CENG 443
Introduction to Object-Oriented Programming Languages and Systems

Object-Oriented Design Principles

The Object-Oriented Paradigm

- What is object-oriented (OO) paradigm?
  - provide a set of techniques for analysing, decomposing, and modularising software system architectures
  - characterized by structuring the system architecture on the basis of its *objects* (and classes of objects) rather than the *actions* it performs
- What is the rationale for using OO?
  - In general, systems evolve and functionality changes, but objects and classes tend to remain stable over time
  - Use it for **large systems**
  - Use it for **systems that change often**

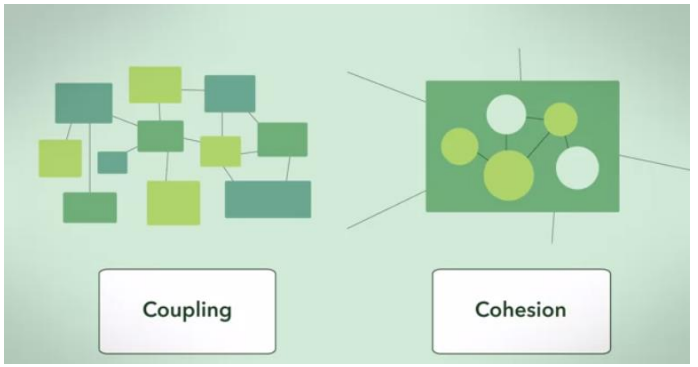
Signs of Rotting Design

1. Rigidity
   - code difficult to change
   - every change causes a cascade of subsequent changes in dependent modules
   - management reluctance to change anything becomes policy
2. Fragility
   - even small changes can cause cascading effects
   - code breaks in unexpected places
3. Immobility
   - code is so tangled that it's impossible to reuse anything
   - the software is simply rewritten instead of reused

Signs of Rotting Design

4. Viscosity
   - viscosity of the design
     - When the design preserving changes are harder to employ than the hacks, then the viscosity of the design is high.
     - It is easy to do the wrong thing, but hard to do the right thing.
   - viscosity of the environment
     - comes about when the development environment is slow and inefficient
     - if compile times are very long, engineers will be tempted to make changes that don't force large recompiles, even though those changes are not optimal from a design point of view

Causes of Rotting Design



- Dependency Management
  - coupling and cohesion
  - Coupling refers to the interdependencies between modules
    - Loose coupling vs tight coupling
  - Cohesion describes how related the functions within a single module are.
    - Low cohesion implies that a given module performs tasks which are not very related to each other and hence can create problems as the module becomes large.
  - It can be controlled!
    - create *dependency* firewalls so that dependencies do not propagate

SOLID Principles

- SRP – Single Responsibility Principle
  - A class should have only one reason to change
- OCP – Open / Closed Principle
  - Software entities should be open for extension, but closed for modification
- LSP – Liskov Substitution Principle
  - Inheritance should ensure that any property proved about supertype objects also holds for subtype objects
- ISP – Interface Segregation Principle
  - Clients should not be forced to depend on methods that they do not use.
- DIP – Dependency Inversion Principle
  - High-level modules should not depend on low-level modules.

From: Agile Software Development: Principles, Patterns, and Practices. Robert C. Martin, Prentice Hall, 2002

Single Responsibility Principle (SRP)

- A class should have only one reason to change

every module or class should have **responsibility** over a **single** part of the functionality provided by the software, and that **responsibility** should be entirely encapsulated by the class.

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.

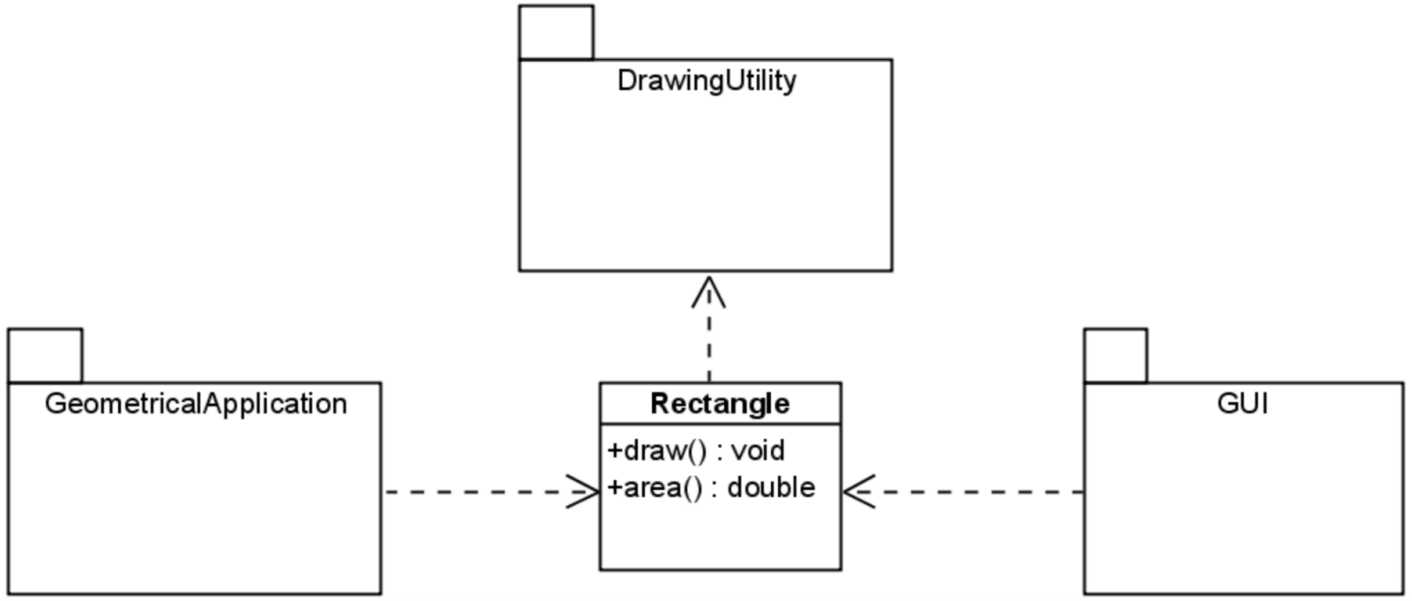
- A class with only one responsibility will have only one reason to change!

Responsibility and Cohesion

- **Cohesion measures the degree of togetherness among the elements of a class.**
- 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
- Therefore, **a class with low cohesion violates SRP**

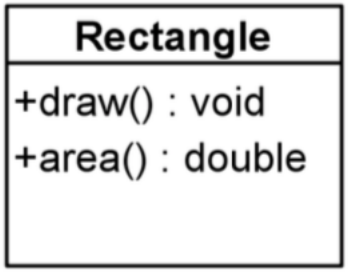
Example

- 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()).



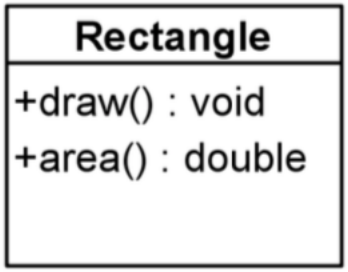
Problems of Rectangle

- 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.



Why is it a problem?

Problems of Rectangle

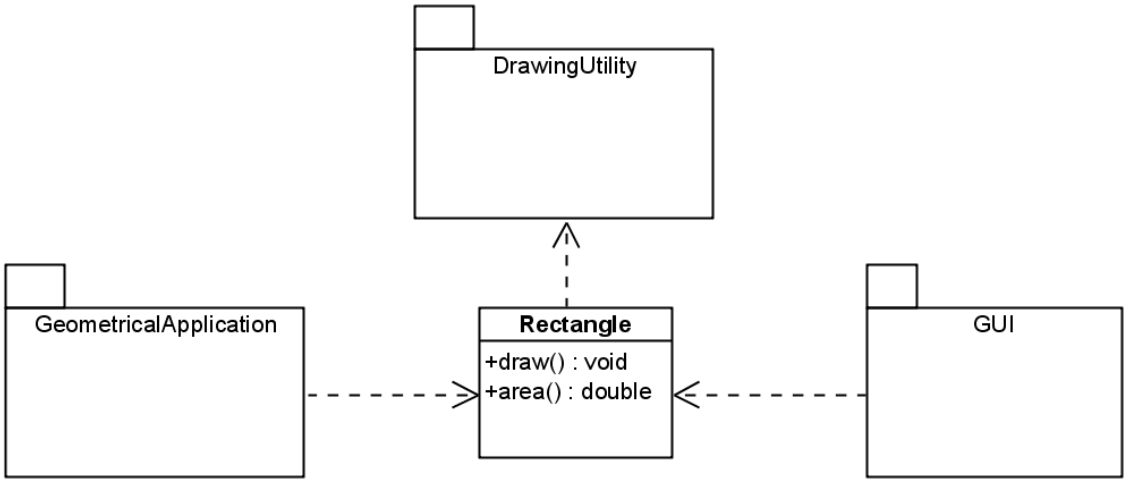


- **Rectangle is hard to use!**
  - It has multiple reasons to change.
  - Even if we want to use only one of its responsibilities, we must depend on both of them.
  - We inherit the effects of changes along every possible axis of change (= responsibility)

