding new shapes is easy! Just implement a new realization of `Shape`.

`drawAllShapes` only depends on `Shape`! We can reuse it efficiently.

An Alternative Design



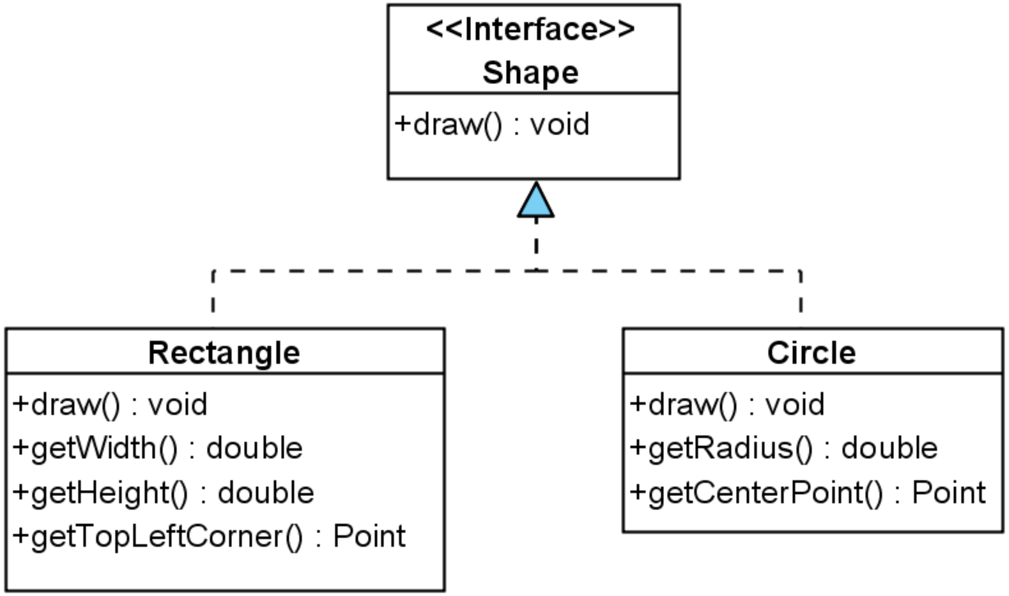

```
public void drawAllShapes(List<Shape> shapes) {
        for(Shape shape : shapes) {
                shape.draw();
        }
}
```

This solution complies with OCP!

Is this statement correct?

No, because the design is **not open with respect to other kinds of changes**.

Problematic Changes



Current abstractions are more of an hindrance to these kinds of change.

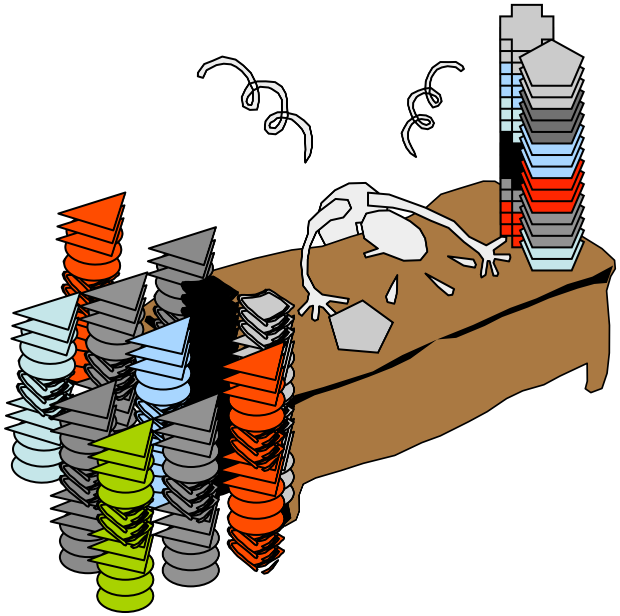
- Consider **extending** the design with further **shape functions**
  - shape transformations,
  - shape dragging,
  - calculating of shape intersection, shape union, etc.

- Consider **adding support for different operating systems**.
  The implementation of the drawing functionality varies for different operating systems.

2.3.4 Abstractions May Support or Hinder Change

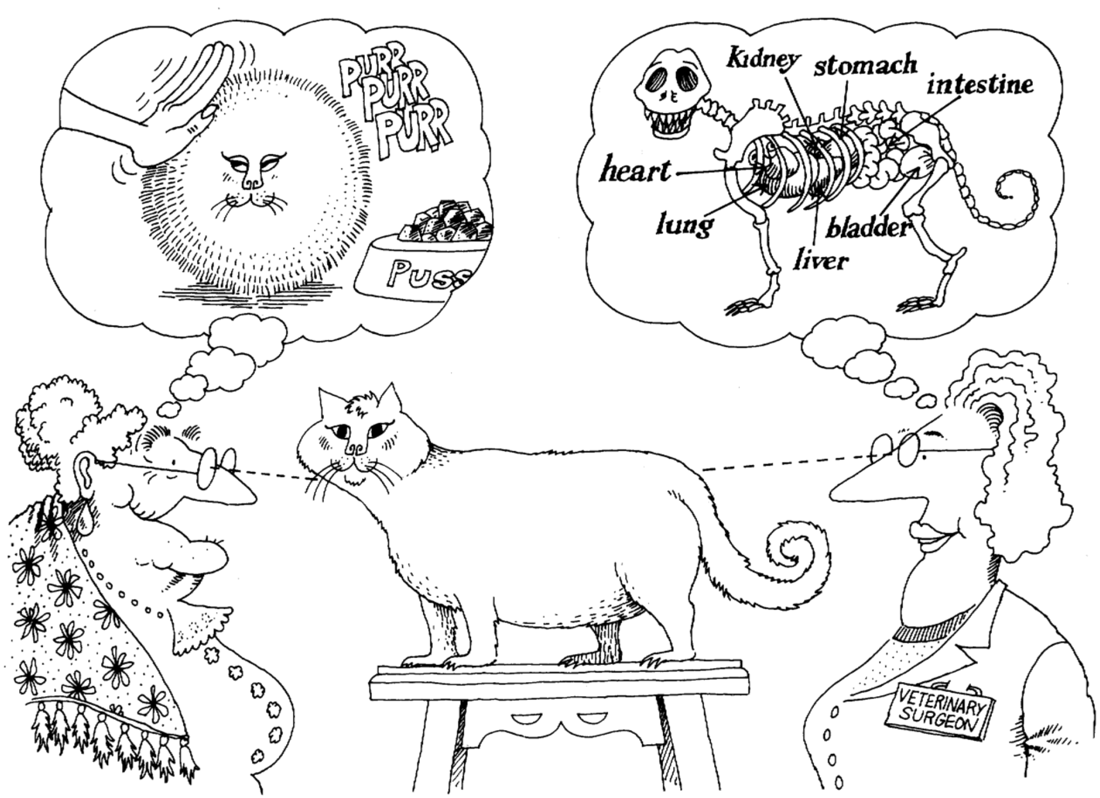
Change is easy if change units correspond to abstraction units.

Change is tedious if change units do not correspond to abstraction units.



Abstractions Reflect a Viewpoint

No matter how “open” a module is, there will always be some kind of change that requires modifcation



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

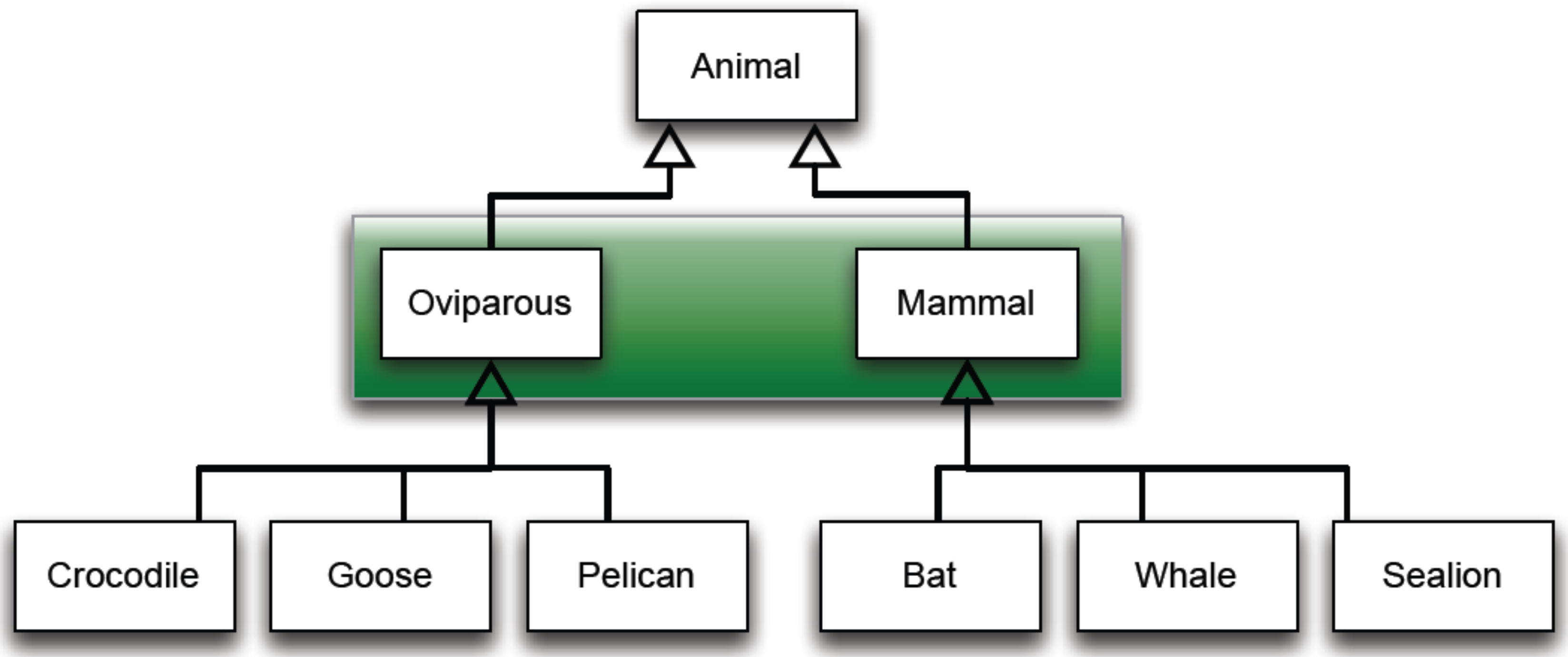
Reason:
There is **no model** that **is natural to all contexts**.

