Ingegneria del Software T

Design Principles

Design quality

- **Design quality** is an elusive concept
- Quality depends on specific **organisational priorities**
- A ‘good’ design may be
  - the most reliable,
  - the most efficient,
  - the most maintainable,
  - the cheapest, …
- The attributes discussed here are concerned with the **maintainability of the design**

What Makes a Design “Bad”?

- ✓ **Misdirection**: fails to meet requirements
- ✓ **Software Rigidity**: a single change affects many other parts of the system
- ✓ **Software Fragility**: a single change affects unexpected parts of the system
- ✓ **Software Immobility**: it is hard to reuse in another application
- ✓ **Viscosity**: it is hard to do the right thing, but easy to do the wrong thing

Software Rigidity

- The tendency for software **to be difficult to change**
- **Symptom**: every change causes a cascade of subsequent changes in dependent modules
- **Effect**: managers fear to allow developers to fix non-critical problems – they don't know when the developers will be finished

Software Fragility

- The tendency of the software **to break in many places** every time it is changed ▶ changes tend to cause **unexpected behavior** in other parts of the system (often in areas that have no conceptual relationship with the area that was changed)
- **Symptom**: every fix makes it worse, introducing more problems than are solved ▶ such software is impossible to maintain
- **Effect**: every time managers authorize a fix, they fear that the software will break in some unexpected way

Software Immobility

- The **inability to reuse software** from other projects or from parts of the same project
- **Symptom**: a developer discovers that he needs a module that is similar to one that another developer wrote. But the module in question has too much baggage that it depends upon. After much work, the developer discovers that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate
- **Effect**: and so the software is simply rewritten instead of reused

Viscosity

- Developers usually find more than one way to make a change: some preserve the design, others do not (i.e. they are hacks)
- The tendency to **encourage software changes that are hacks** rather than software changes that preserve original design intent
  - **Viscosity of design**: the design preserving methods are harder to employ than the hacks
  - **Viscosity of environment**: the development environment is slow and inefficient (compile times are very long, source code control system requires hours to check in just a few files, …)
- **Symptom**: it is easy to do the wrong thing, but hard to do the right thing
- **Effect**: the software maintainability degenerates due to hacks, workarounds, shortcuts, temporary fixes, …

Why bad design results?

- Obvious reasons:
  - lack of design skills/design practices
  - changing technologies
  - time/resource constraints
  - domain complexity, …
- Not so obvious:
  - **software rotting is a slow process** – even originally clean and elegant design may degenerate over the months/years
  - **requirements** often **change** in the way the original design or designer did not anticipate
  - unplanned and improper module **dependencies** creep in
    - ▶ dependencies go unmanaged

Requirement Changes

- We, as software engineers, know full well that requirements change
- Indeed, most of us realize that the requirements document is the **most volatile document** in the project
- If our designs are failing due to the constant rain of changing requirements**, it is our designs that are at fault**
- We must somehow find a way to make our designs resilient to such changes and protect them from rotting

Dependency Management

- Each of the four symptoms mentioned above is either directly, or indirectly caused by **improper dependencies between the modules** of the software
- It is the **dependency architecture** that is degrading, and with it the ability of the software to be maintained
- In order to forestall the degradation of the dependency architecture, the dependencies between modules in an application must be managed
- Object Oriented Design is replete with **principles** and **techniques** for managing module dependencies

Design Principles

- The **Single Responsibility Principle** (SRP)
- The **Dependency Inversion Principle** (DIP)
- The **Interface Segregation Principle** (ISP)
- The **Open/Closed Principle** (OCP)
- The **Liskov Substitution Principle** (LSP)

Premessa

Il principio zero

- Il principio zero è un principio di logica noto come **rasoio di Occam**:
  "*Entia non sunt multiplicanda praeter necessitatem*"
  ovvero: **non bisogna introdurre concetti che non siano strettamente necessari**
- È la forma "colta" di un principio pratico:
  "**Quello che non c'è non si rompe**" (H. Ford)
- Tra due soluzioni bisogna preferire quella
  - che introduce il minor numero di ipotesi e
  - che fa uso del minor numero di concetti