- **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

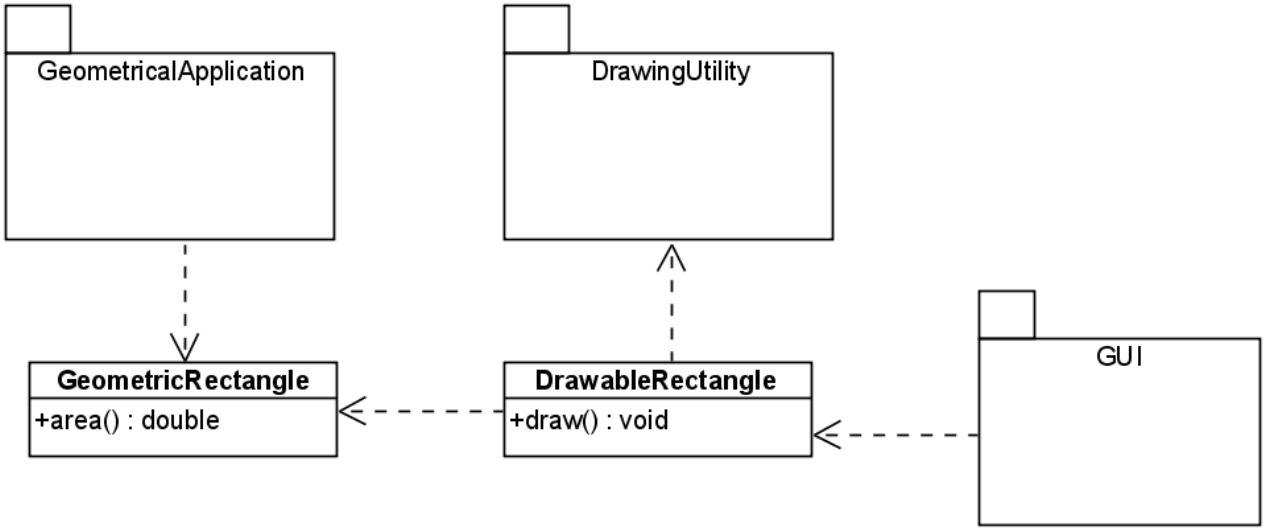
- Unnecessary dependency between GeometricApplication and DrawingUtility (DrawingUtility classes have to be deployed along with Rectangle) even if we only want to use the geometrical functions of rectangles.



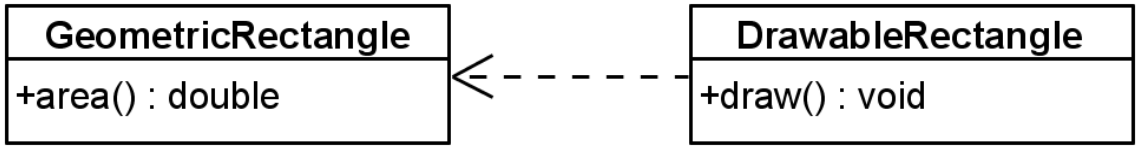
- **Problem**: If drawing functionality changes in the future, we need to retest Rectangle also in the context of GeometricalApplication!

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.



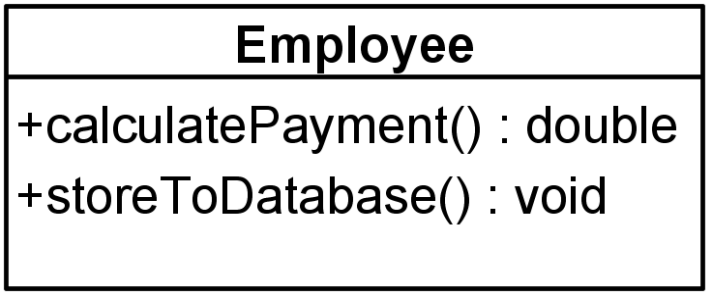
Two Classes with High Cohesion



- **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.

Employee Example

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

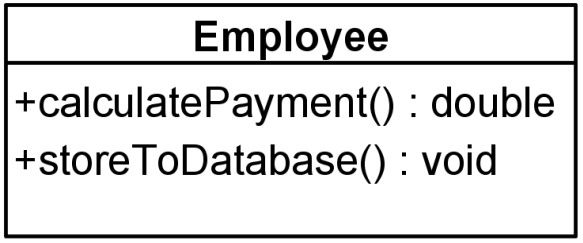


Should we split responsibilities?

Employee Represents a Typical SRP-Violation

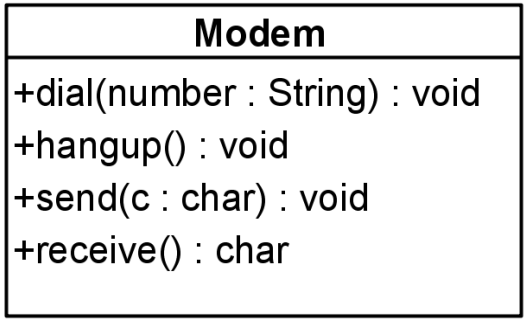
- Calculating the payment of an employee is part of the **business rules**.
  - It corresponds to a real-world concept the application shall implement.
- Storing the employee information in the database is a **technical aspect**.
  - It is a necessity of the IT architecture that we have selected; does 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 unpredictable.

Most probably we should split in this case.



Modem Example

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



Should we split?

To Split or Not to Split Modem?

- Break down the question to:
  1. Do we expect connection management and data communication to constitute **different axes of change?**
  2. Do we expect them to change together, or independently.
  3. Will these responsibilities be **used by different clients?**
  4. Do we plan to provide **different configurations** of modems **to different customers?**

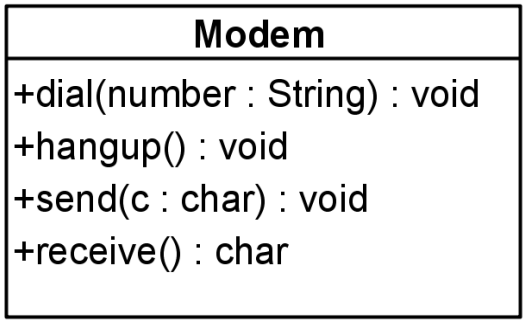
To Split or Not to Split Modem?

- **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.

Modem Example

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



Should we split?

To Apply or Not to Apply

- Decide based on the nature of responsibilities:
  - changed together / not changed 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.

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 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.
- **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
  - Force developers to build the abstractions needed for testability
- 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

Summary

- 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.

- 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

Open-Closed Principle (OCP)

- "*Software Systems change during their life time*"
  - both better designs and poor designs have to face the changes;
  - good designs are stable

> *Software entities should be open for extension, but closed for modification*
>
> **B. Meyer**, 1988 / quoted by **R. Martin**, 1996

- Be open for extension
  - module's behavior can be extended
- Be closed for modification
  - source code for the module must not be changed
- *Modules should be written so they can be extended without requiring them to be modified*

Extension and Modification

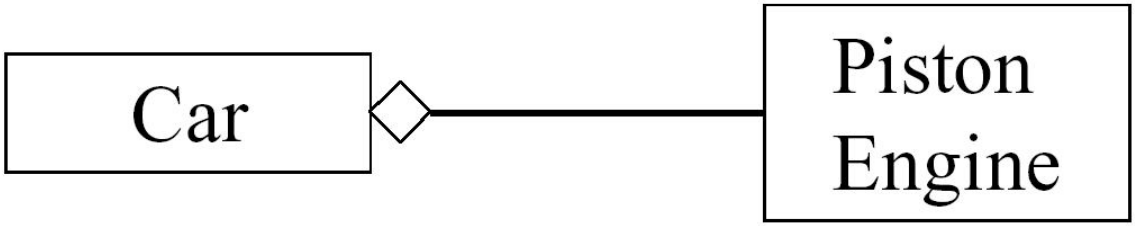
- **Extension**: Extending the *behavior* of an module.
- **Modification***:* Changing the *code* of an module.

- **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?

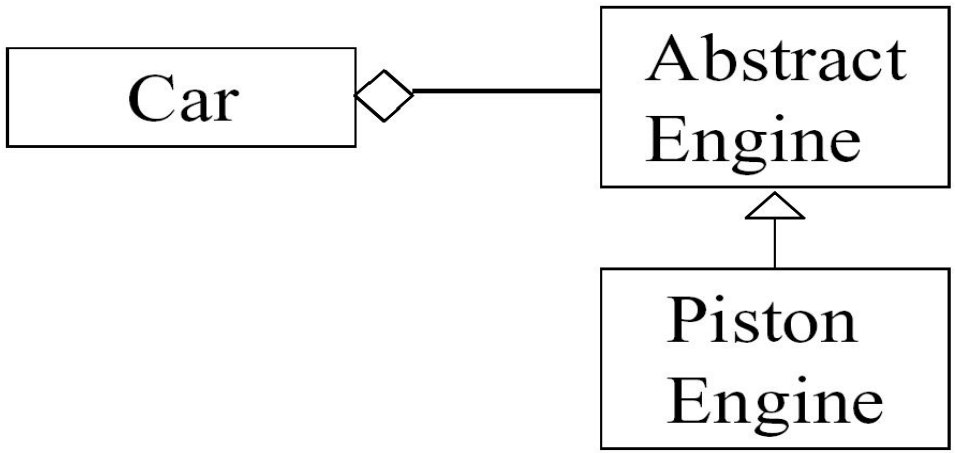
- Question: Why not simply change the code if I needed?
  1. Module was **already delivered to customers**, a change will not be accepted.
     - If you need to change something, hopefully you opened your module for extension!
  2. Module is a **third-party library only available as binary code**.
     - If you need to change something, hopefully the third-party opened the module for extension!
  3. **Most importantly**: not changing existing code for the sake of implementing extensions enables incremental compilation, testing, debugging.

Open the door ...