Viewpoints Illustrated: The Case of a Zoo

- Imagine: Development of a "Zoo Software".

- Three stakeholders:
  - Veterinary surgeon
    What matters is how the animals reproduce!
  - Animal trainer
    What matters is the intelligence!
  - Keeper
    What matters is what they eat!

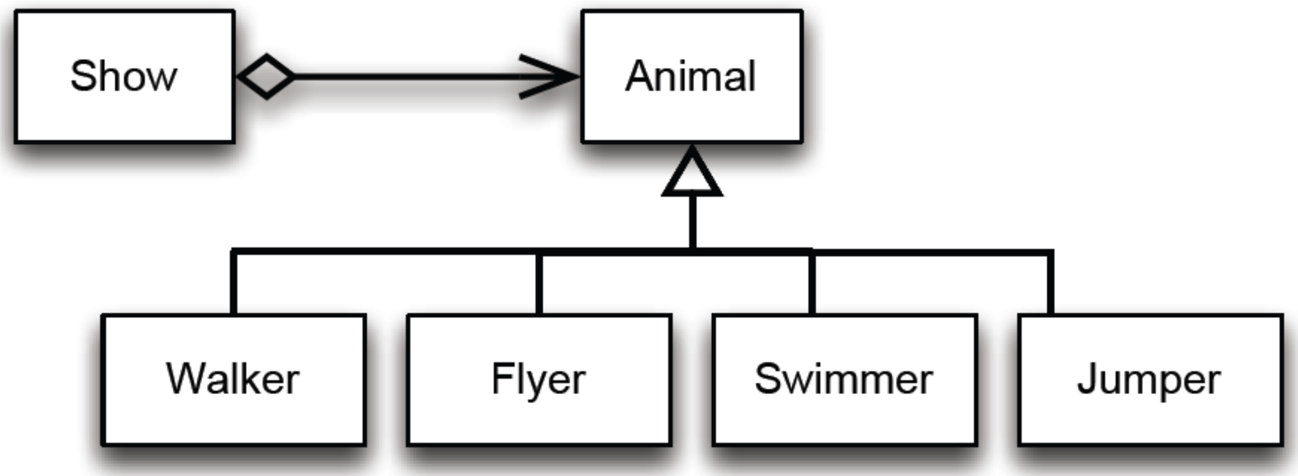
One Possible Class Hierarchy



The veterinary surgeon has "won"!

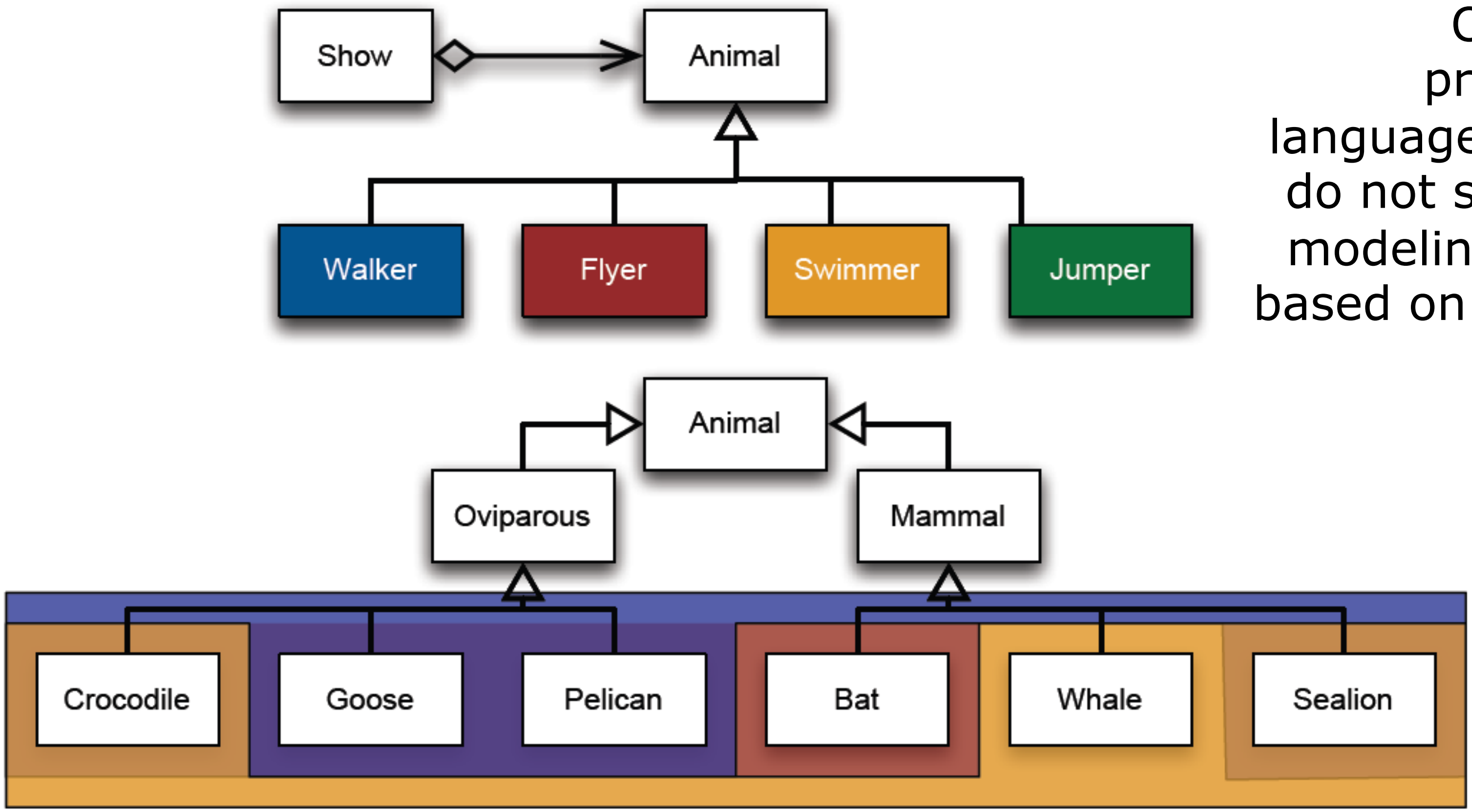
The World from Trainer's Viewpoint

"The show shall start with the pink pelicans and the African geese ***flying*** *across the stage. They are to* ***land*** *at one end of the arena and then* ***walk*** *towards a small door on the side. At the same time, a killer whale should* ***swim*** *in circles and* ***jump*** *just as the pelicans fly by. After the jump, the sea lion should* ***swim*** *past the whale,* ***jump*** *out of the pool, and* ***walk*** *towards the center stage where the announcer is waiting for him."*



Models Reflecting Different Viewpoints Overlap

- Overlapping: Elements of a category in one model correspond to several categories in the other model and the other way around.
- Adopting the veterinary viewpoint hinders changes that concern trainer's viewpoint and the other way around.



Our current programming languages and tools do not support well modeling the world based on co-existing viewpoints.

An Interim Take Away …

No matter how “closed” a module is, there will always be some kind of change against which it is not closed.

2.3.5 Strategic and Agile Opening

Strategic Opening

- Choose the kinds of changes against which to open your module.
  - Guess at the most likely kinds of changes.
  - Construct abstractions to protect from those changes.

- Prescience (*Voraussicht*) derived from experience:
  - Experienced designer hopes to know the user and an industry well enough to judge the probability of different kinds of changes.
  - Invoke OCP for the most probable changes.

Be Agile ...

- Guesses about likely kinds of changes that the application will suffer over time will often be wrong.

- **Conforming to OCP is expensive**.
  - Development time and effort to create the appropriate abstractions
  - Created abstractions might increase the complexity of the design.
    - Needless, Accidental Complexity.
    - Incorrect abstractions supported/maintained even if not used.

- **Be agile**: In doubt, wait for changes to happen. No elaborate upfront design.

2.3.6 Takeaway

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modifications.

- **Abstraction is the key to supporting OCP.**
- **No matter how "open" a module is, there will always be some kind of change which requires modification.**
- **Limit the Application of OCP to changes that are Likely.**
  - After all wait for changes to happen.
  - Stimulate change (agile spirit).

