SUBTOPIC 10

PRINCIPLES OF OBJECT-ORIENTED DESIGN

Pedro J. Ponce de León

English version by Juan Antonio Pérez

Version 20111202

Department of Computer Languages and Systems

Universitat d'Alacant
Universidad de Alicante

Contents

Introduction. Object-oriented design

- Designing software
  - In the context of software, **design** is a problem-solving process whose objective is to find and describe a way to implement the system's functional requirements, while respecting the constraints imposed by the quality, platform and process requirements (including the budget and deadlines), and while adhering to general principles of good quality.

Introduction. Object-oriented design

- Object-oriented software design
  - OO design is the process of problem solving and planning for a software solution by using not only general software design principles, but also specific principles of OO software and OO developing platforms (languages, frameworks, libraries, etc.).
  - The main goal of design principles is to improve the internal and external factors of software quality (see subtopic 1).
  - Design will assume the existence of concepts such as encapsulation, polymorphism (dynamic binding), inheritance, etc.

General principles of design

- Overall goals
  - Increasing profit by reducing cost and increasing revenue. For most organizations, this is the central objective. However, there are a number of ways to reduce cost, and also many different ways to increase the revenue generated by software.
  - Ensuring that we actually conform to the requirements, thus solving the customers' problems.
  - Accelerating development. This helps reduce short-term costs, helps ensure the software reaches the market soon enough to compete effectively, and may be essential to meet some deadline faced by the customer.
  - Increasing qualities such as usability, efficiency, reliability, maintainability and reusability.

General principles of design

Review (from subtopic 3): coupling and cohesion

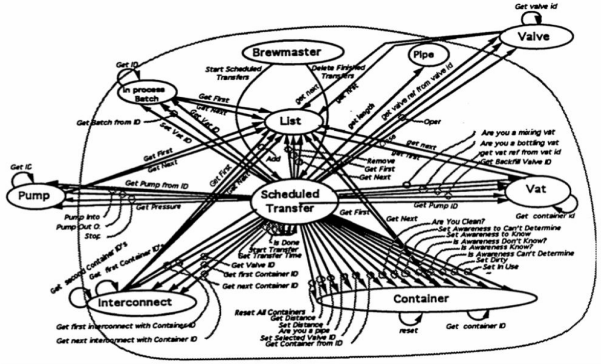
- **Coupling** describes the relationship between software components.
  - Low coupling (desirable) may be attained by moving a task into the list of responsibilities of the component that holds the necessary data.
- **Cohesion** is the degree to which the responsibilities of a single component form a meaningful unit.
  - High cohesion (desirable) is achieved by associating in a single component tasks that are related in some manner; e.g. the necessity to access a common data value.

General principles of design

Review (from subtopic 3)

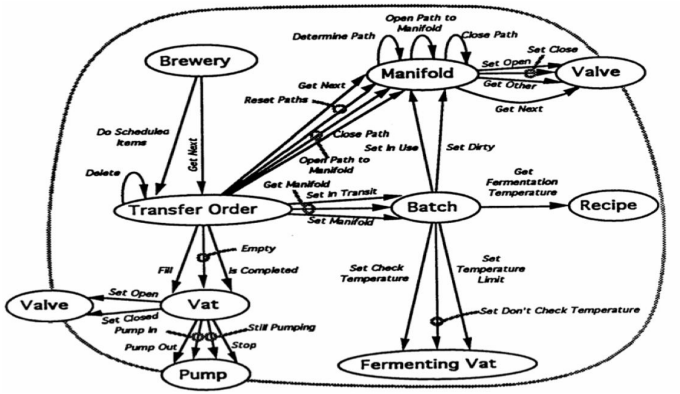
- **Low coupling** is often a sign of a well-structured computer system and a good design, and when combined with **high cohesion**, supports the general goals of high readability and maintainability.

General principles of design



sistema demasiado acoplado

sistema desacoplado y más cohesionado



Acoplamiento

Cohesión

Principles of object-oriented design

1. The open-closed principle
2. The Liskov substitution principle
3. The single responsibility principle
4. The interface segregation principle
5. The dependency inversion principle

Open-closed principle

- “All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version” (1)

- Open-closed principle
  - ***Software entities (classes, modules, methods, etc.) should be open for extension, but closed for modification”*** (2)
  - You should design modules that **never change**. When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

(1) Object Oriented Software Engineering a Use Case Driven Approach, Ivar Jacobson, Addison Wesley, 1992, p 21.