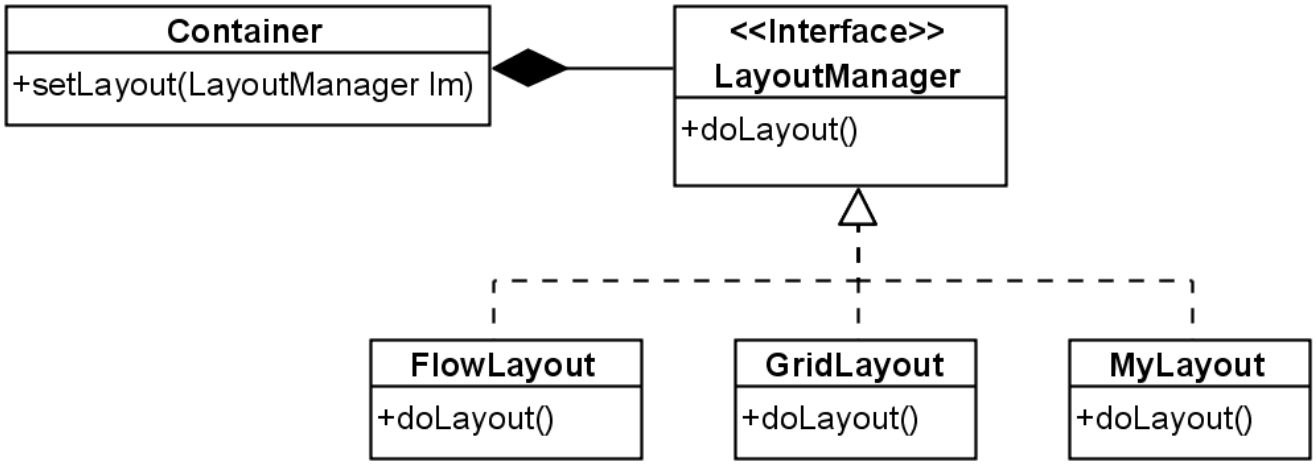
- How to make the Car run efficiently with a TurboEngine?
- Only by changing the Car in the given design!

… But Keep It Closed!



- A class must not depend on a concrete class!
- It must depend on an abstract class using polymorphic dependencies (calls)
- Abstraction is the key

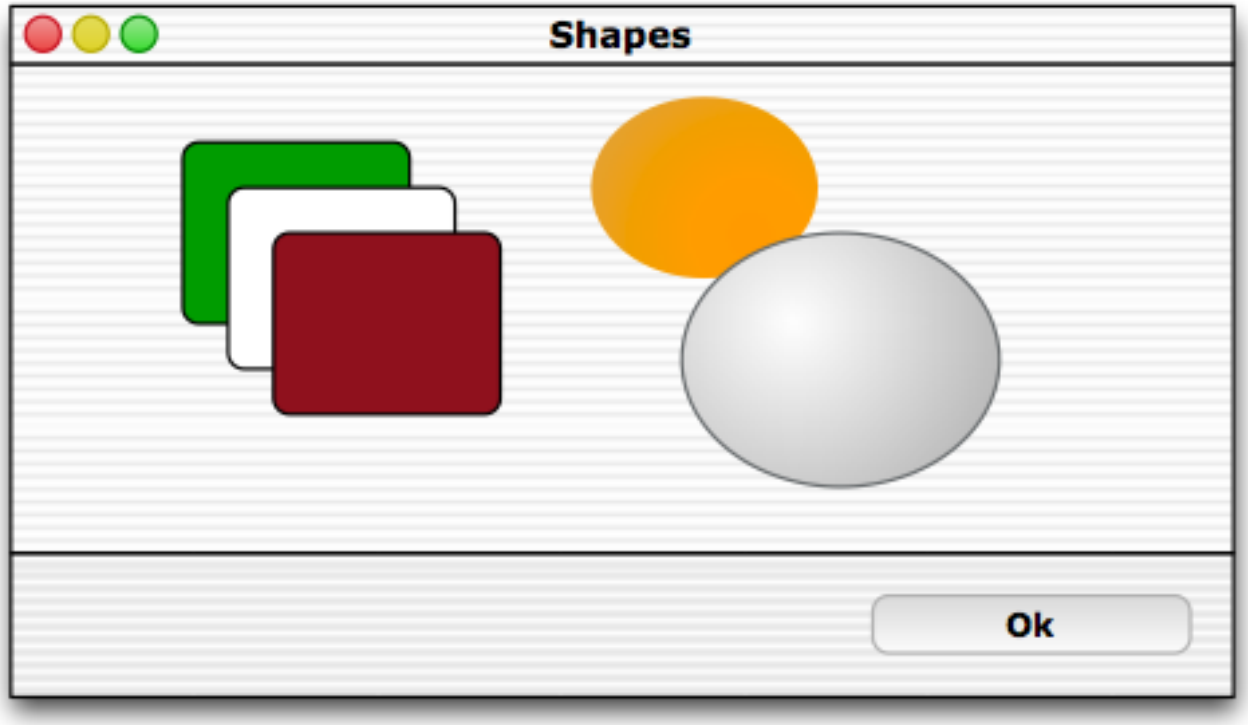
Abstracting over Variations



- 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.