Class Design Principles

- 2.1 About Class Design Principles (CDPs)
- 2.2 Single Responsibility Principle (SRP)
- 2.3 The Open-Closed Principle (OCP)
- 2.4 Liskov Substitution Principle (LSP)
- 2.5 Interface Segregation Principle (ISP)
- 2.6 Dependency Inversion Principle (DIP)

2.4 Liskov Substitution Principle (LSP)

Subtypes must be behaviorally substitutable for their base types.
Barbara Liskov, 1988

2.4 Liskov Substitution Principle (LSP)

- 2.4.1 The Essence of LSP
- 2.4.2 Introduction to LSP by Example
- 2.4.3 The Essence of LSP Revisited
- 2.4.4 More (Realistic) Examples
- 2.4.5 Mechanisms for Supporting LSP
- 2.4.6 Advantages of Design-by-Contract
- 2.4.7 Takeaway

2.4.1 The Essence of LSP

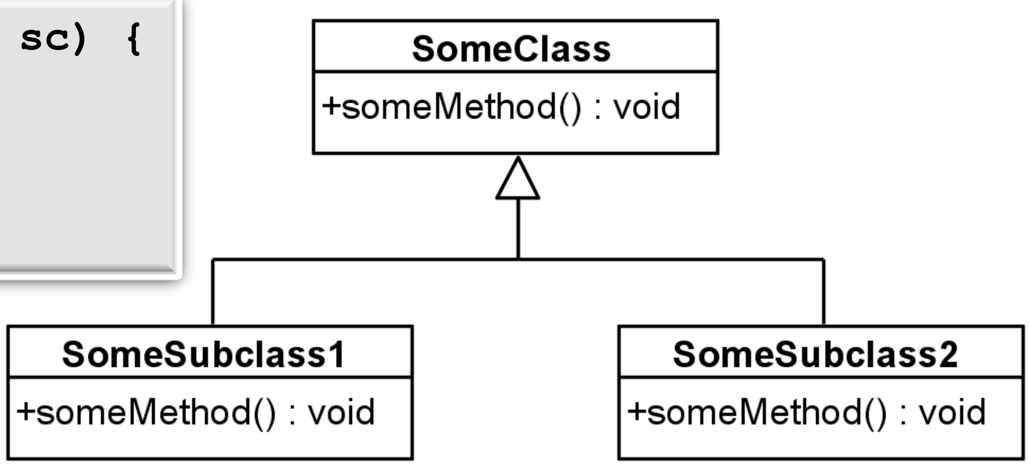
- We identified class inheritance and subtype polymorphism as primary mechanisms for supporting OCP in object-oriented designs.

- **LSP provides us with design rules that govern** this particular **use of inheritance and subtype polymorphism**.

- LSP:
  - gives us a way to characterize good inheritance hierarchies,
  - increases our awareness about traps that will cause us to create hierarchies that do not conform to OCP.

The Essence of LSP

```java
void someClientMethod(SomeClass sc) {
        ...
        sc.someMethod();
        ...
}
```



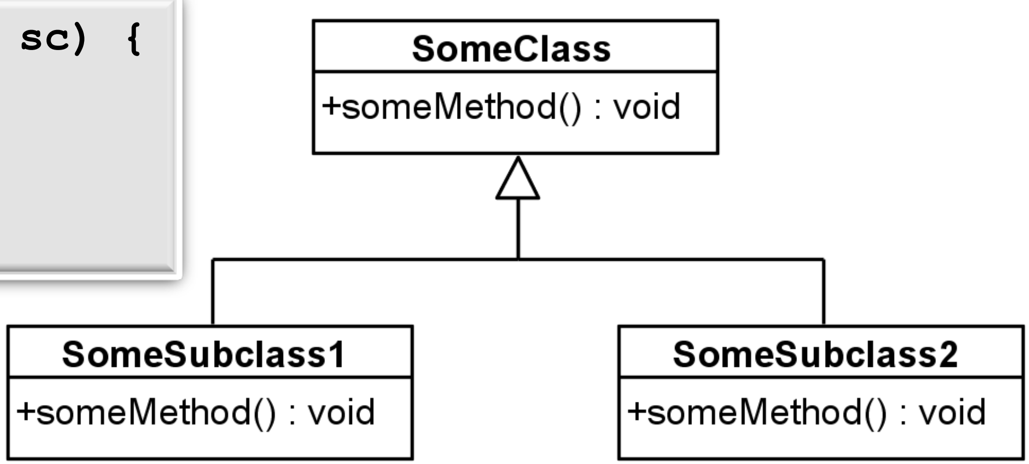
In `someClientMethod`, `sc` can be an instance of `SomeClass` or any of its subclasses.

OO (Java) subtyping rules tell us that `SomeSubclass1`, `SomeSubclass2` are substitutable for `SomeClass` in `someClientMethod`.

So what does LSP add to the common OO subtyping rules?

The Essence of LSP

```
void someClientMethod(SomeClass sc) {
        ...
        sc.someMethod();
        ...
}
```



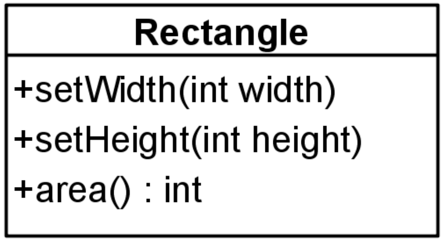
SLP additionally requires ***behavioral substitutability.***

It's not enough that instances of `SomeSubclass1` and `SomeSubclass2` provide all methods that `SomeClass` declares.

These methods should also behave like their heirs.

`someClientMethod` should not be able to distinguish objects of `SomeSubclass1` and `SomeSubclass2` from objects of `SomeClass`.

2.4.2 Introduction to LSP by Example



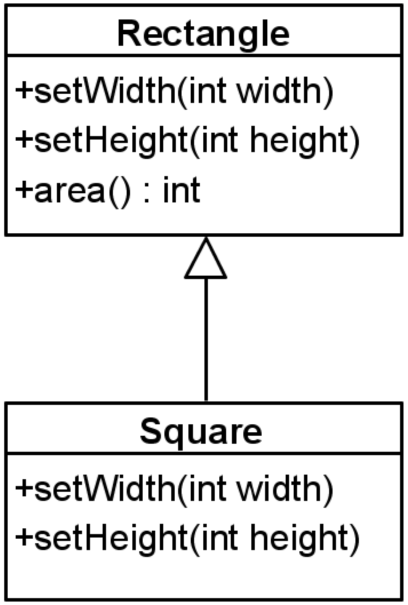

```
class Rectangle {
   public void setWidth(int width) {
         this.width = width;
   }
   public void setHeight(int height) {
         this.height = height;
   }
   public void area() { ... }

   ...
}
```

- Assume we have rectangles.
- We now want to introduce squares.

  A square is mathematically a rectangle; so, we decide to implement `Square` as a subclass of `Rectangle`.

Implementing `Square` as a Subclass of `Rectangle`




```
class Square extends Rectangle
  public void setWidth(int width) {
          super.setWidth(width);
          super.setHeight(width);
  }
  public void setHeight(int height) {
          super.setWidth(height);
          super.setHeight(height);
  }
  ...
}
```

We override `setHeight` and `setWidth` to ensure that `Square` instances always remain mathematically valid. We reuse the rest of `Rectangle`.

This model is self-consistent! So, everything is great!

Do you see any problems?

A Broken Client

```
void someClientMethod(Rectangle rec) {
        rec.setWidth(5);
        rec.setHeight(4);
        assert(rec.area() == 20);
}
```

- Java subtyping rules tell us that we can pass `Square` everywhere a `Rectangle` is expected.

  But, what happens if we pass a square to `someClientMethod`?

- `someClientMethod` works fine with `Rectangle`.

  But, it breaks when `Square` is passed!

- `someClientMethod` makes an assumption that is true for `Rectangle`: setting `width` and `height` do not have mutual effects.
  This assumption does not hold for `Square`.

A design that is self-consistent is not necessarily consistent with clients!

Isn’t a Square a Rectangle?

- Not as far as `someClientMethod` is concerned.

- The behavior of a `Square` object is not consistent with the expectations of `someClientMethod` on the behavior of a `Rectangle`.
- `someClientMethod` can distinguish `Square` objects from `Rectangle` objects.

- The `Rectangle`/`Square` hierarchy violates LSP!
  `Square` is NOT BEHAVIORALLY SUBSTITUTABLE for `Rectangle`.

Isn’t a Square a Rectangle?

- A square complies with mathematical properties of a rectangle.

  A square has four edges and right angles … it is mathematically a rectangle.

- But, a `Square` does not comply with the expected *behavior* of a `Rectangle`!

  Changing the height/width of a `Square`, behaves differently from changing the height/width of a `Rectangle`.

Validity of Designs Relative to Clients

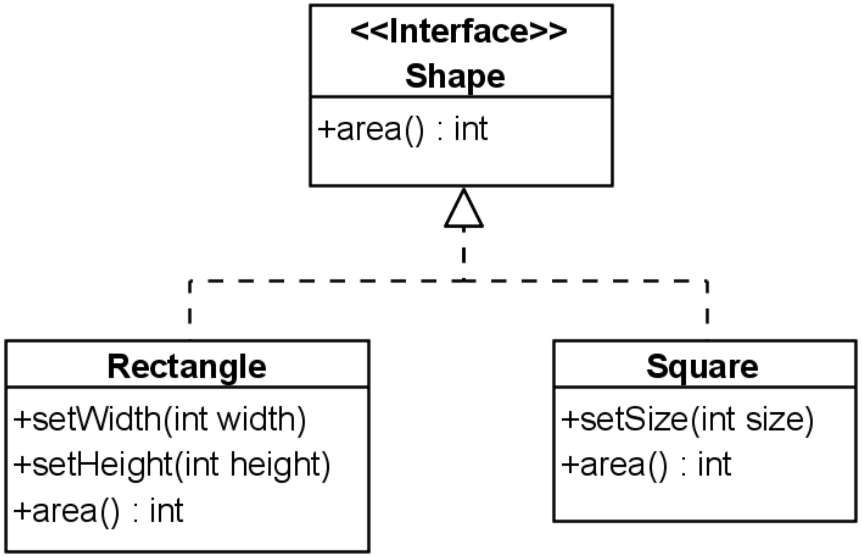
- **A model viewed in isolation cannot be meaningfully validated!**

  The validity of a model depends on the clients that use the model and must be judged from their perspectives.

- Inspecting the `Square`/`Rectangle` hierarchy in isolation did not show any problems.

  It actually seemed to be a self-consistent design.
  We had to inspect the clients to identify problems.

A LSP-Compliant Solution



- `Rectangle` and `Square` are siblings.
- The interface `Shape` declares common methods.

- When clients use `Shape` as a representative for specific shapes they cannot make any assumptions about the behavior of the methods.