(2) Object Oriented Software Construction, Bertrand Meyer, Prentice Hall, 1988, p 23

Open-closed principle

- Using the principles of object oriented design, it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviors.
- The abstractions are abstract base classes, and the unbounded group of possible behaviors is represented by all the possible derivative classes.
- It is possible for a module to manipulate an abstraction. Such a module can be closed for modification since it depends upon an abstraction that is fixed. Yet the behavior of that module can be extended by creating new derivatives of the abstraction.

Open-closed principle

- Example

We have an application that must be able to draw circles and squares on a standard GUI. A list of the circles and squares will be created in the appropriate order and the program must walk the list in that order and draw each circle or square.

Solution without RTTI or dynamic binding:

```
enum TipoFigura { circulo, cuadrado }
class Figura {
  private TipoFigura tipo;
  public getTipo() { return tipo; }
}
class Circulo extends Figura {
  private Punto centro;
  private float radio;
  public void dibujarCirculo()  { ... }
}
class Cuadrado extends Figura {
  private Punto puntoSuperiorIzqda;
  private float lado;
  public void dibujarCuadrado()  { ... }
}
```

```
// client code (bad smell!)

void dibujarFiguras(List<Figura> lf) {
  Iterator<Figura> it = lf.iterator();
  while (it.hasNext() {
    Figura fig = it.next();
    switch(fig.getTipo()) {
      case circulo:
        Circulo ci = (Circulo)fig;
        ci.dibujarCirculo(); break;
      case cuadrado:
        Cuadrado cu = (Cuadrado)fig;
        cu.dibujaCuadrado();  break;
    }
  }
}
```

Open-closed principle

- Example

  DibujarFiguras() does not conform to the open-closed principle because it cannot be closed against new kinds of shapes.

  Moreover, adding a new shape to the application means hunting for every place that such switch statements (or if/else chains) exist, and adding the new shape to each.

Open-closed principle

- Example

Solution with RTTI:

```
class Figura { }
class Circulo extends Figura {
  private Punto centro;
  private float radio;
  public void dibujarCirculo()  { … }
}
class Cuadrado extends Figura {
  private Punto puntoSuperiorIzqda;
  private float lado;
  public void dibujarCuadrado()  { … }
}
```

```
// client code (bad smell!)

void dibujarFiguras(List<Figura> lf) {
  Iterator<Figura> it = lf.iterator();
  while (it.hasNext() {
    try {
      Circulo ci = (Circulo)it.next(); // unsafe
downcasting
      Cuadrado cu = (Cuadrado)it.next();
     ci.dibujarCirculo();
     cu.dibujarCuadrado();
    } catch (ClassCastException cce) { … }
  }
}
```

dibujarFiguras() does not conform to the open-closed principle yet

(RTTI must be used only when the open-closed principle is not violated)

Open-closed principle

- Example

Solution which conforms to the open-closed principle

```
abstract class Figura {
  abstract public void dibujar();
}
class Circulo extends Figura {
  private Punto centro;
  private float radio;
  public void dibujar()  { ... }
}
class Cuadrado extends Figura {
  private Punto puntoSuperiorIzqda;
  private float lado;
  public void dibujar()  { ... }
}
```

```
// client code

void dibujarFiguras(List<Figura> lf) {
  Iterator<Figura> it = lf.iterator();
  while (it.hasNext() {
    Figura fig = it.next();
    fig.dibujar();
 }
}
```

dibujarFiguras() conforms to the open-closed principle. Its behavior can be extended without modifying it by adding new derivative classes.

Open-closed principle

- Since programs that conform to the open-closed principle are changed by adding new code, rather than by changing existing code, they do not experience the cascade of changes exhibited by non-conforming programs.

- It should be clear that no significant program can be 100% closed. For example, consider what would happen to the dibujarFiguras() method if we decided that all circles should be drawn before any squares. The dibujarFiguras() method is not closed against a change like this.

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

Open-closed principle

- The designer must choose the kinds of changes against which to close his design. This takes a certain amount of prescience derived from experience.

- Code closure can be attained in this case by using some kind of “ordering abstraction” (*)

(*) Find the solution here: http://www.objectmentor.com/resources/articles/ocp.pdf

Open-closed principle

- **Heuristics**
  - Encapsulation: attributes should be declared private, rather than public or protected.
    - When the member variables of a class change, every function that depends upon those variables must be changed. Thus, no function that depends upon a variable can be closed with respect to that variable.
  - No global variables... ever.
    - No module that depends upon a global variable can be closed against any other module that might write to that variable.
    - However, there are cases where the convenience of a global is significant. Global variables such as System.out are common examples.
  - Using RTTI may be dangerous (see the example in p. 12)