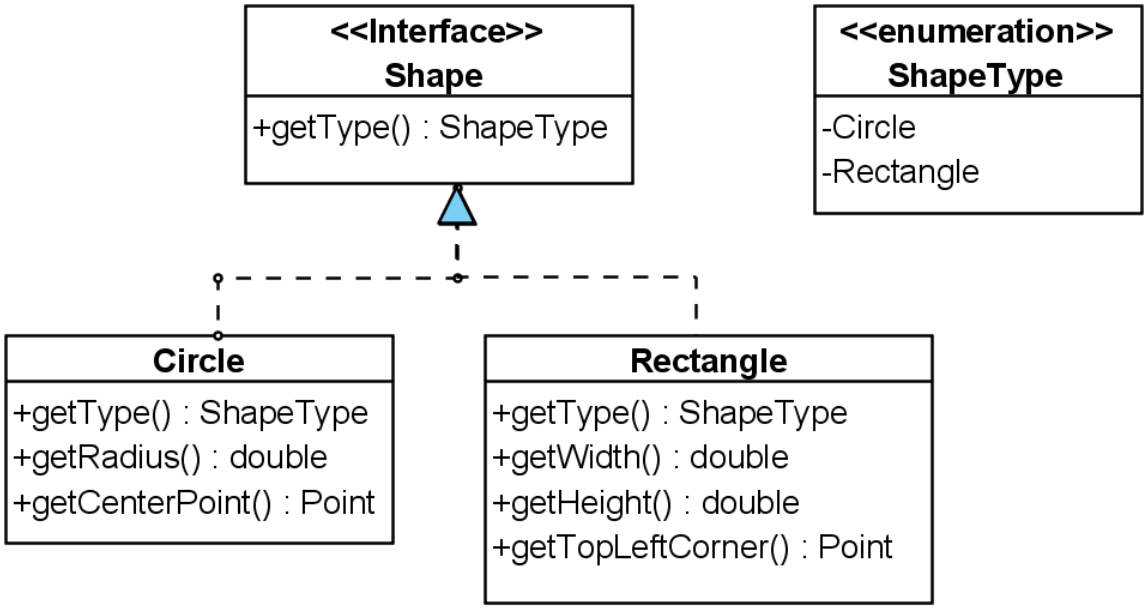
OCP by Example

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



A Possible Design

- 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

- 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) {
...
}
```

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

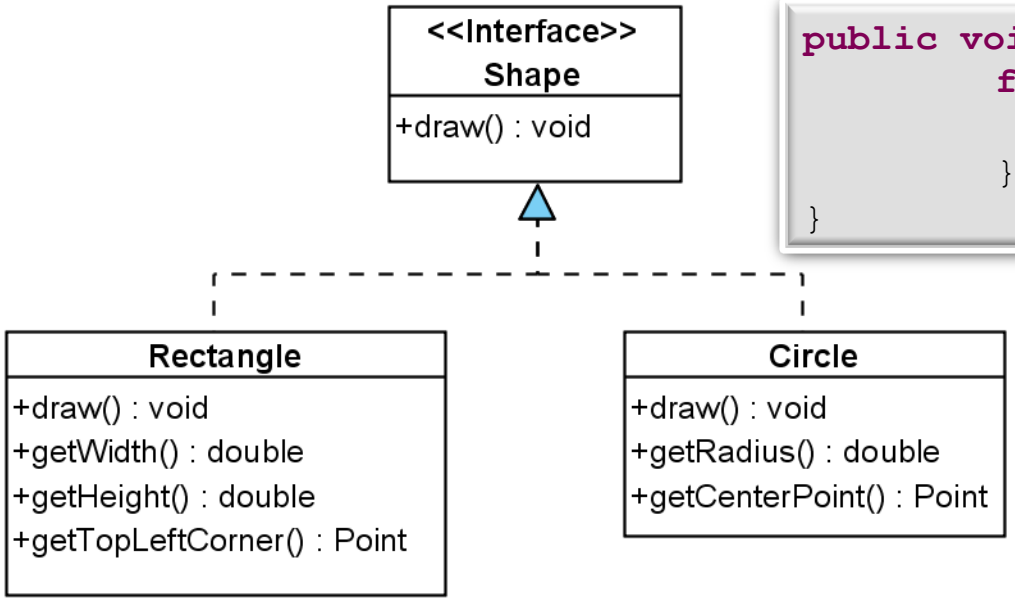
- **Rigid designs** are hard to change – every change causes many changes to other parts of the system.
  - Our example design is rigid: Adding a new shape causes many existing classes to be changed.
- **Fragile designs** tend to break in many places when a single change is made.
  - Our example design 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.

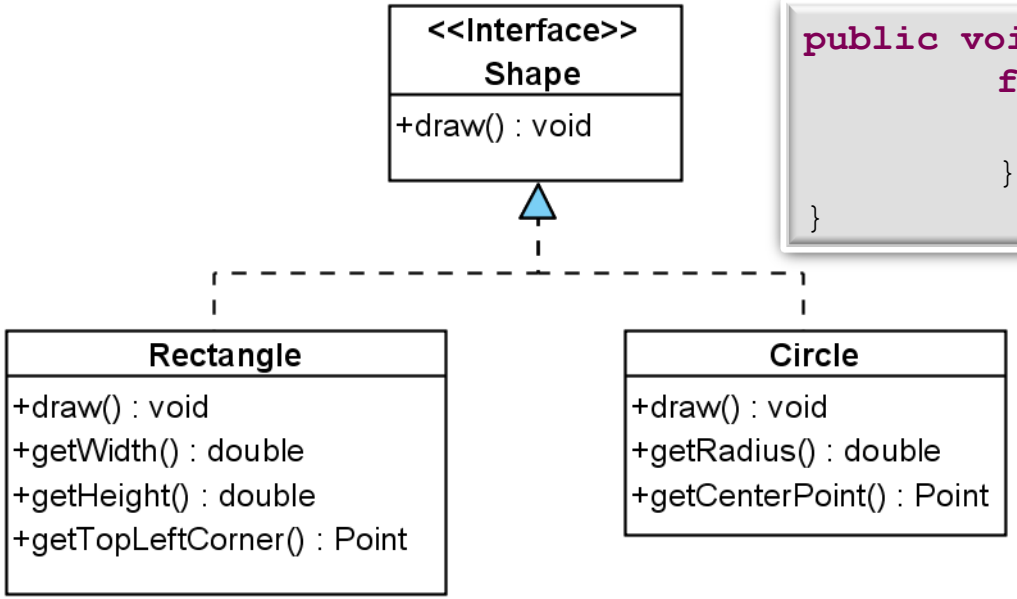
An Alternative Design




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

- **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.

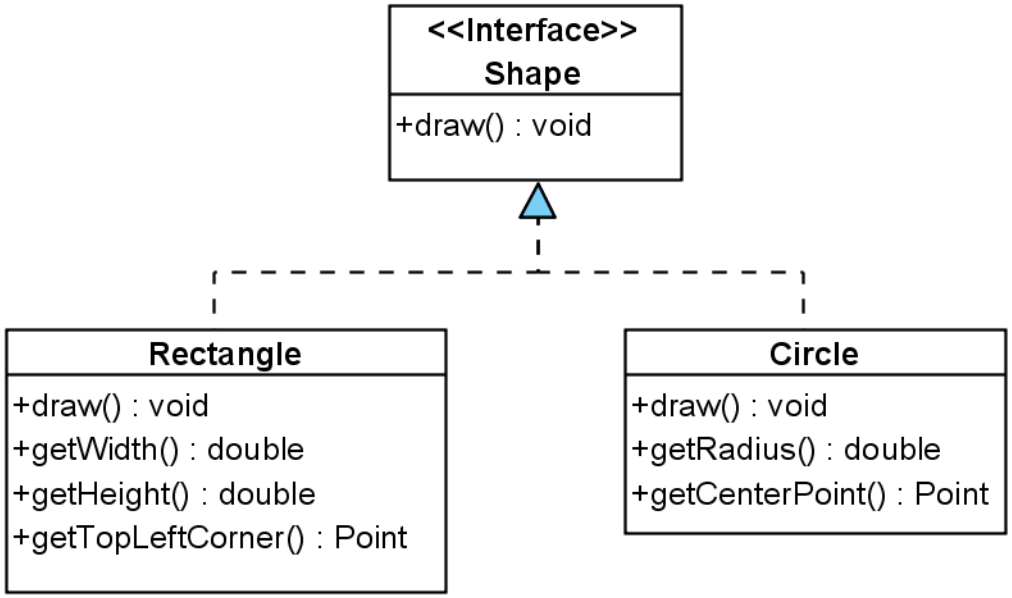
An Alternative Design




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

Problematic Changes

- 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.



Abstractions are the key

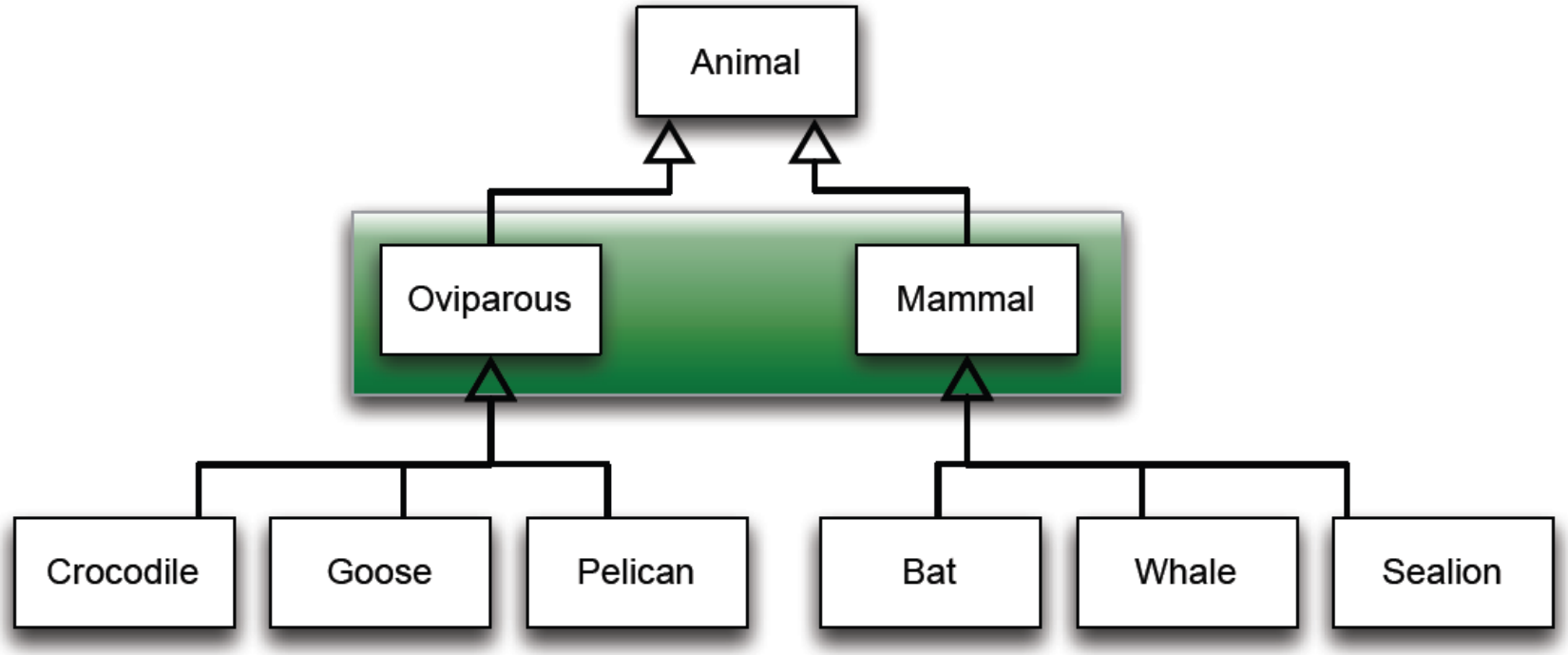
- May Support or Hinder Change
  - Change is easy if change units correspond to abstraction units
  - Change is tedious if they do not correspond to abstraction units
- Reflect a viewpoint
  - No matter how ”open” a module is, there will always be some kind of change that requires modification
    - 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!

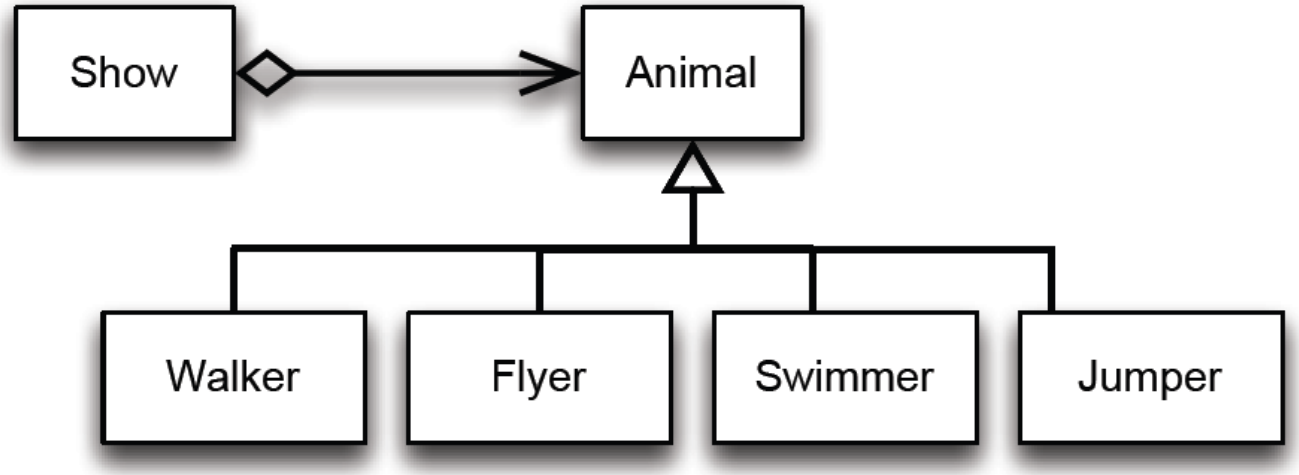
One Possible Class Hierarchy



The veterinary surgeon has won !

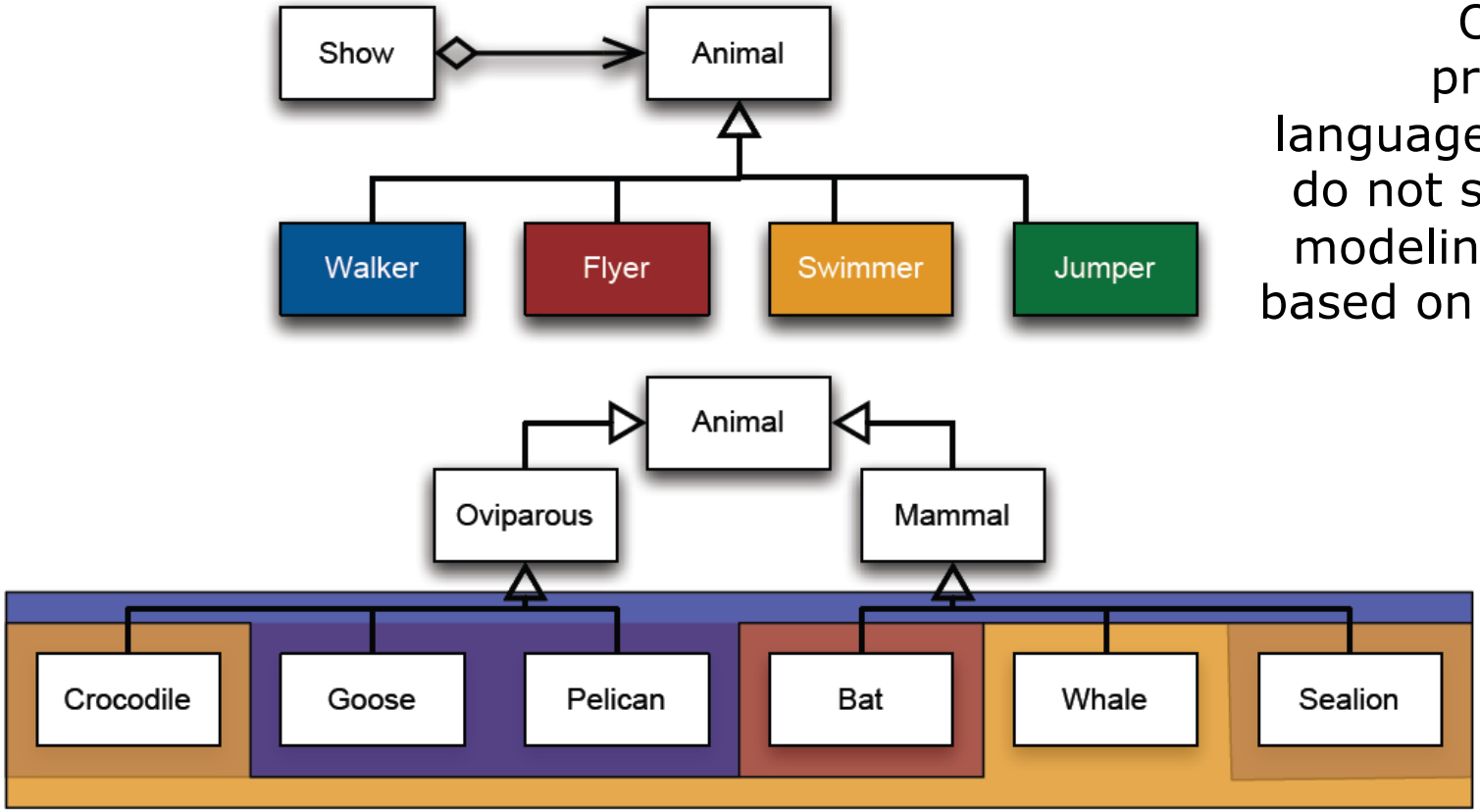
The World from Trainer's Viewpoint

*"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 Takeaway

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

Strategic Closure

"*No significant program can be 100% closed* "

R.Martin, *"The Open-Closed Principle," 1996*

- Closure not *complete* but *strategic*

- Closure Against What?
  - You have to choose which changes you'll isolate yourself against.
  - What if we have to draw all circles first? Now DrawAllShapes must be edited (or we have to hack something)
- Opened Where?
  - Somewhere, someone has to instantiate the individual shapes.
  - It's best if we can keep the dependencies confined

Takeaway

- 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).

OCP Heuristics

Make all object-data private
No Global Variables!

- Changes to public data are always at risk to "open" the module
  - They may have a rippling effect requiring changes at many unexpected locations;
  - Errors can be difficult to completely find and fix. Fixes may cause errors elsewhere.

OCP Heuristics (2)

RTTI is Ugly and Dangerous!

- RTTI is ugly and dangerous
  - If a module tries to dynamically cast a base class pointer to several derived classes, any time you extend the inheritance hierarchy, you need to change the module
  - recognize them by type **`switch`**-es or **`if-else-if`** structures
- Not all these situations violate OCP all the time
  - when used only as a "filter"

Liskov Substitution Principle (LSP)

- Subtypes must be behaviorally substitutable for their base types.

–Barbara Liskov, 1988 (ACM Turing Award Receiver)

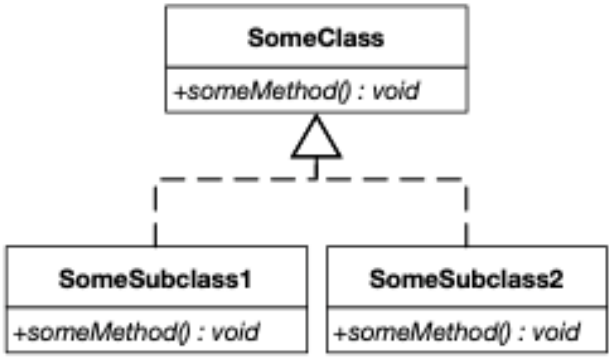
The Liskov Substitution Principle

- Class inheritance and subtype polymorphism as primary mechanisms for supporting the open-closed principle (OCP) in object-oriented designs
- The Liskov Substitution Principle
  - ... 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 the open-closed principle.

Substitutability

- In object-oriented programs, subclasses are substitutable for superclasses in client code:

```
void clientMethod(SomeClass sc)
{
    …
    sc.someMethod();
    …
}
```



- In clientMethod, sc may be an instance of SomeClass or any of its subclasses.
  Hence, if clientMethod works with instances of SomeClass, it does so with instances of any subclass of SomeClass. They provide all methods of SomeClass and eventually more.

Example

```
class Rectangle {

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

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

  public void area() { return height * width;}

}
```

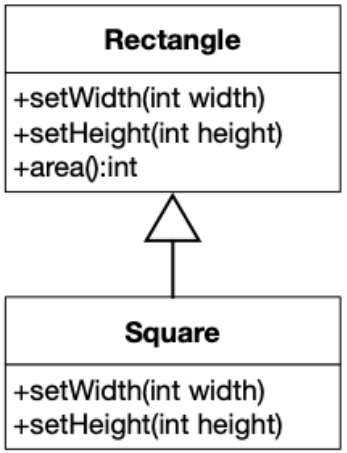
Assume that we want to implement a class Square and want to maximize reuse.

```
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);
  }

}
```



What do you think about this design?

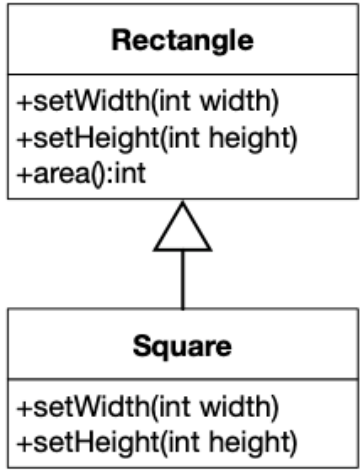
Critique