Premessa
Semplicità e semplicismo

- La **semplicità** è un fattore importantissimo
  - il software deve fare i conti con una notevole componente di complessità, generata dal contesto in cui deve essere utilizzato
  - quindi è estremamente importante **non aggiungere altra complessità** arbitraria
- Il problema è che
  - la semplicità richiede uno **sforzo non indifferente** (**è molto più facile essere complicati che semplici**)
  - in generale le soluzioni più semplici vengono in mente per ultime
- Bisogna fare poi molta attenzione ad essere semplici ma non semplicistici
  "**Keep it as simple as possible but not simpler**" (A. Einstein)

Premessa

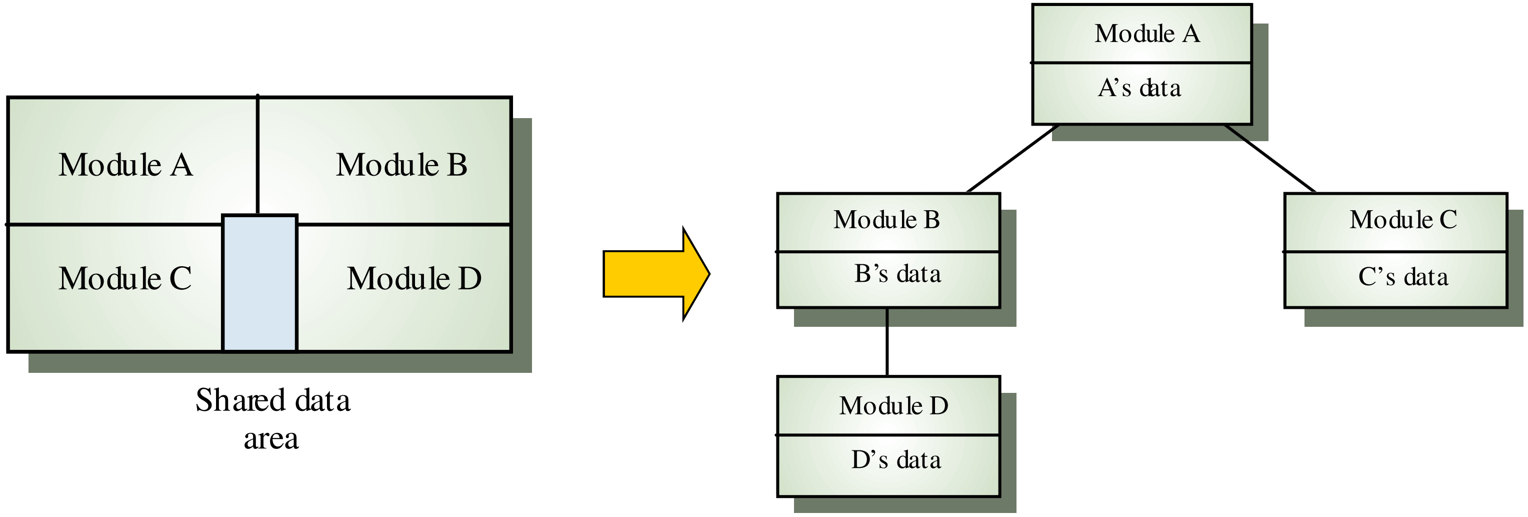
Divide et impera

- La **decomposizione** è una tecnica fondamentale per il controllo e la gestione della complessità
- Non esiste un solo modo per decomporre il sistema
  ▶ la **qualità della progettazione** dipende direttamente dalla **qualità delle scelte di decomposizione** adottate
- In questo contesto il **principio fondamentale** è: **minimizzare il grado di accoppiamento tra i moduli del sistema**
- Da questo principio è possibile ricavare diverse regole:
  - minimizzare la quantità di interazione fra i moduli
  - eliminare tutti i riferimenti circolari fra moduli
  - …