Open-closed principle

- **Conclusions**
  - *Conformance to this principle is what yields the greatest benefits claimed for object oriented technology; i.e. reusability and maintainability.*
  - *Yet conformance to this principle is not achieved simply by using an object oriented programming language. Rather, it requires a dedication on the part of the designer to apply abstraction to those parts of the program that the designer feels are going to be subject to change.*

The Liskov Substitution Principle

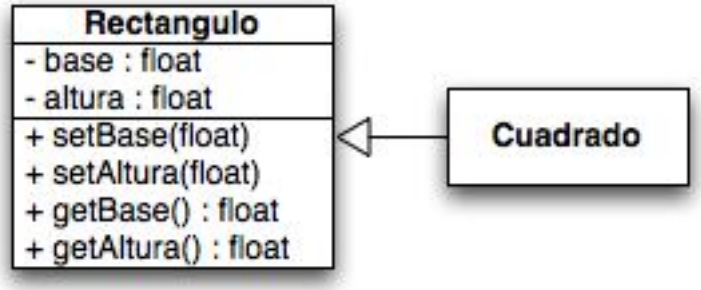
- See subtopic 3 (interface inheritance)
- First definition:
  - “If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T." - Barbara Liskov, Data Abstraction and Hierarchy, SIGPLAN Notices, 1998.
- When she says "the behavior of P is unchanged", she doesn't mean that the program has the same effect on the world; she means that the program doesn't have to condition its interaction with the objects on whether they are o1 or o2. To the client program, both objects fulfill the contract, so the client can ignore the differences between them.

The Liskov Substitution Principle

- **The Liskov substitution principle**
  - Alternative formulation:
    - Methods that use references to base classes must be able to use objects of derived classes without knowing it.
  - The importance of this principle becomes obvious when you consider the consequences of violating it.
    - If there is a method which does not conform to this principle, then it uses a reference to a base class, but must know about all the derivatives of that base class. Such a function violates the open-closed principle because it must be modified whenever a new derivative of the base class is created.

The Liskov Substitution Principle

- **Example**



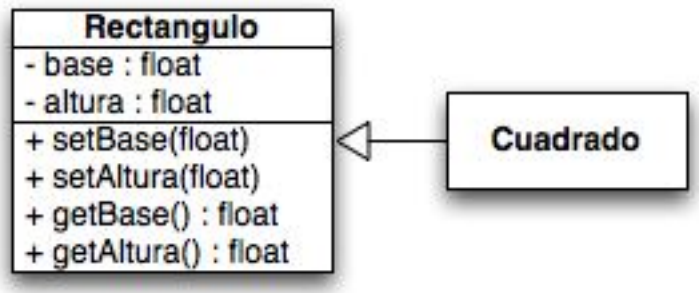
A square IS A rectangle with its base identical to its height.

Actually, we do not need two attributes for a square; the methods setBase() and setAltura() are not appropriate for squares. To sidestep the problem, we could override both as follows:

```
void setBase(float b) {
  super.setBase(b);
  super.setAltura(b);
}

void setAltura(float a) {
  super.setBase(a);
  super.setAltura(a);
}
```

The Liskov Substitution Principle



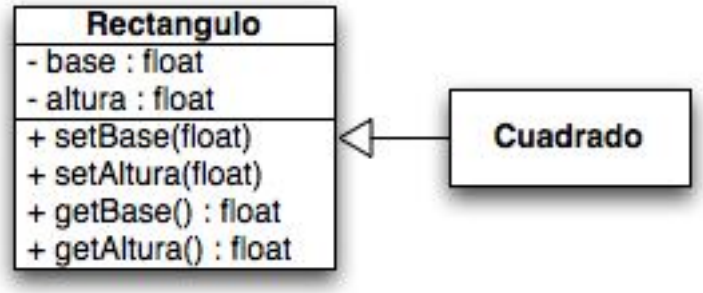
We might conclude that the model is now self consistent and correct. However...

```
// client code, unit test
void testSetters(Rectangulo r) {
  r.setBase(5);
  r.setAltura(4);
  assertEquals("Setters",r.getBase() * r.getAltura(), 20);
}
```

Here is the real problem: Was the programmer who wrote that test justified in assuming that changing the width of a Rectangulo leaves its height unchanged? This is a very reasonable assumption, but passing a Cuadrado to this method will result in problems. Therefore, there exist functions that take references to Rectangulo objects, but cannot operate properly upon Cuadrado objects.

The behavior of a Cuadrado object is not consistent with the behavior of a Rectangulo object. Behaviorally, a Cuadrado IS not A Rectangle! And it is behavior that software is really all about. This principle makes clear that in OOD the ISA relationship pertains to behavior. Not intrinsic private behavior, but extrinsic public behavior; behavior that clients depend upon.