```
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);
  }

}
```



- Is it mathematically correct?
  - A square has the mathematical properties of a rectangle: A square has four edges and only right angles and is therefore a rectangle.
- Is this implementation correct?
  - With this overriding of setHeight and setWidth – to set both dimensions to the same value – instances of Square remain mathematically valid squares.
- Does Java type system allow substitution?
  - We can pass Square wherever Rectangle is expected, as far as the Java type system is concerned.
- **But, we may break assumptions that clients of Rectangle make about the "behavior" of a Rectangle.**

The client is the keyword here!

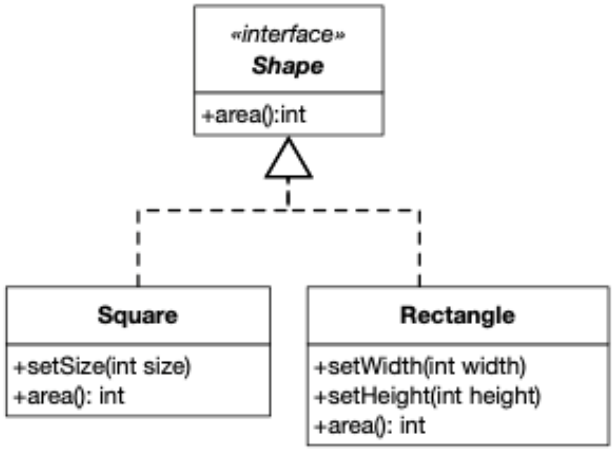
```
void clientMethod(Rectangle rec)
{
  rec.setWidth(5);
  rec.setHeight(4);
  assert(rec.area() == 20);
}
```

- This client that works with Rectangle but not with Square
- The clientMethod method makes an assumption that is true for Rectangle
  - setting the width respectively height has no effect on the other attribute. This assumption does not hold for Square.
- The Rectangle/Square hierarchy violates the Liskov Substitution Principle (LSP)!
  - Square is behaviorally not a correct substitution for Rectangle.
- A Square does not comply with the behavior of a rectangle:
  - Changing the height/width of a square behaves differently from changing the height/width of a rectangle.
  - Actually, it doesn't make sense to distinguish between the width and the height of a square.

Interim Takeaway Messages

- Programmers do not define entities that are something, but entities that behave somehow.
- A model viewed in isolation can not be meaningfully validated!
- The validity of a model depends on the clients that use it.
  - Inspecting the Square/Rectangle hierarchy in isolation did not show any problems. In fact it even seemed like a self-consistent design. We had to inspect the clients to identify problems.
- The validity of a model must be judged against the possible uses of the model. We need to anticipate the assumptions that clients will make about our classes.

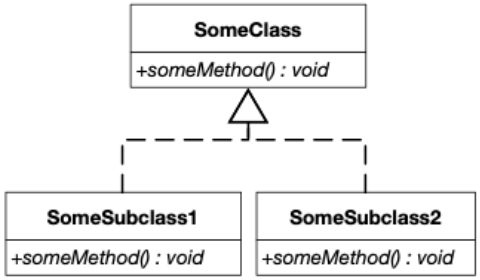
LSP-compliant Design for Rectangle/Square



- Clients of Shape cannot make any assumptions about the behavior of setter methods.
- When clients want to change properties of the shapes, they have to work with the concrete classes.
- When clients work with the concrete classes, they can make true assumptions about the computation of the area.

Behavioral Substitutability

- What does the Liskov Substitution Principle add to the common object-oriented subtyping rules?



- It’s not enough that instances of SomeSubclass1 and SomeSubclass2 provide all methods declared in SomeClass. These methods should also behave like their heirs!
- A client method should not be able to distinguish the behavior of objects of SomeSubclass1 and SomeSubclass2 from that of objects of SomeClass.

Design by Contract

- Solution to the validation problem: A technique for explicitly stating what may be assumed.
- 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.
- We can specify contracts using Pre-, Post-Conditions and Invariants. They must be respected by subclasses and clients can rely on them.

Behavioral Subtyping Rules

- Rule for Preconditions:
  - Preconditions of a class imply preconditions of its subclasses.
  - Preconditions may be replaced by (equal or) **weaker** ones.
- Rule for Postconditions:
  - Postconditions of a class are implied by those of its subclasses.
  - Postconditions may be replaced by equal or **stronger** ones.
- Rule for Invariants:
  - can be replaced by **stronger** ones

• 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.

• 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.

BigNatural vs BigInteger

```
//@mathmodel n integer
//@constraint n >= 0
interface BigNatural {

  //@alters n
  //@ens n = #n+1
  void increment();

  //@alters n
  //@ens n=max(0,#n-1)
  void decrement();
}
```

```
//@mathmodel n integer
//@constraint
interface BigInteger {

  //@alters n
  //@ens n = #n+1
  void increment();

  //@alters n
  //@ens n = #n-1
  void decrement();
}
```

Should BigNatural extend BigInteger?

For behavioral subtyping, ask

- Is BigNatural's invariant *stronger*?
- Do all BigNatural methods *require less*?
- Do all BigNatural methods *ensure more*?

```
//@mathmodel n integer
//@constraint n >= 0
interface BigNatural {

  //@alters n
  //@ens n = #n+1
  void increment();

  //@alters n
  //@ens n=max(0,#n-1)
  void decrement();
}
```

```
//@mathmodel n integer
//@constraint
interface BigInteger {

  //@alters n
  //@ens n = #n+1
  void increment();

  //@alters n
  //@ens n = #n-1
  void decrement();
}
```

Is BigNatural's invariant *stronger*?

- BigNatural invariant is n >= 0
- BigInteger invariant is true

Yes

```
//@mathmodel n integer
//@constraint n >= 0
interface BigNatural {
```

```
//@mathmodel n integer
//@constraint
interface BigInteger {
```

Do all BigNatural methods *require less*?

- increment() requires the same (true) in both
- decrement() requires the same (true) in both

Yes

```
interface BigNatural {
  //@alters n
  void increment();
  //@alters n
  void decrement();
}
```

```
interface BigInteger {
  //@alters n
  void increment();
  //@alters n
  void decrement();
}
```

Do all BigNatural methods *ensure more*?

- BigNatural decrement() ensures #n>0 ==> n=#n-1
- BigInteger decrement() ensures n=#n-1

No

```
interface BigNatural {
  //@ens n = #n+1
  void increment();

  //@ens n=max(0,#n-1)
  void decrement();
}
```

```
interface BigInteger {
  //@ens n = #n+1
  void increment();

  //@ens n = #n-1
  void decrement();
}
```

Postconditions cannot be weakened in the derivative.
(You cannot guarantee less than the parent.)

Example violating client

- noop() is correct for BigInteger, but not for BigNatural

```
BigInteger noop(BigInteger i) {
  i.decrement();
  i.increment();
  return i;
}
```

Java Modeling Language

- In JML, specifications are written as Java annotation comments to the Java program, which hence can be compiled with any Java compiler.
- To process JML specifications several tools exist:
  - an assertion-checking compiler (jmlc) which performs runtime verification of assertions,
  - a unit testing tool (jmlunit),
  - an enhanced version of javadoc (jmldoc) that understands JML specifications and
  - an extended static checker ([ESC/Java](http://en.wikipedia.org/wiki/ESC/Java)) a static verification tool that uses JML as its front-end.

```
public class Rectangle implements Shape {

  private int width;
  private int height;

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

Straightforward examples of violations of the Liskov Substitution Principle

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

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

- In both cases, composition would have been preferrable.

Takeaway messages

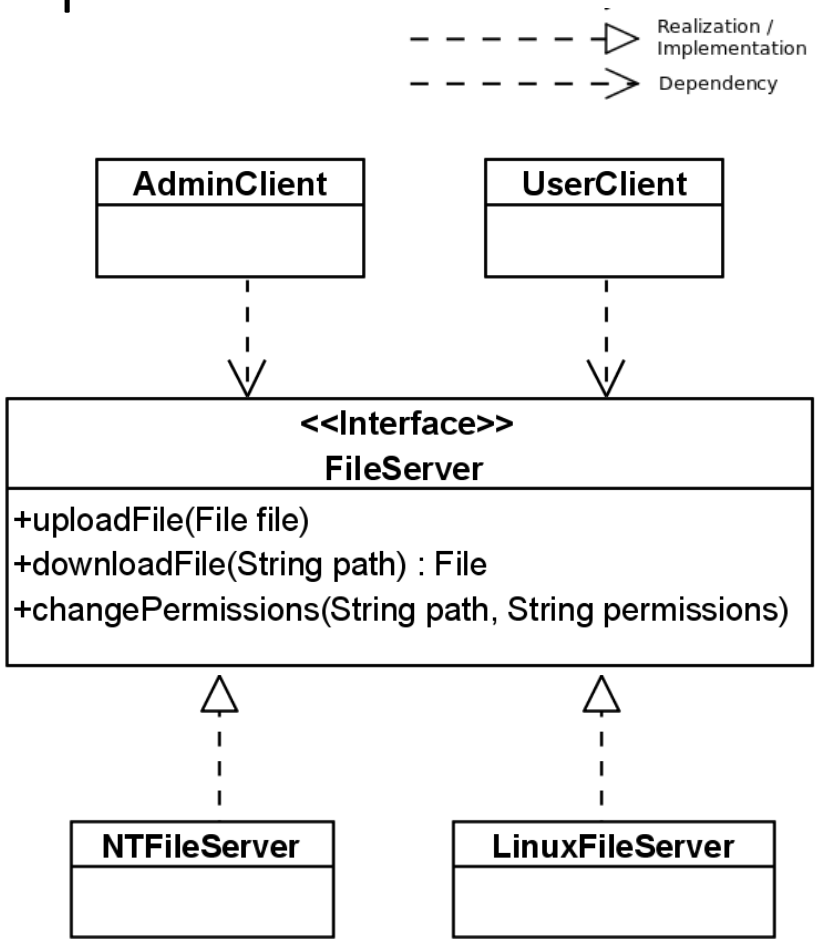
- Behavioral subtyping extends “standard” OO subtyping. LSEP Additionally ensures that assumptions of clients about the behavior of a base class are not broken by subclasses.
- Behavioral subtyping helps with supporting OCP. LSEP Only behavioral subclassing (subtyping) truly supports open-closed designs.
- 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)

Interface Segregation Principle (ISP)

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

Introduction to ISP by Example

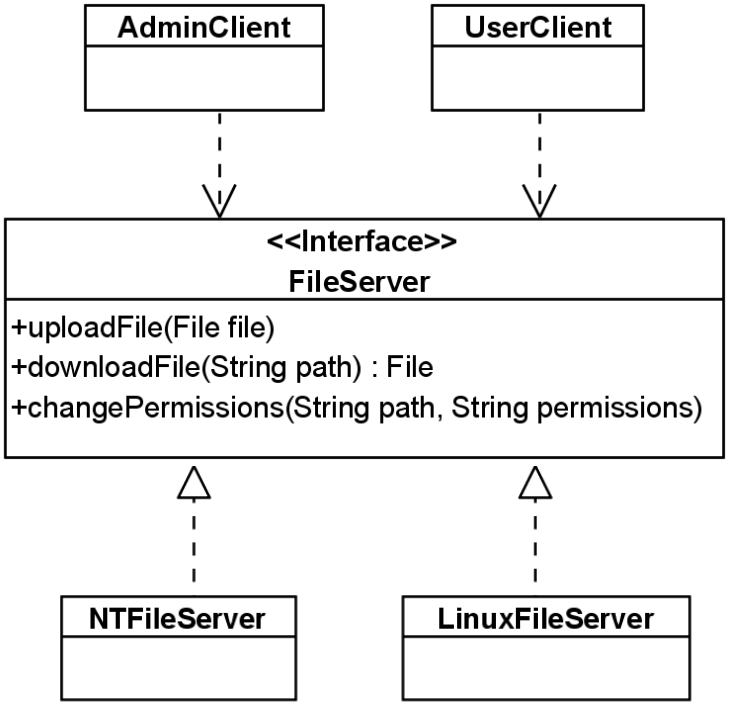
- Consider the design of a file server system.
- The interface FileServer declares methods provided by any file server.
- Various classes implement this interface for different operating systems.
- Two clients are implemented for the file server:
  - AdminClient, uses all methods.
  - UserClient, uses only the upload/download methods



Any problems?