- Clients that want to change properties of shapes have to work with the concrete classes and make specific assumptions about them.

2.4.3 The Essence of LSP Revisited

Let p(x) be an observable property of all objects x of type T. Then p(y) should be true for all objects y of type S where S is a subtype of T.

Note: “observable” means “observable by a program using type T”

2.4.4 More (Realistic) Examples

- In the following,
  - we will mention some examples of LSP violations in Java's platform classes
  - will consider a more sophisticated example

- **Goal**: Indicate that violations of LSP are realistic and sophisticated, hence easy to run into them.

LSP Violation “Smells”

- Derivates that override a method of the super-class by an empty method often violate LSP.
- Derivates that document that certain methods inherited from the super-class should not be called by clients.
- Derivates that throw additional (unchecked) exceptions violate LSP.
- …

LSP Violations in Java Platform Classes

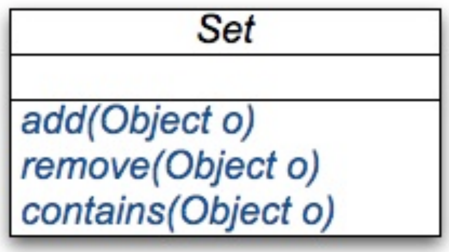
- `Properties` inherits from `Hashtable`

> *Because Properties inherits from Hashtable, the put and putAll methods can be applied to a Properties object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not Strings. The setProperty method should be used instead. If the store or save method is called on a "compromised" Properties object that contains a non-String key or value, the call will fail.*

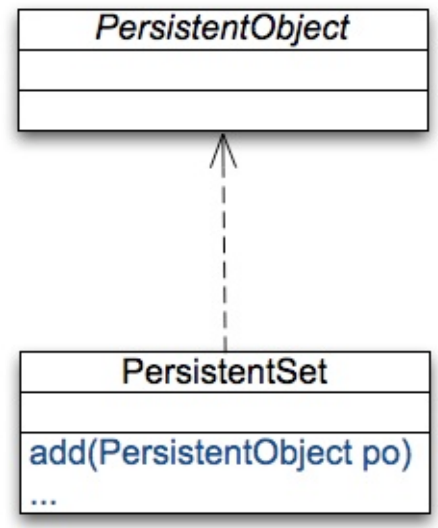
- `Stack` inherits from `Vector`
- ...
- ... I will leave it to you to discover more of them ...

The Case of Implementing a Persistent Set

Consider the following scenario.
We have implemented a library of container classes, including the interface `Set` (e.g. using Java 1.4). We want to extend the library with support for persistent sets.

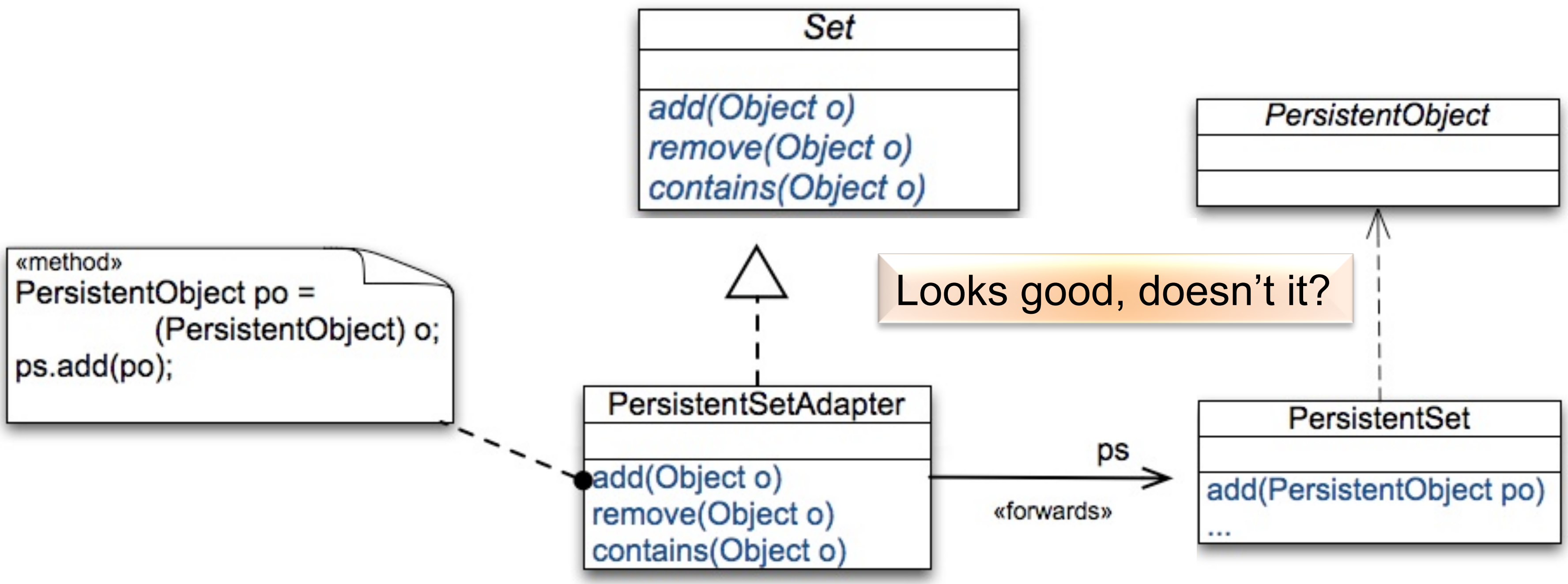


A third-party container class capable of persistence, called `PersistentSet`, is also available.
It accepts objects of type `PersistentObject`.



The Case of Implementing a Persistent Set

We implement our persistent set in `PersistentSetAdapter`.
It implements `Set`, refers to an object of the third party class `PersistentSet`, called `ps`, and implements the operations declared in `Set` by forwarding to `ps`.



The Problem with the Solution Idea

- Only `PersistentObject`s can be added to `PersistentSet`. Yet, nothing in `Set` states this explicitly.

- A client adding elements to a set (`fill` method below) has no idea whether the set is persistent and cannot know whether the elements to fill must be of type `PersistentObject`.
- Passing an arbitrary object will cause the cast in `PersistentSetAdapter` to fail, breaking a method that worked fine before `PersistentAdpaterSet` was introduced.

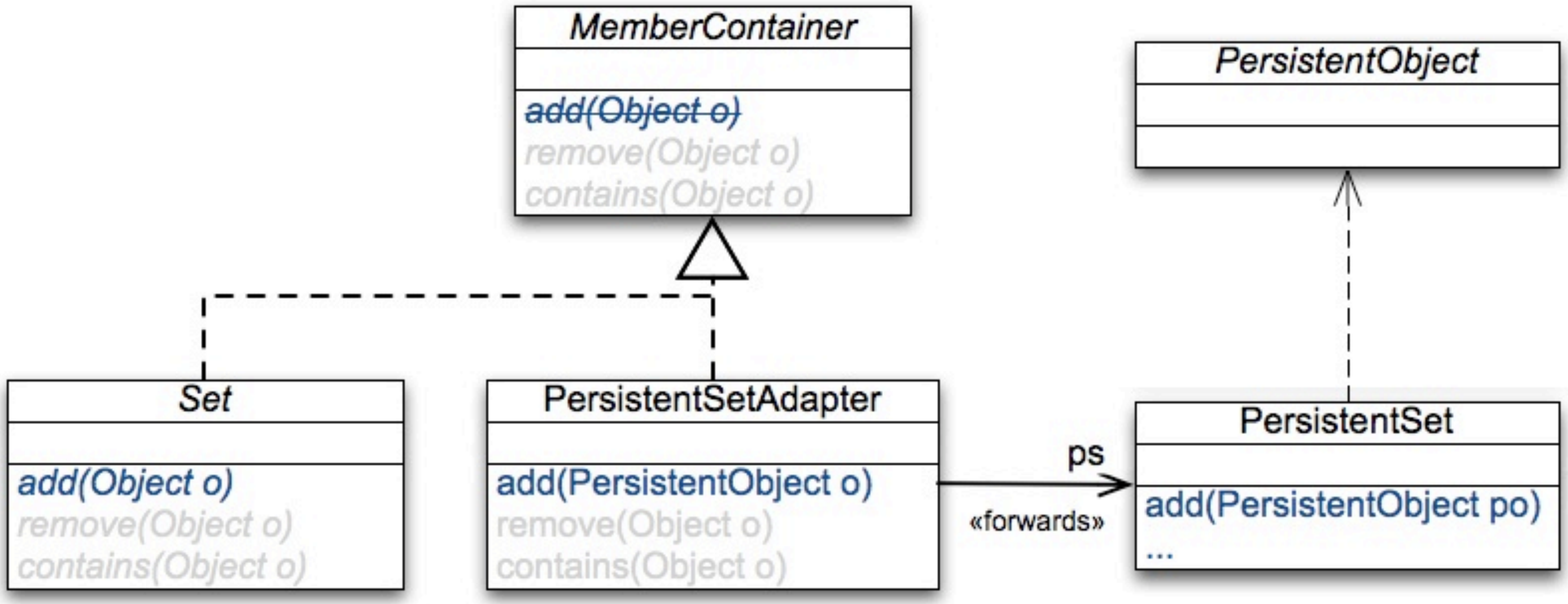
A method

```
public void fill(Set s) {
      fill-the-set-with-some-objects
}
```

Somewhere else...

```
Set s = new PersistentSetAdapter(); // Problem!
fill (s);
```

LSP Compliant Solution



Conclusion:
`PersistentSetAdapter` does not have a behavioral IS-A relationship to `Set`.
We must separate their hierarchies and make them siblings.

2.4.5 Mechanisms for Supporting LSP

The question is: What mechanisms can we use to support LSP?