The Liskov Substitution Principle



Design by contract (1)

It provides mechanisms for ensuring that the substitution principle holds.

Using this scheme, methods of classes declare preconditions and postconditions.

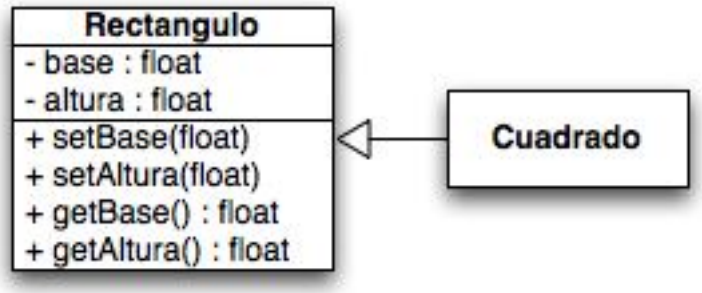
The preconditions must be true in order for the method to execute. Upon completion, the method guarantees that the postcondition will be true.

One possible postcondition for Rectangulo.setAltura() could be

```
assert( (altura == a) && (base == old.base) )
```

(1) *Object Oriented Software Construction*, Bertrand Meyer, Prentice Hall, 1988

The Liskov Substitution Principle



Design by contract (cont'd)

Now the rule for the preconditions and postconditions for derivatives, as stated by Meyer, is: “when redefining a method in a derivative class, you may only replace its precondition by a weaker one, and its postcondition by a stronger one.”

Thus, derived objects must not expect users to obey preconditions that are stronger than those required by the base class. That is, they must accept anything that the base class could accept. Also, derived classes must conform to all the postconditions of the base. That is, their behaviors and outputs must not violate any of the constraints established for the base class.

In our example the postcondition in Cuadrado.setAltura() will not contain (base == old.base), thus violating the contract of the base class.

The dependency inversion principle

- **The definition of a “bad design” in software**
  - It is hard to change because every change affects too many other parts of the system. (**Rigidity**)
  - When you make a change, unexpected parts of the system break. (**Fragility**)
  - It is hard to reuse in another application because it cannot be disentangled from the current application. (**Immobility**)

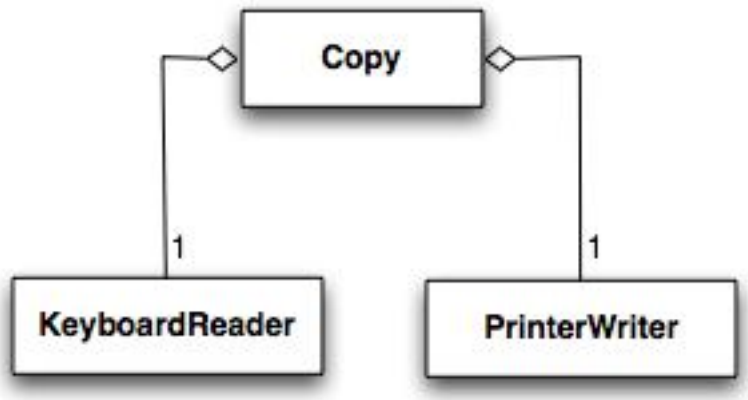
- What is it that makes a design rigid, fragile and immobile? It is the interdependence of the modules within that design.

The dependency inversion principle

- **Interdependence of the modules is the cause of “bad design”**
  - **Rigidity:** a single change to heavily interdependent software begins a cascade of changes in dependent modules. This makes the cost of the change impossible to predict. Managers, faced with such unpredictability, become reluctant to authorize changes.
  - **Fragility:** often the new problems are in areas that have no conceptual relationship with the area that was changed. Such fragility greatly decreases the credibility of the design and maintenance organization.
  - **Immobility:** the desirable parts of the design to be changed are highly dependent upon other details that are not desired.

The dependency inversion principle

- **Example: the 'Copy' module**




```
void Copy()
{
  int c;

  while ((c = KeyboardReader.read()) != EOF)
       PrinterWriter.write(c)
}
```

- Copy() is not reusable in any context which does not involve a keyboard or a printer.