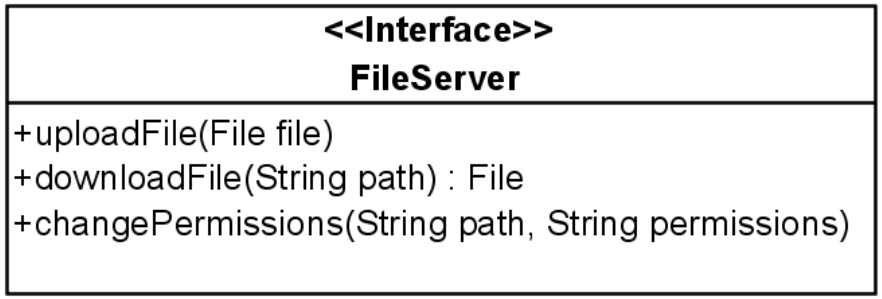
Problems of the Proposed Design

- 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.

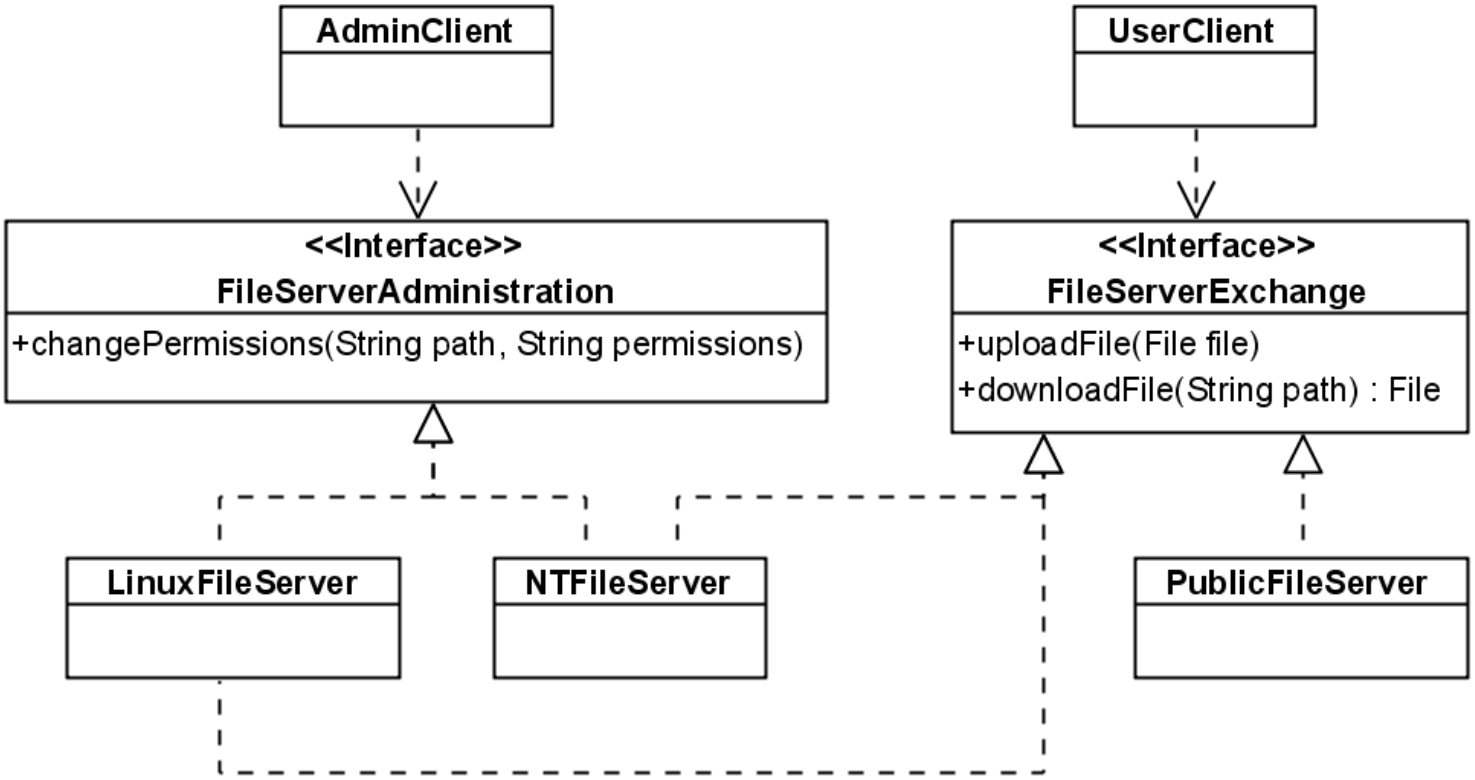


A Polluted Interface

- 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.



An ISP Compliant Solution



Proliferation of Interfaces

- ISP should not be overdone!
  Otherwise you will end up with 2^n-1 interfaces for a class with n methods.

- 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.

Takeaway

- 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.

Dependency Inversion Principle

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

II. Abstractions should not depend on details.
Details should depend on abstractions

R. Martin, 1996

- OCP states the goal; DIP states the mechanism

- A base class in an inheritance hierarchy should not know any of its subclasses
- Modules with detailed implementations are not depended upon, but depend themselves upon abstractions

The Rationale of DIP

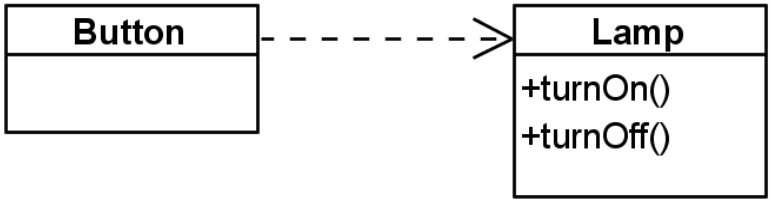
- **High-level, low-level Modules.**
- Good software designs are structured into modules.
  - High-level modules contain the important policy decisions and business models of an application
    - The identity of the application.
  - Low-level modules contain detailed implementations of individual mechanisms 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

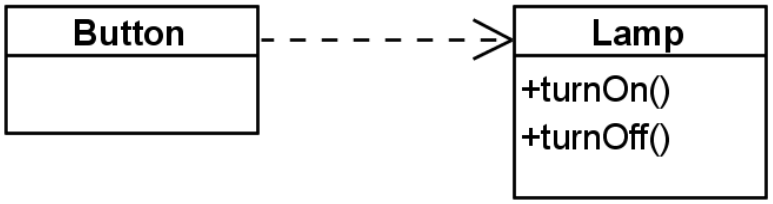
Introduction to DIP by Example



- Consider a design excerpt from a 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.

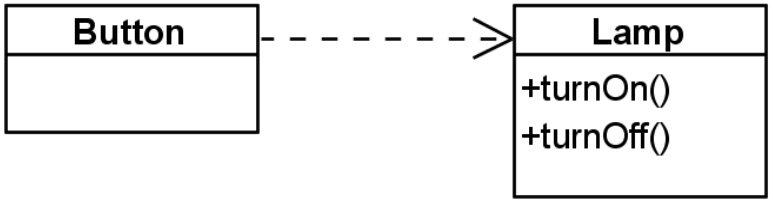
Any problems?

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.

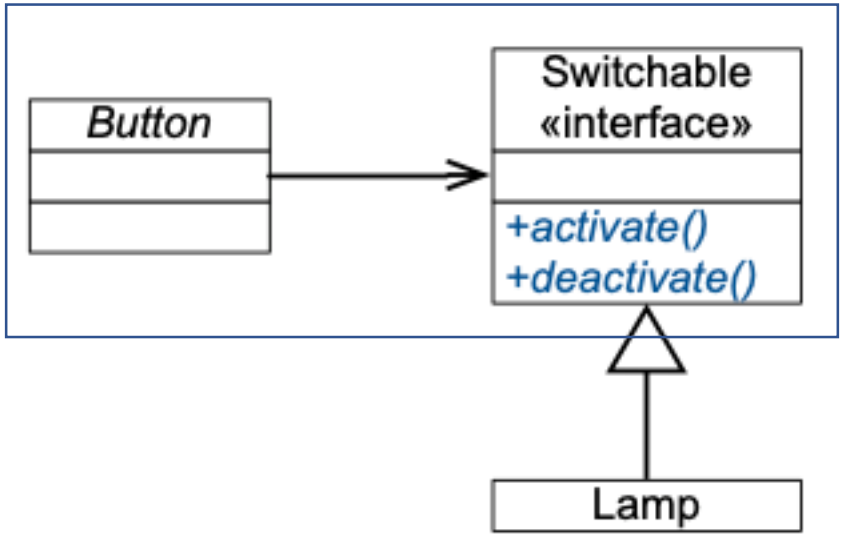
The High-Level Policy



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.