The Validation Problem

- We said:

  A model viewed in isolation cannot be meaningfully validated with respect to LSP!

  Validity must be judged from the perspective of possible usages of the model.

- Hence, we need to anticipate assumptions that clients make about our models – which is de facto impossible.

  Most of the times we will only be able to view our model in isolation;

  We do not know how it will be used and how it will be extended by means of inheritance.

- Trying to anticipate them all might yield needles complexity.

2.4.5.1 Explicit Contracts for Clients and Subclasses

Solution to the validation problem:

A technique for explicitly stating what may be assumed.

Design-by-Contract.

Two main aspects of design-by-contract.

- **Contracts**. Classes explicitly specify properties:
  - that must be respected by subclasses
  - on which clients can rely.
- **Contract enforcement**. Tools to check (statically or dynamically) the implementation of subclasses against contracts of superclasses.

Specifying Explicit Contracts

The programmer of a class defines a contract, that abstractly specifies the behavior on which clients can rely on.

Pre- and Post-conditions

- Declared for every method of the class.
- Preconditions must be true for the method to execute.
- Post-conditions must be true after the execution of the method.

Invariants

- Properties that are always true for instances of the class.
- May be broken temporarily during a method execution, but otherwise hold.

A Possible Contract for `Rectangle.setWidth`

```
public class Rectangle implements Shape {

        private int width;
        private int height;

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

}
```

- Precondition for `setWidth`: `w` > 0
- Post-condition for `setWidth`: `getWidth() = w`
  `getHeight()` was not changed

Enforcement or Behavioral Subtyping

- **Subclasses must conform to the contract of their base class!**
- This is called **behavioral subtyping**.
- It ensures, that clients won’t break when instances of subclasses are used in the guise of instances of their heirs!

What do you think should the subtyping rules look like?

Behavioral Subtyping

Rule for preconditions

- Preconditions may be replaced by equal or weaker ones.
- Preconditions of a class imply preconditions of subclasses.

Rule for post-conditions

- Post-conditions may be replaced equal or stronger ones.
- Post-conditions of a class are implied by those of its subclasses.

Behavioral Subtyping

Rationale for the preconditions rule

- A derived class must not impose more obligations on clients.
- Conditions that clients obey to before executing a method on an object of the base class should suffice to call the same method on instances of subclasses.

Rationale for the post-conditions rule

- Properties assumed by clients after executing a method on an object of the base class still hold when the same method is executed on instances of subclasses.
- The guarantees that a method gives to clients can only become stronger.

Contracts and Types

- Contracts exceed what can be expressed using only types.
- Type systems have some desirable properties that contracts do not have (in general)
  - Static, compositional checking
    - That methods adhere to their declared types
    - That types of overridden methods are refined in a LSP-consistent way
      - Such as covariance for return types, contravariance for argument types
- Some things expressed in contracts can also be expressed in more powerful type systems
  - Contracts can also be seen as part of types
- Generics and variance annotations (Java, Scala) form a powerful specialized contract language

Behavioral Subtyping is Undecidable in General

- By Rice's theorem any interesting property about the behavior of programs is undecidable.
- This applies to contracts and LSP, too.
- This is not news, since already plain type checking is undecidable.
- Standard solution: Err on the safe side.
- LSP is useful however in reasoning about the design of class hierarchies.

Languages and Tools for Design-by-Contract

- **Comments as contracts.**
  Easy and always possible, but not machine checkable.

- **Unit-tests as contracts.**
  Machine checkable, but not declarative, possibly cumbersome, always incomplete (tests check only single program runs).

- **Formalisms and tools for specifying contracts in a declarative way and enforcing them**.
  - The Eifel language has built-in support for design-by-contracts (the term was coined by B. Meyer).
  - Java Modeling Language (JML) uses annotations to specify pre-/post-conditions for Java http://www.eecs.ucf.edu/~leavens/JML/
  - More recent languages, e.g., IBMs X10, integrate DbC into the language’s type system by means of dependent types (types that depend on values).

2.4.5.2 Contracts in Documentation

One should document any restrictions on how a method may be overridden in subclasses.

The Contract of `Object.equals`

- The method `equals` in `Object` implements identity-based equality to mean: "*Each instance of a class is equal only to itself*"

- Java classes may override it to implement "logical equality".

- The documentation of `Object.equals` consists almost entirely of restrictions on how it may be overridden.

equals

`public boolean equals(Object obj)`

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation:

- It is *reflexive*: for any reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals`
- It is *consistent*: for any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently ret `equals` comparisons on the object is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any refere only if `x` and `y` refer to the same object (`x==y` has the value `true`).

The Contract of `Object.equals`

The `equals` method implements an equivalence relation:

- It is **reflexive**:
  For any reference value `x`, `x.equals(x)` must return true.
- It is **symmetric**:
  For any reference values `x` and `y`, `x.equals(y)` must return true if and only if `y.equals(x)` returns true.
- It is **transitive**:
  For any reference values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` must return true.
- It is **consistent**:
  For any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in `equals` comparisons on the object is modified.
- For any non-null reference value `x`, `x.equals(null)` must return false.

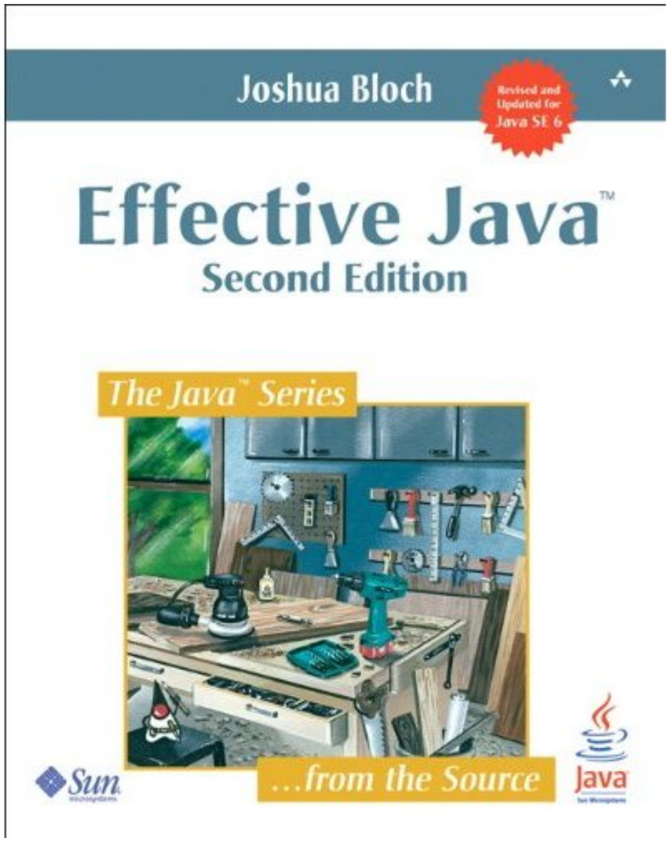
The Contract of `Object.equals`

- Violations of these restrictions may have dire consequences and it can be very difficult to pin down the source of the failure.
- **No class is an island**.

  Instances of a class are often passed to another.
- **Many classes**, including all collection classes, **depend on the objects passed to them obeying the `equals` contract**.

The Contract of `Object.equals`

Read more in
http://java.sun.com/developer/Books/effectivejava/Chapter3.pdf



"An excellent book, crammed with good advice on using the Java programming language and object-oriented programming in general."–

Gilad Bracha, coauthor of The Java™ Language Specification, Third Edition

In the following: we will discuss two restrictions on overriding equals from chapter 3 of the book.

Example Implementation of `equals`

```
/**
* Case-insensitive string. Case of the original string is
* preserved by toString, but ignored in comparisons.
*/
public final class CaseInsensitiveString {
   private String s;
   public CaseInsensitiveString(String s) {
        if (s == null) throw new NullPointerException();
        this.s = s;
   }

   public boolean equals(Object o) {
      if (o instanceof CaseInsensitiveString)
         return s.equalsIgnoreCase(((CaseInsensitiveString)o).s);
      if (o instanceof String)
         return s.equalsIgnoreCase((String)o);
      return false;
   }
... // Remainder omitted
}
```

What do you think?

Example Implementation of `equals`

```
/**
* Case-insensitive string. Case of the original string is
* preserved by toString, but ignored in comparisons.
*/
public final class CaseInsensitiveString {
   private String s;
   public CaseInsensitiveString(String s) {
        if (s == null) throw new NullPointerException();
        this.s = s;
   }

   public boolean equals(Object o) {
      if (o instanceof CaseInsensitiveString)
         return s.equalsIgnoreCase(((CaseInsensitiveString)o).s);
      if (o instanceof String)
         return s.equalsIgnoreCase((String)o);
      return false;
   }
... // Remainder omitted
}
```

BROKEN:
Violates symmetry!

One-way
interoperability!

Example Implementation of `equals`

- The problem: While `CaseInsensitiveString.equals` knows about ordinary strings, `String.equals` is oblivious to case-insensitive strings.

- No one knows what `list.contains(s)` would return in the code below. The result may vary from one Java implementation (of ArrayList) to another.

```
...
CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
String s = "polish";
List list = new ArrayList();
list.add(cis);
...
return list.contains(s);
```

Once you have violated equals contract, you simply don’t know how other objects will behave when confronted with your object.

The Implementation of `java.net.URL.equals`

- `java.net.URL`'s `equals` method violates the consistent part of equals contract.
- The implementation of that method relies on the IP addresses of the hosts in URLs being compared.
- Translating a host name to an IP address can require network access, and it isn't guaranteed to yield the same results over time.
- This can cause the URL equals method to violate the equals contract, and it has caused problems in practice.
  (Unfortunately, this behavior cannot be changed due to compatibility requirements.)

The Imperative of Documenting Contracts

- It is particularly important to carefully and precisely document methods that may be overridden because one can not deduce the intended specification from the code.