Premessa

Make all object data private

- Changes to public data are always at great 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

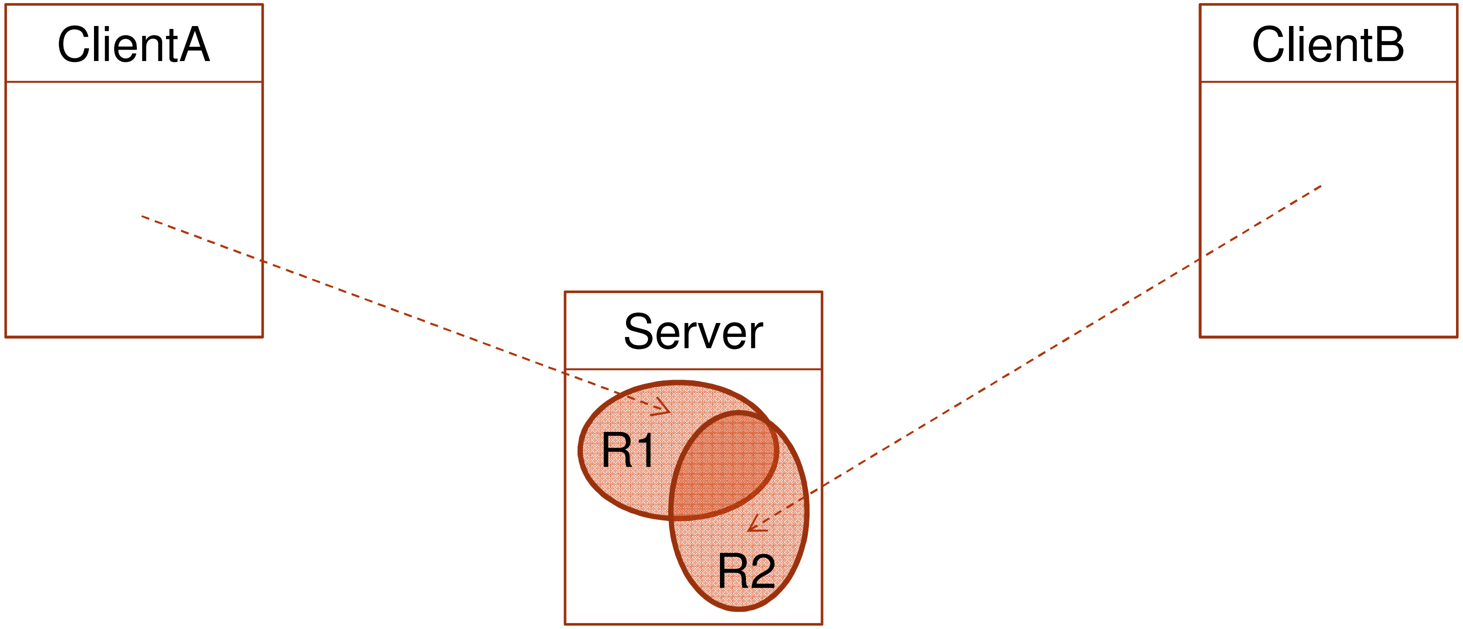


The Single Responsibility Principle

- ***There should never be more than one reason for a class to change*** (R. Martin)
- ***A class has a single responsibility: it does it all, does it well, and does it only*** (One Responsibility Rule – B. Meyer)