The dependency inversion principle

- **Example: the 'Copy' module**

Consider a new program that copies keyboard characters to a disk file.

```
enum OutputDevice {printer, disk}

void Copy(OutputDevice dev)
{
        int c;
        while ((c = KeyboardReader.read()) != EOF)
                if (dev == printer)
                        PrinterWriter.write(c);
                else DiskWriter.write(c);
}
```

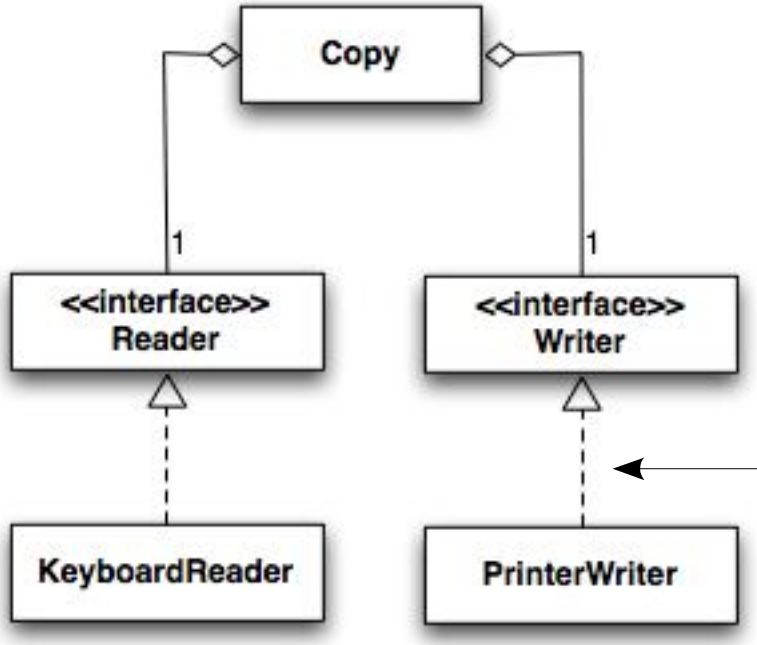
This adds new interdependencies to the system. As time goes on, and more and more devices must participate in the copy program, the “Copy” module will be littered with if/else statements and will be dependent upon many lower level modules. It will eventually become rigid and fragile.

The dependency inversion principle

- **Example: the 'Copy' module**

Dependency inversion

The module containing the high level policy, the Copy() module, is dependent upon the low level detailed modules. We should find a way to make it independent of these details.



The dependency inversion principle

- **Example: the 'Copy' module**

```
interface Reader
{
 int read();
}

interface Writer
{
 void write(char);
}

void Copy(Reader r, Writer w)
{
  int c;
  while((c = r.read()) != EOF)
    w.write(c);
}
```

Dependencies have been inverted: Copy() now depends on the interfaces (abstractions), in the same way as concrete readers and writers.

The dependency inversion principle

- **Example: the 'Copy' module**

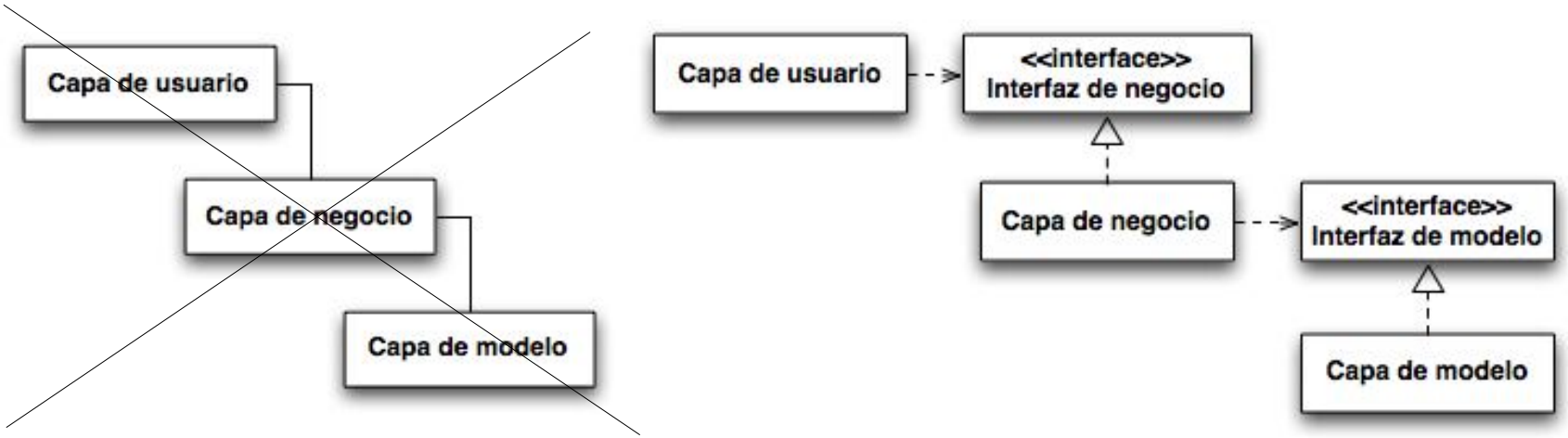
A. High level modules should not depend upon low level modules. Both should depend upon abstractions.
B. Abstractions should not depend upon details. Details should depend upon abstractions.