- Compare the documentation of the requirements of `equals` with the implementation of `equals` in java.lang.Object given below!

```
public boolean equals( Object ob )
    {
        return this == ob;
    }
```

The Imperative of Documenting Contracts

- RFC 2119 defines keywords - may, should, must, etc. – which can be used to express so-called „subclassing directives".

```
/**
 * Subclasses should override.
 * Subclasses may call super
 * New implementation should call addPage
 */

public void addPages() {...}
```

On the Quality of the Documentation

When documenting methods that may be overridden, one must be careful to document the method in a way that will make sense for all potential overrides of the function.

When Contracts are Too Specific

Consider the class `Point2D` … and its subclass `Point3D`

```
class Point2D {
...
  /** Indicate whether two points are equal.
    * Returns true iff the values of the respective
    * x and y coordinates of this and ob are equal.
  */

  @Override public boolean equals(Object ob) {
     if (ob instanceof Point2D) {
        Point2D other = (Point2D) ob;
        return this.x == that.x && this.y == other.y;
     }
     else
        return false;
   }
...
}
```

When Contracts are Too Specific

```
class Point3D extends Point2D {
...

  @Override public boolean equals(Object ob) {
        if (ob instanceof Point3D) {
             Point3D other = (Point3D) ob ;
             return
                 this.z == other.z && super.equals(ob3D);
         }
        else
           return super.equals(ob);
  }
...
}
```

What do you think about it?

`Point3D` violates LSP!

When Contracts are Too Specific

aOrB may not behave according to our expectations when passed two **Point3D** objects as parameters. Do you see why?

```
void aOrB(Point2D p, Point2D q)
{
    p.setX(x_0);
    p.setY(y_0);
    q.setX(x_1);
    q.setY(y_1);

    if (p.equals(q))
       { ... do a ...}
    else
       { ... do b ...}
}
```

Consider the case :
x_0 == x_1,
y_0 == y_1,

Based on the specification of **equals** in **Point2D**, our expectations are that the true-sub-expression of **if** will be executed in this case. However, when **p** and **q** are **Point3D** objects, and **p.z != q.z** , the else-sub-expression will be executed instead.

When Contracts are Too Specific

`Point3D.equals` is not really guilty for that.
The problem is rather that the expectations of the client are too "high" due to the documentation of **`Point2D.equals`** being too specific.

Should reword the documentation of **`Point2D.equals`** to be more flexible, e.g., "returns true iff the coordinates of the receiver and argument points are equal"

One could also argue that **`Point3D`** should not be a subtype of **`Point2D`** anyway.

2.4.5.3 Java Modeling Language (JML)

- A behavioral interface specification language that can be used to specify the behavior of Java modules.
- Specifications written as Java annotation comments to the Java program, which hence can be compiled with any Java compiler.