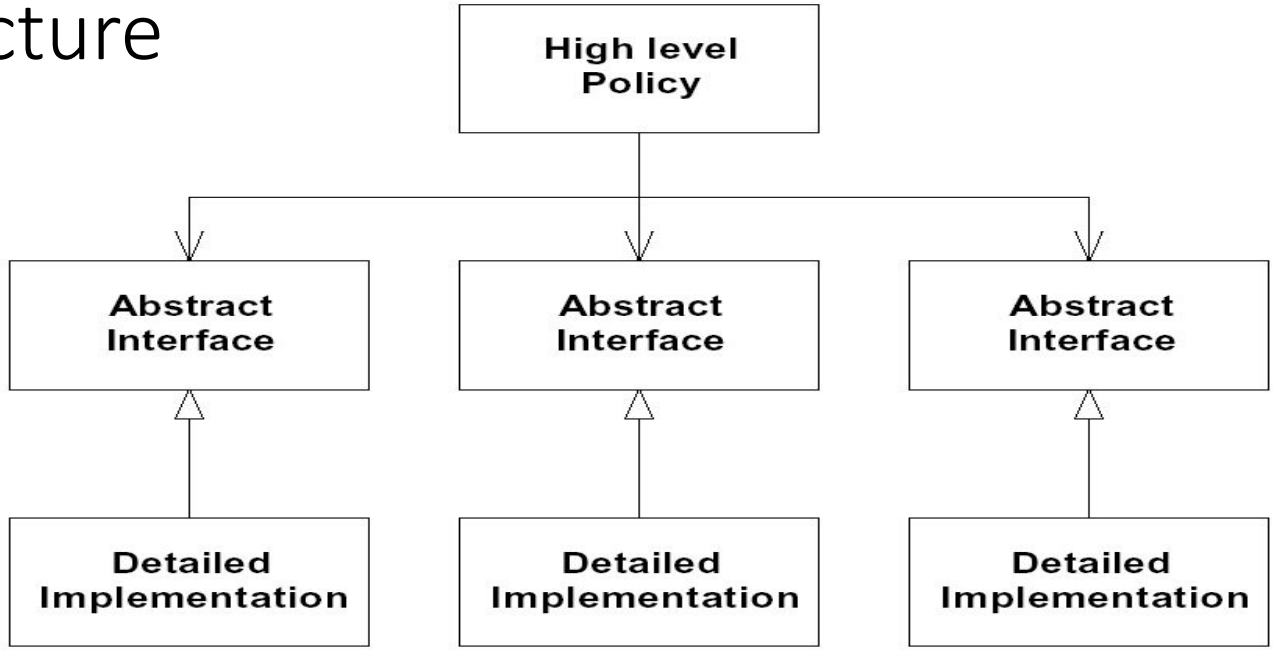
A DIP-Compliant Solution



The high level policy

- Now Button only depends on abstractions!
  It can be reused with various classes that implement Switchable.
- 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!

OO Architecture

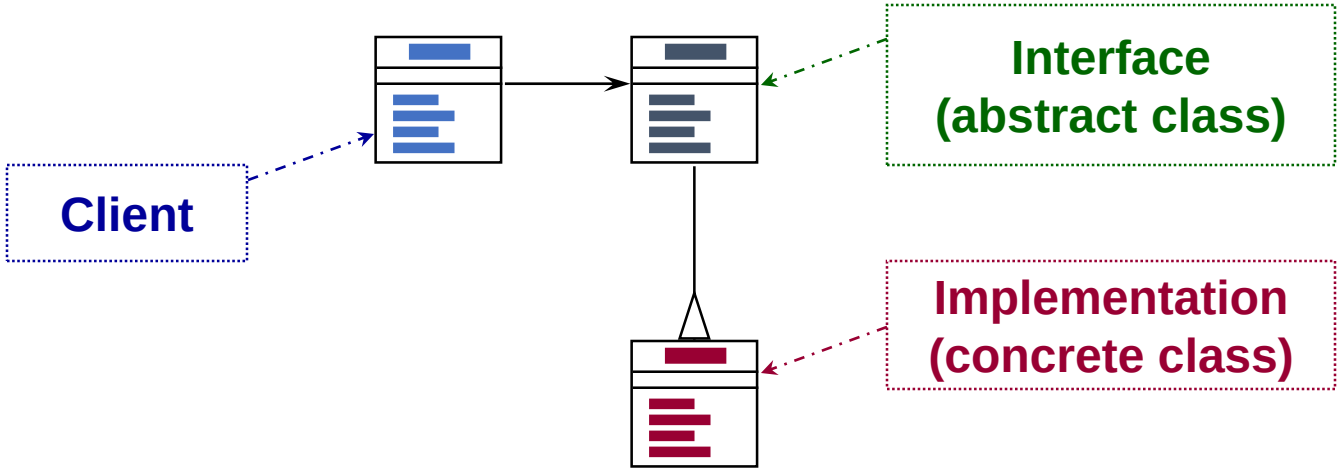


- Good software designs are structured into High-level, low-level modules.
  - High-level modules contain the important policy decisions and business models of an application – The identity of the application.
  - Low-level modules contain detailed implementations of individual mechanisms needed to realize the policy.

DIP Related Heuristic

**Design to an interface,
not an implementation!**

- Use inheritance to avoid direct bindings to classes:



Design to an Interface

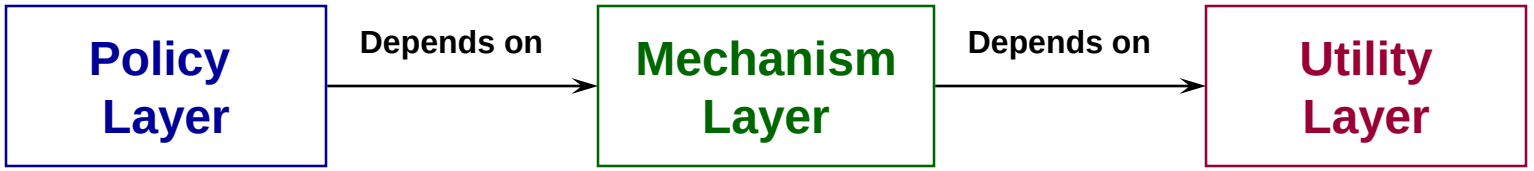
- **Abstract classes/interfaces:**
  - tend to change much less frequently
  - abstractions are ‘hinge points’ where it is easier to extend/modify
  - shouldn’t have to modify classes/interfaces that represent the abstraction (OCP)

- Exceptions
  - Some classes are very unlikely to change;
    - therefore little benefit to inserting abstraction layer
    - Example: String class
  - In cases like this can use concrete class directly
    - as in Java or C++

DIP Related Heuristic

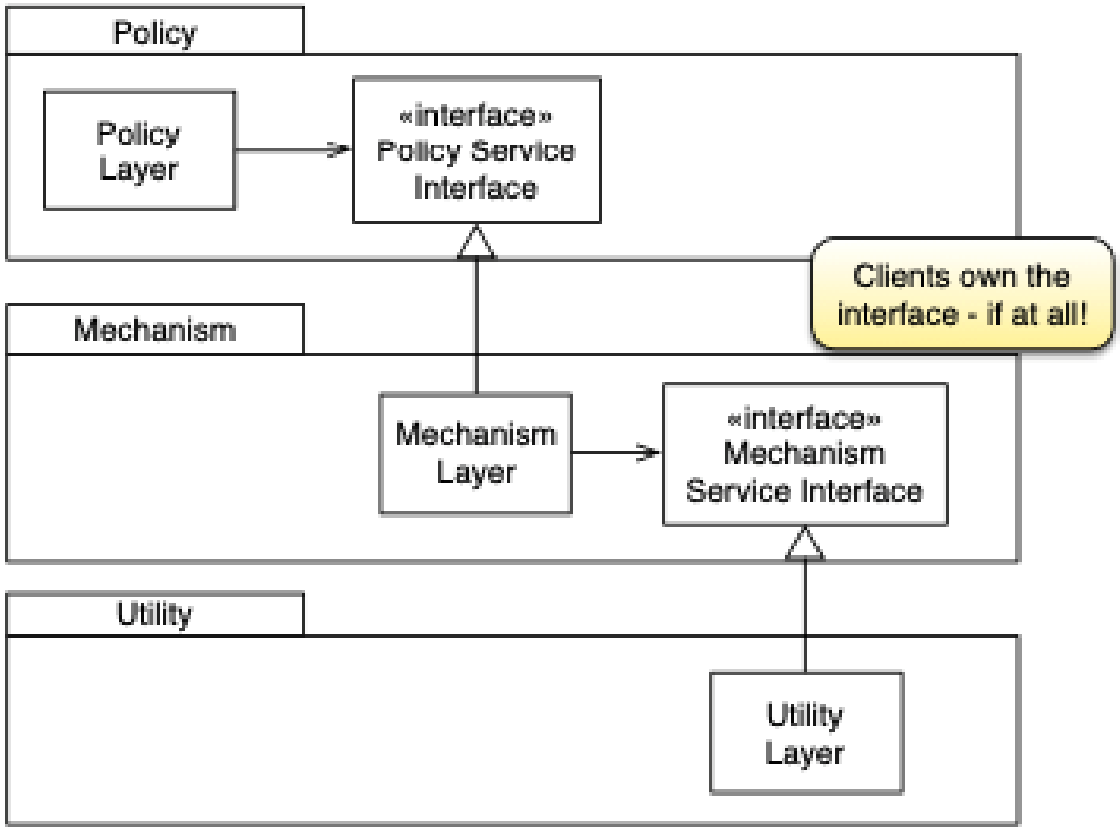
Avoid Transitive Dependencies

- Avoid structures in which higher-level layers depend on lower-level abstractions:
  - In example below, Policy layer is ultimately dependant on Utility layer.



Solution to Transitive Dependencies

- Use inheritance and abstract ancestor classes to effectively eliminate transitive dependencies:



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.

Takeaway

- 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.

The Founding Principles

- The three principles are closely related
- Violating either LSP or DIP invariably results in violating OCP
  - LSP violations are latent violations of OCP
- It is important to keep in mind these principles to get most out of OO development...