This is the principle that is at the very heart of framework design.

The dependency inversion principle

- **The dependency inversion principle**

According to 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”



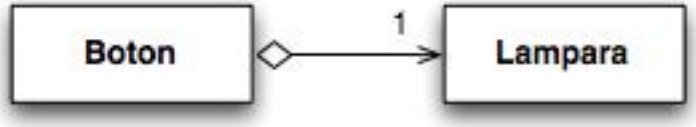
Object Solutions, Grady Booch, Addison Wesley, 1996, p54

The dependency inversion principle

- **The dependency inversion principle**

Dependency Inversion can be applied wherever one class sends a message to another.

For example, consider the case of the button object and the lamp object.




```
class Lampara {
  public void encender() {}
  public void apagar() {}
}
```

(It will not be possible to reuse the Boton class to control a Motor object)

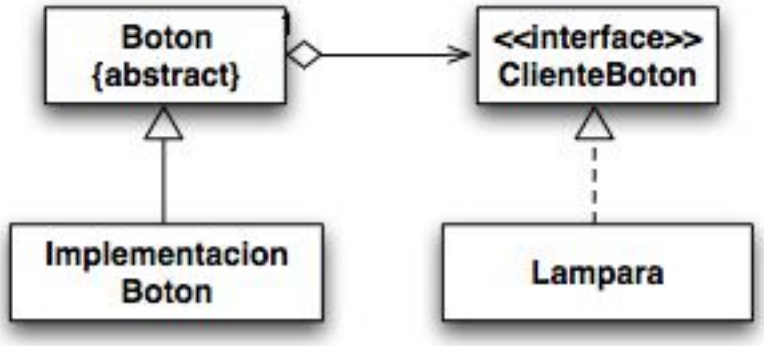
```
class Boton {
  public Boton(Lampara l)
    { lamp = l;}
  public void detectar() {
    boolean botonOn = getEstado();
    if (botonOn)
      lamp.encender();
    else
      lamp.apagar();
  }
  private Lampara lamp;
}
```

- **The dependency inversion principle**

To conform to the principle of dependency inversion, we must isolate an abstraction from the details of the problem.

In the Button/Lamp example, the underlying abstraction is to detect an on/off gesture from a user and relay that gesture to a target object.

What mechanism is used to detect the user gesture? Irrelevant! What is the target object? Irrelevant!



Homework: write Java code for this design which proceeds in a similar way to the code in the previous slide.

The dependency inversion principle

- **Conclusions**

The dependency inversion principle implies the separation of interface and implementation in different modules.

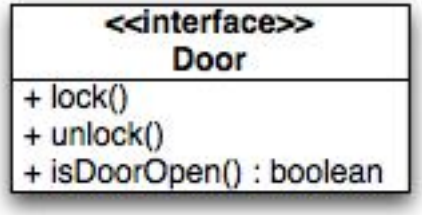
Corollary: Abstract classes should not depend upon concrete classes. Concrete classes should depend upon abstract classes.

The Interface segregation principle

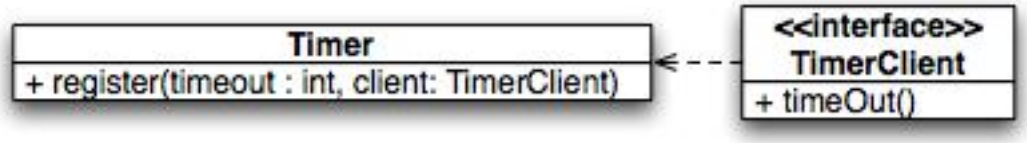
- **'Fat' interfaces**
  - Classes that have “fat” interfaces are classes whose interfaces are not cohesive.
  - In other words, the interfaces of the class can be broken up into groups of methods. Each group serves a different set of clients.
  - The Interface Segregation Principle acknowledges that there are objects that require non-cohesive interfaces; however it suggests that clients should not know about them as a single class.
  - Instead, clients should know about different abstract base classes or interfaces that have cohesive interfaces.