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

        /*@ requires w > 0;
          @ ensures getHeight() = \old(getHeight())
                    && getWidth() = w; @*/
        public void setWidth(double w)  {
                this.width = w;
        }
}
```

Java Modeling Language (JML)

Several tools exist that process JML specifications.

- An assertion-checking compiler (jmlc) - runtime verification of assertions.
- A unit testing tool (jmlunit).
- An enhanced version of javadoc (jmldoc) that understands JML specifications.
- Extended Static Checker (ESC/Java) is a static verification tool that uses JML as its front-end.

2.4.6 Advantages of Design-by-Contract

- **Explicit statement of obligations and rights between clients and servers.**
  Clients have the obligation to satisfy pre-conditions and the right to expect post-conditions.
  … and the other way around.

- **Machine checkable contracts help to avoid constantly checking arguments.**
  Especially checking if an argument is `null`.

- **Contracts as documentation and abstraction.**
  - Document by saying what a method require and what it ensures.
  - Often machine checkable!
  - Separates the interface from the implementation.
    Contract: What is done (not constructive)
    Implementation: How is it done (constructive)

2.4.7 Takeaway

> *Subtypes must be behaviorally substitutable for their base types.*
> *Barbara Liskov, 1988*

- **Behavioral subtyping extends "standard" OO subtyping.**

  Additionally ensures that assumptions of clients about the behavior of a base class are not broken by subclasses.

- **Design-by-Contract is a technique for supporting LSP.**

  Makes the contract of a class to be assumed by the clients and respected by subclasses explicit (and checkable).

- **DbC** does not guarantee LSP, though
  - Contracts specify only a subset of the observable properties

Class Design Principles

- 2.1 About Class Design Principles (CDPs)
- 2.2 Single Responsibility Principle (SRP)
- 2.3 The Open-Closed Principle (OCP)
- 2.4 Liskov Substitution Principle (LSP)
- 2.5 Interface Segregation Principle (ISP)
- 2.6 Dependency Inversion Principle (DIP)

2.5 Interface Segregation Principle (ISP)

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

2.5 Interface Segregation Principle (ISP)

- 2.5.1 The Rationale Behind ISP
- 2.5.2 Introduction to ISP by Example
- 2.5.3 Proliferation of Interfaces
- 2.5.4 Takeaway

2.5.1 The Rationale Behind ISP

- When clients are forced to depend on methods they do not use, they become subject to changes to these methods, which other clients force upon the class.

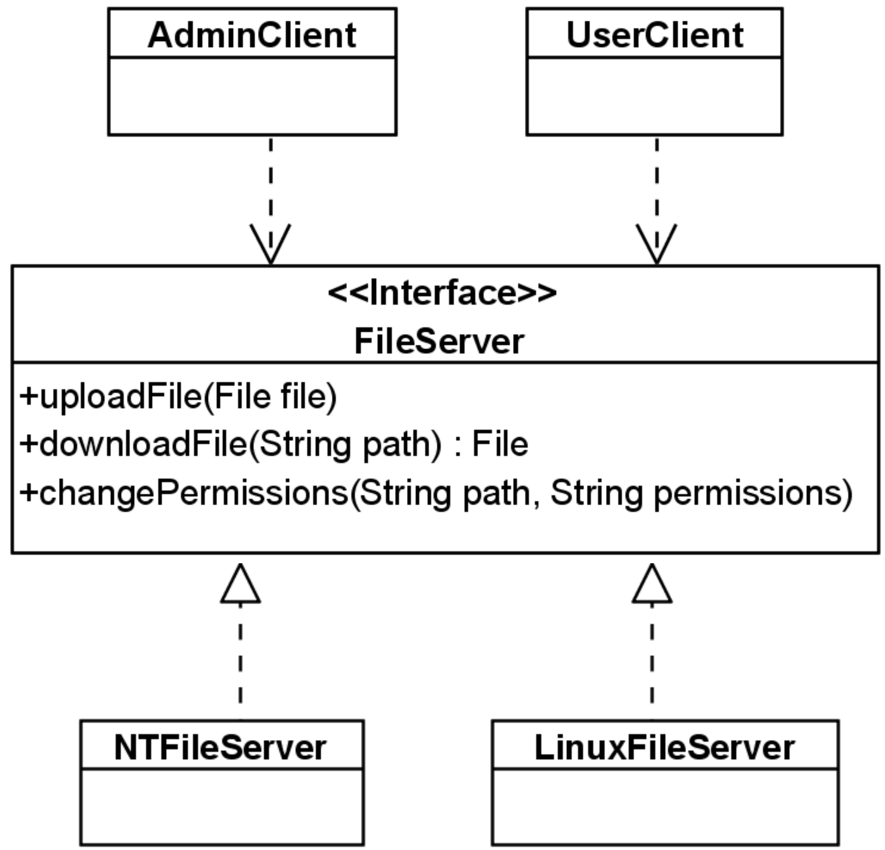
- This causes coupling between all clients.

2.5.2 Introduction to ISP by Example

- Consider the design of a file server system.
- The interface `FileServer` declares methods provided by any file server.
- Various classes implement this interface for different operating systems.
- Two clients are implemented for the file server:

  `AdminClient`, which uses all methods.

  `UserClient`, which uses only the upload/download methods.



Do you see any problems?

Problems of the Proposed Design

- Having the option of calling `changePermissions()` does not make sense when implementing `UserClient`.
  The programmer must avoid calling it by convention instead of by design.

- Modifications to `changePermissions()` triggered by needs of `AdminClient` may affect `UserClient`, even though it does not use `changePermissions()`.
  - Mainly an issue with binary compatibility. A non-issue with dynamic linking.

- There may be servers that do not use a permission system.
  If we wanted to reuse `UserClient` for these servers, they would be forced to implement `changePermissions`, even though it wont be used.

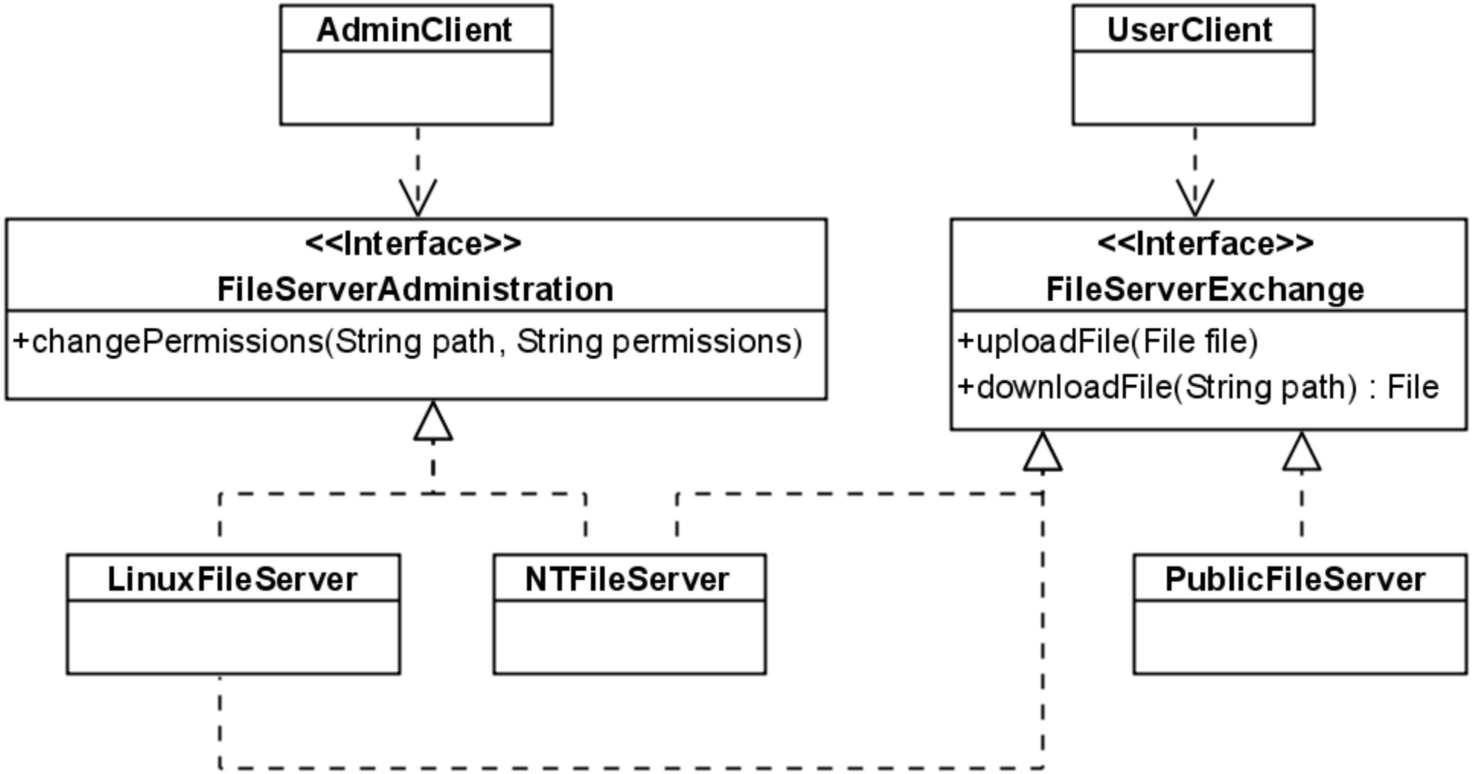
A Polluted Interface

<<Interface>> **FileServer**
+uploadFile(File file) +downloadFile(String path) : File +changePermissions(String path, String permissions)

- `FileServer` is a polluted interface.
  - It declares methods that do not belong together.
  - It forces classes to depend on unused methods and therefore depend on changes that should not affect them.

- ISP states that such interfaces should be split.

A ISP-Compliant Solution



2.5.3 Proliferation of Interfaces

- ISP should not be overdone!
  Otherwise you will end up with 2^n-1 interfaces for a class with n methods.
  (an argument for structural subtyping?)

- A class implementing many interfaces may be a sign of a SRP-violation!

- Try to group possible clients of a class and have an interface for each group.

2.5.4 Takeaway

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

- Interfaces that declare unrelated methods force clients to depend on changes that should not affect them.

- Polluted interfaces should be split.

- But, be careful with interface proliferation.

Class Design Principles

- 2.1 About Class Design Principles (CDPs)
- 2.2 Single Responsibility Principle (SRP)
- 2.3 The Open-Closed Principle (OCP)
- 2.4 Liskov Substitution Principle (LSP)
- 2.5 Interface Segregation Principle (ISP)
- 2.6 Dependency Inversion Principle (DIP)

2.6 Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

2.6 Dependency Inversion Principle (DIP)

- 2.6.1 The Rationale of DIP
- 2.6.2 Introduction to DIP by Example
- 2.6.3 Layers and Dependencies
- 2.6.4 Naive Heuristic for Ensuring DIP
- 2.6.5 Takeaway

2.6.1 The Rationale of DIP

High-level, low-level Modules.

Good software designs are structured into modules.

- High-level modules contain the important policy decisions and business models of an application – The identity of the application.