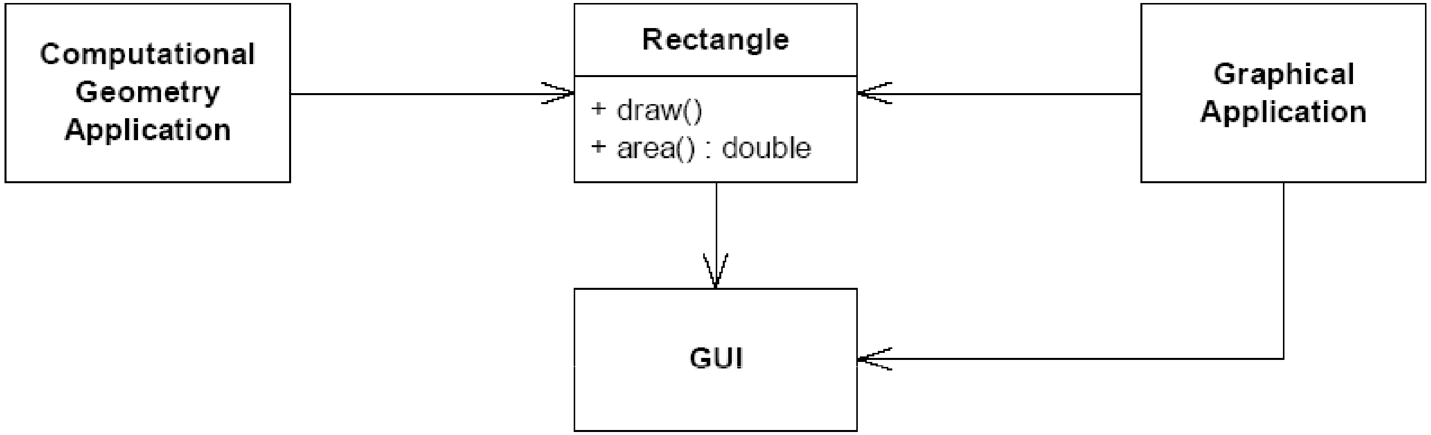
- If a class has more than one responsibility, then the responsibilities become coupled
- Changes to one responsibility may impair or inhibit the class' ability to meet the others
- This kind of coupling leads to **fragile designs** that break in unexpected ways when changed

The Single Responsibility Principle



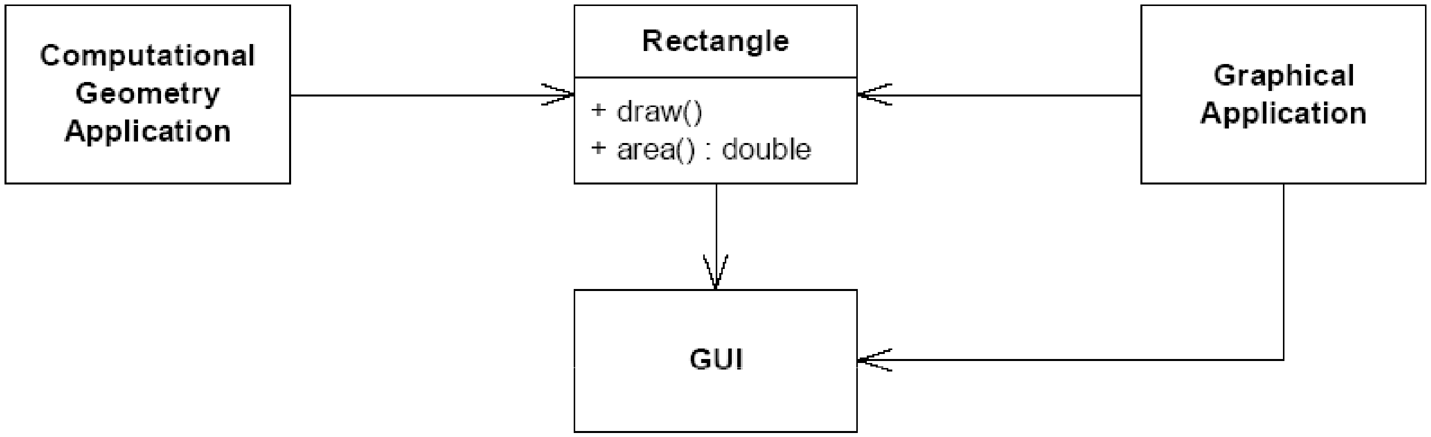
Una modifica di R1 può avere delle conseguenze anche su ClientB che NON utilizza direttamente R1