The Interface segregation principle

- **Interface pollution**
  - Consider a security system with Door objects that can be locked and unlocked, and which know whether they are open or closed:

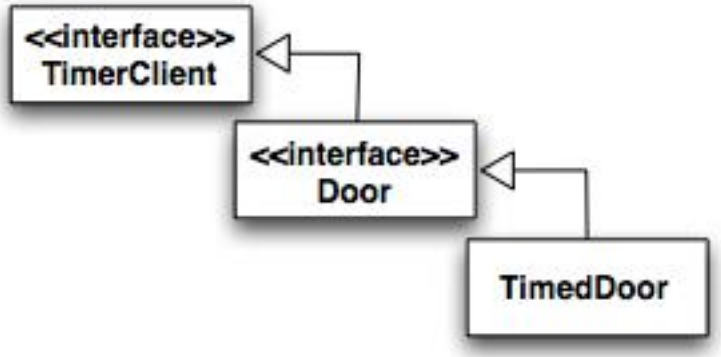


  - Now consider that the implementation TimedDoor needs to sound an alarm when the door has been left open for too long. In order to do this the TimedDoor object communicates with another object called a Timer. When an object wishes to be informed about a timeout, it calls the Register function of the Timer. The arguments of this function are the time of the timeout, and a pointer to a TimerClient object whose TimeOut function will be called when the timeout expires.



The Interface segregation principle

- **Interface pollution**
  - Common solution:



  - Problems: not all varieties of Door need timing. If timing-free derivatives of Door are created, those derivatives will have to provide nil implementations for the TimeOut method.
  - This is the syndrome of interface pollution. The interface of Door has been polluted with an interface that it does not require. It has been forced to incorporate this interface (and get 'fat') solely for the benefit of one of its subclasses.

The Interface segregation principle

- **The interface segregation principle**

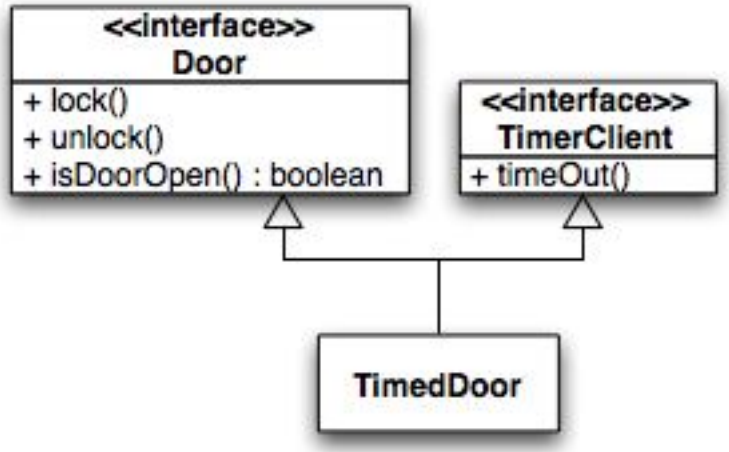
> Clients should not be forced to depend upon interfaces that they do not use.

How can we separate the interfaces when they must remain together?

- The answer to this lies in the fact that clients of an object do not need to access it through the interface of the object. Rather, they can access it through
  - delegation, or
  - multiple interface inheritance

The Interface segregation principle

- **Separation through multiple interface inheritance**



- TimedDoor inherits from both Door and TimerClient. Although clients of both base classes can make use of TimedDoor, neither actually depend upon the TimedDoor class. Thus, they use the same object through separate interfaces.

The Interface segregation principle

- **Conclusions**
  - We have discussed the disadvantages of “fat interfaces”; i.e. interfaces that are not specific to a single client. Fat interfaces lead to inadvertent couplings between clients that ought otherwise to be isolated.
  - Fat interfaces can be segregated into abstract base classes or interfaces that break the unwanted coupling between clients.

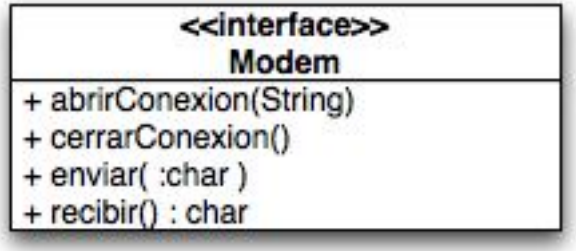
The single responsibility principle

There should never be more than one reason for a class to change.

- When the requirements change, that change will be manifest through a change in responsibility amongst the classes.
- If a class assumes more than one responsibility, then there will be more than one reason for it to change.
- If a class has more then one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class' ability to meet the others (fragility).