- Low-level modules contain detailed implementations of individual mechanisms needed to realize the policy.

High-level policy:
The abstraction that underlies the application;
the truth that does not vary when details are changed;
the system inside the system;
the metaphor.

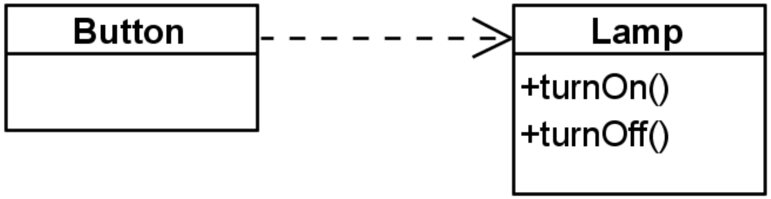
The Rationale of DIP

- High-level policies and business processes is what we want to reuse.

- If high-level modules depend on the low-level modules changes to the lower level details will force high-level modules to change.
- It becomes harder to use them in other contexts.

- It is the high-level modules that should influence the low-level details

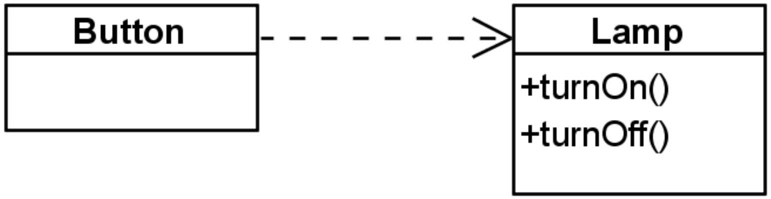
2.6.2 Introduction to DIP by Example



- Consider a design excerpt from the smart home scenario.
- `Button`
  - Is capable of "sensing" whether it has been activated/deactivated by the user.
  - Once a change is detected, it turns the `Lamp` on respectively off.

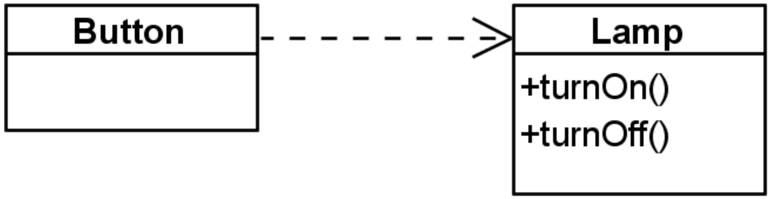
Do you see any problem with this design?

Problems with `Button/Lamp`



- We cannot reuse `Button` since it depends directly on `Lamp`. But there are plenty of other uses for `Button`.
- `Button` should not depend on the details represented by `Lamp`.
- These are symptoms of the real problem (Violation of DIP):
- The high-level policy underlying this (mini) design is not independent of the low-level details.

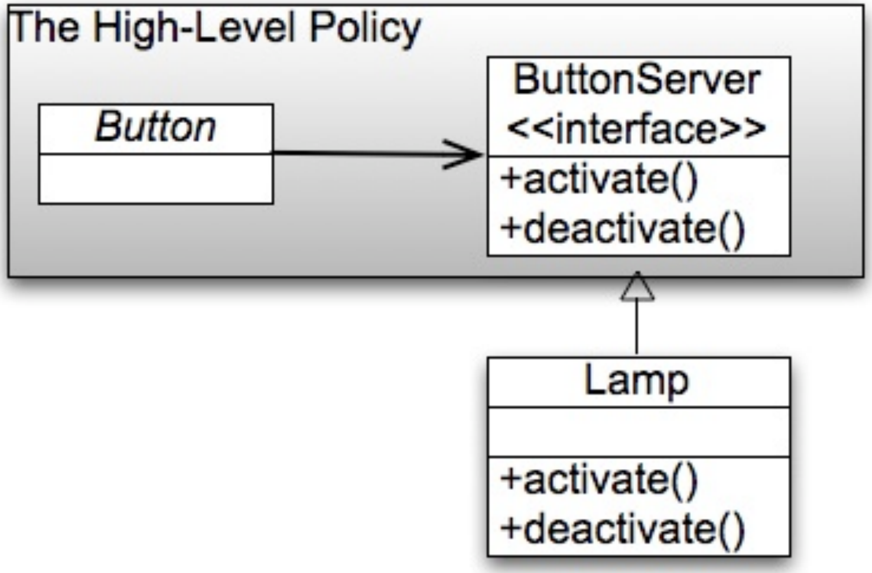
The High-Level Policy



The underlying abstraction is the detection of on/off gestures and their delegation to a server object that can handle them.

- If the interface of `Lamp` is changed, `Button` has to be adjusted, even though the policy that `Button` represents is not changed!
- To make the high-level policy independent of details we should be able to define it independent of the details of `Lamp` or any other specific device.

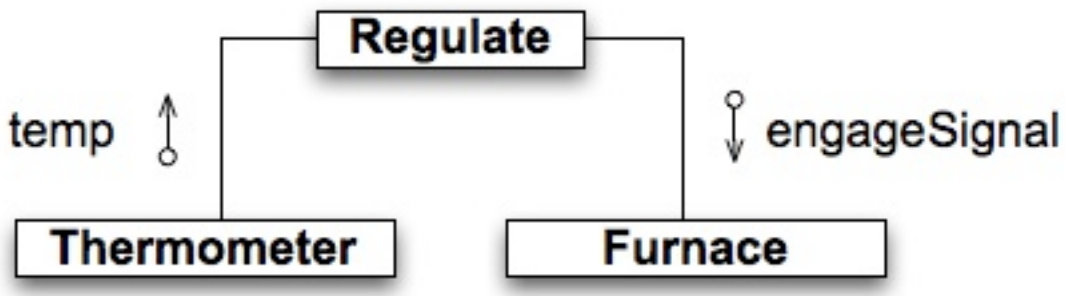
A DIP-Compliant Solution



- Now `Button` only depends on abstractions!
  It can be reused with various classes that implement `ButtonServer`.
- Changes in `Lamp` will not affect `Button`!

- The dependencies have been inverted:
  `Lamp` now has to conform to the interface defined by `Button`.
- Actually: both depend on an abstraction!

A Quick Quiz

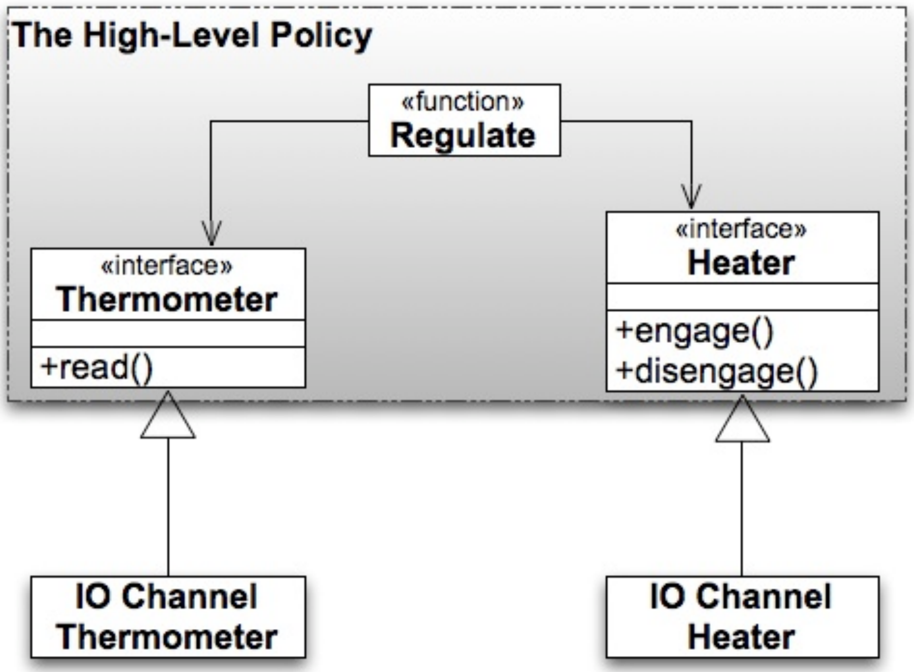


- Three subprograms: `Regulate` calls the other two.
  - `Regulate` pulls data about the current temperature from the `Thermometer` component and
  - `Regulate` signals the `Furnace` component to increase or decrease heat.

Does it conform to DIP?

If not, how would you make it DIP-compliant?

Answer to the Quiz

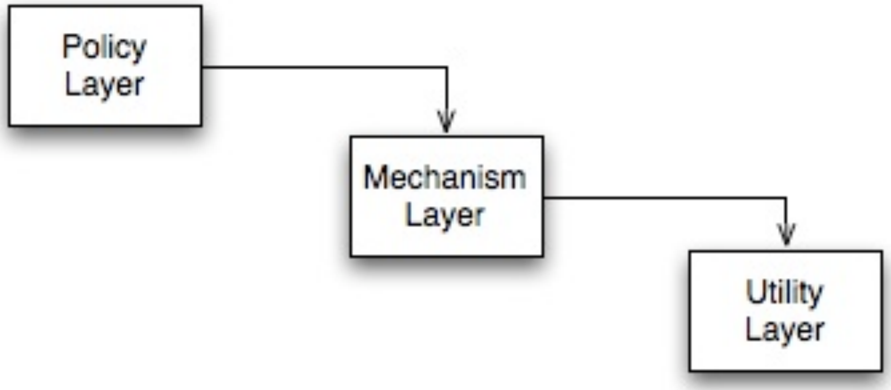


2.6.3 Layers and Dependencies

Grady Booch

„...all well-structured object-oriented architectures have clearly defined layers, with each layer providing some coherent set of services through a well-defined and controlled interface..."

A possible interpretation of Booch's statement...

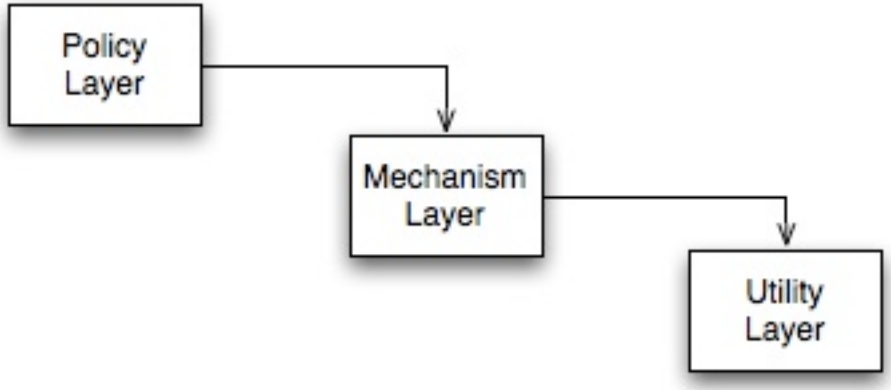


The higher the module is positioned in a layered architecture, the more general the function it implements. The lower the module, the more detailed the function it implements.

What do you think about this interpretation?

Layers and Dependencies

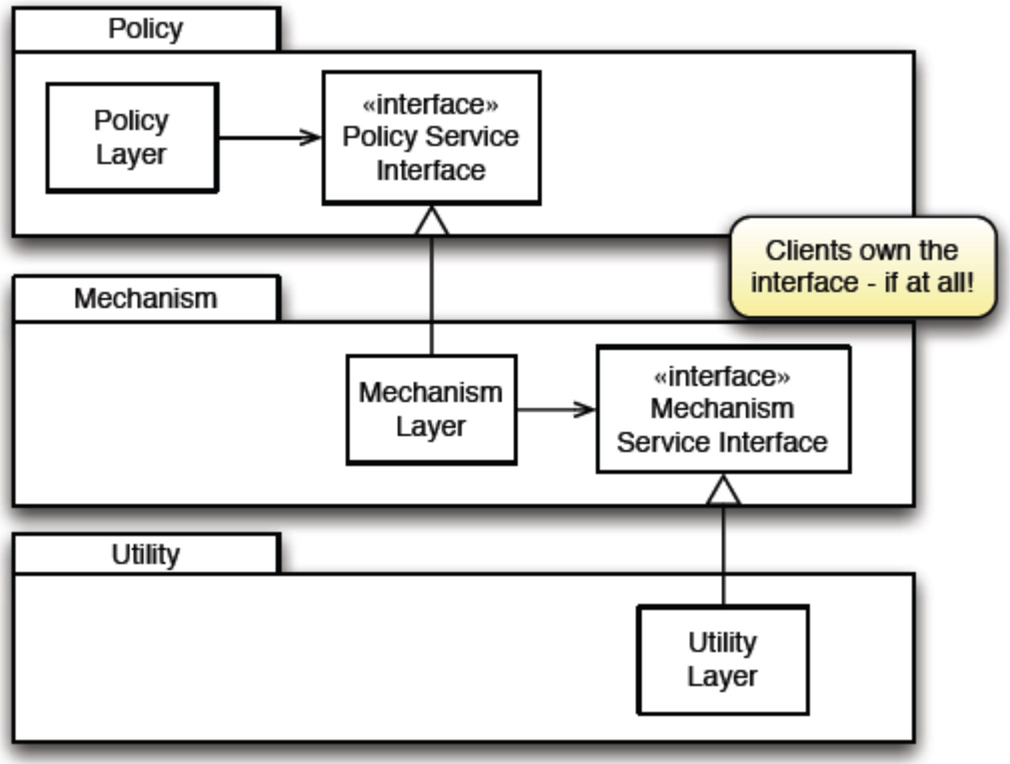
A possible interpretation of Booch's statement...



This interpretation clearly violates DIP. Higher-level modules depend on lower-level modules.

This is actually a typical structure of a layered architecture realized with structured programming.

Inverted Layer Dependencies



- Usually, we think of utility libraries as owning their own interfaces.
- A relict from structured programming era.
- Due to ownership inversion, `Policy` is unaffected by changes in `Mechanism` or `Utility`.

- An upper-layer declares (owns) interfaces for services it needs.
- Lower-layer implements these interfaces.
- Upper-layer uses lower-layer by the interface.
  The upper layer does not depend on the lower-layer.
- Lower-layer depends on the interface declared by the upper-layer.

2.6.4 Naive Heuristic for Ensuring DIP

DO NOT DEPEND ON A CONCRETE CLASS.

All relationships in a program should terminate on an abstract class or an interface.

- No class should hold a reference to a concrete class.
- No class should derive from a concrete class.
- No method should override an implemented method of any of its base classes.

Naive Heuristic for Ensuring DIP

DO NOT DEPEND ON A CONCRETE CLASS.

- This heuristic is usually violated at least once in every program.
- Some class will have to create concrete classes.
- Subclass relationships do often terminate at a concrete class.

- The heuristic seems naive for concrete stable classes, e.g., String in Java.
- But, concrete application classes are generally volatile, should not depend on them. Their volatility can be isolated:
  - by keeping them behind abstract interfaces
  - that are owned by clients.

2.6.5 Takeaway

High-level modules should not depend on low-level modules. Both should depend on abstractions.

- Traditional structural programming creates a dependency structure in which policy depends on detail.
  (Policies become vulnerable to changes in the details.)
- Object-orientation enables to invert the dependency:
  - Policy and details depend on abstractions.
  - Service interfaces are owned by their clients.
  - Inversion of dependency is the hallmark of good object-oriented design.
    (Implies an inversion of interface ownership.)
- Rationale behind DIP is arguable. For example, what if existing 3rd party (low-level) libraries are used?
  - One can argue that DIP enables an adapter layer which stops propagation of changes