The single responsibility principle

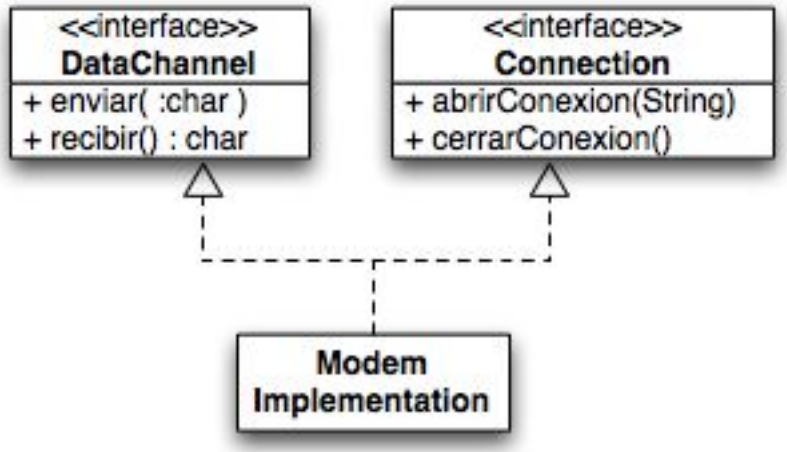
- Example: interface Modem



- This interface looks perfectly reasonable. However, there are two responsibilities being shown here.
  - 1. Connection management (abrirConexion/cerrarConexion)
  - 2. Data communication (enviar/recibir)
- The two sets of functions have almost nothing in common. They'll certainly change for different reasons.
- Moreover, they will be called from completely different parts of the applications that use them. Those different parts will change for different reasons as well.

The single responsibility principle

- Example: Interface Modem
  - This design separates the two responsibilities into two separate interfaces:



  - The two responsibilities are recoupled into a single ModemImplementation class.
  - However, by separating their interfaces we have decoupled the concepts as far as the rest of the application is concerned. Nobody need depend upon this class. Nobody except main needs to know that it exists.

The single responsibility principle

- Conclusions
  - The single responsibility principle is is one of the simplest of the principles, and one of the hardest to get right.
  - Conjoining responsibilities is something that we do naturally.
  - Finding and separating those responsibilities from one another is much of what software design is really about.

Principles of object-oriented design

- All the previous design principles have to do with class design.
- **SOLID** is used as a mnemonic acronym for these five basic principles:
  - **S**ingle Responsibility Principle (SRP)
  - **O**pen-Closed Principle (OCP)
  - **L**iskov Substitution Principle (LSP)
  - **I**nterface Segregation Principle (ISP)
  - **D**ependency Inversion Principle (DIP)

- There exist additional principles of OO design dealing with higher-level structures, such as cohesion and coupling at the package level.

A look at design patterns

- When faced with the task of solving a new problem, most people will consider first previous problems they have encountered that seem to have characteristics in common with the new assignment.
- **Design patterns:**
  - A design pattern is a solution to a design problem.
  - Design patterns must have the following features:
    - They have proved to be useful because they have been succesfully applied to similar problems in the past.
    - They are reusable in the sense that they can be applied to similar problems under different circumstances.
- A design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.

A look at design patterns

- Design patterns can speed up the development process by providing tested, proven development paradigms, avoiding the need to “reinvent the wheel”.
- Most design patterns allow you to achieve higher cohesion and looser coupling for more flexible, reusable applications.
- A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

A look at design patterns

- A simple example: the **adapter pattern**
  - The adapter pattern (often referred to as the wrapper pattern or simply a wrapper) is a design pattern that translates one interface for a class (server) into a compatible interface.
  - An adapter allows classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients while using the original interface.
  - The adapter translates calls to its interface into calls to the original interface (delegation), and the amount of code necessary to do this is typically small.
  - The adapter is also responsible for transforming data into appropriate forms.

A look at design patterns

- A simple example: the **adapter pattern**

```
class MiColeccion implements Collection {

  public boolean isEmpty ()
    { return data.cuantos() == 0; }
  public int size ()
    { return data.cuantos(); }
  public void add (Object nuevo)
    { data.colocarAlFinal(nuevo); }
  public boolean contains (Object test)
    { return data.busca(test) != null; }
  // … etc

  private Vectorcito data = new Vectorcito();
}
```

- 'Vectorcito' is a container class which does not implement the interface Collection
- Adapters are often used to connect software components from different vendors.

Bibliography

- Robert C. Martin. ***Principles of object oriented design***
  http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign

- T.C. Lethbridge, R. Laganiere. ***Object-oriented software engineering : practical software development using UML and Java***
  - *Section 9.1 (the process of design)*

- T. Budd. ***An Introduction to Object-oriented Programming, 3rd ed.***
  - *Ch. 23 (coupling and cohesion)*
  - *Ch. 24 (design patterns)*