The Single Responsibility Principle Example



- One application does computational geometry
  - it uses Rectangle to help it with the mathematics of geometric shapes
  - it never draws the rectangle on the screen
- The other application is graphical in nature
  - it may also do some computational geometry, but
  - it definitely draws the rectangle on the screen

The Single Responsibility Principle Example

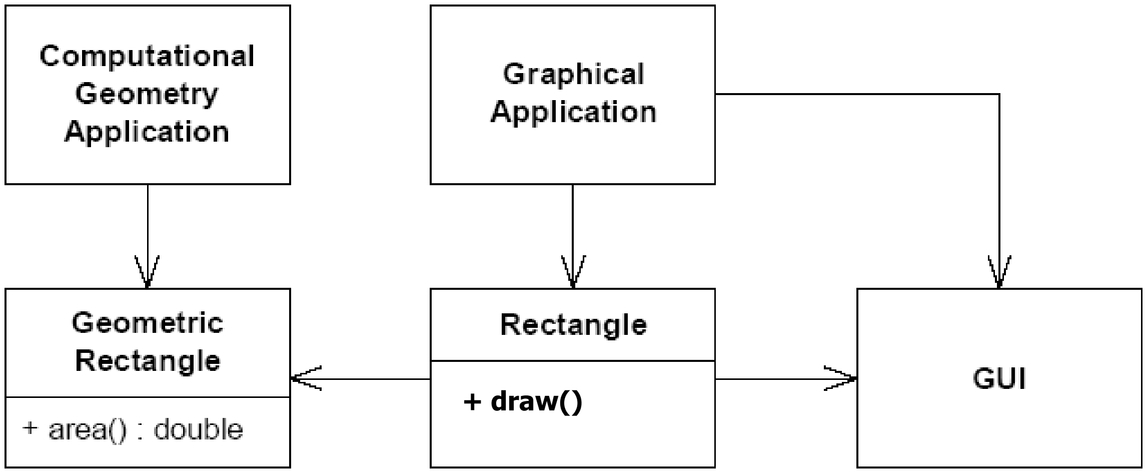


- The **`Rectangle`** class has **two responsibilities**
  - the **first responsibility** is to provide a mathematical model of the geometry of a rectangle
  - the **second responsibility** is to render the rectangle on a graphical user interface

The Single Responsibility Principle Example – **Refactoring**

- A better design is to **separate the two responsibilities into two completely different classes**

- **Extract Class**: create a new class and move the relevant fields and methods from the old class into the new class



The Single Responsibility Principle

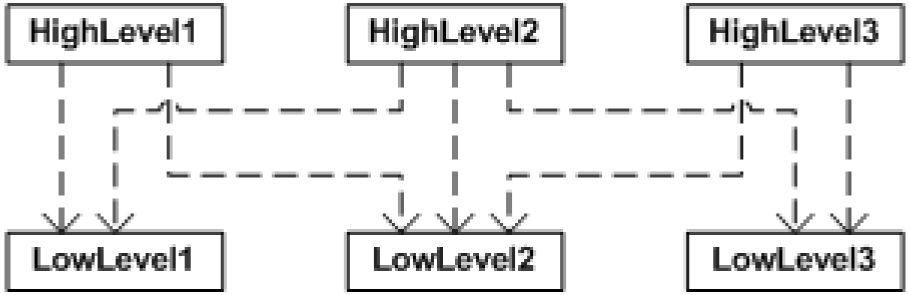
- **Class should have only one reason to change**
- Several responsibilities mean several reasons for changes ▶ more frequent changes
- The SRP is one of the **simplest** of the principle, and one of the **hardest** to get right
  - Conjoining responsibilities is something that we do naturally
  - Finding and separating those responsibilities from one another is much of what software design is really about

The Dependency Inversion Principle

- ***Depend upon abstractions***
  Do not depend upon concretions
  - Every dependency should target an interface, or an abstract class
  - No dependency should target a concrete class

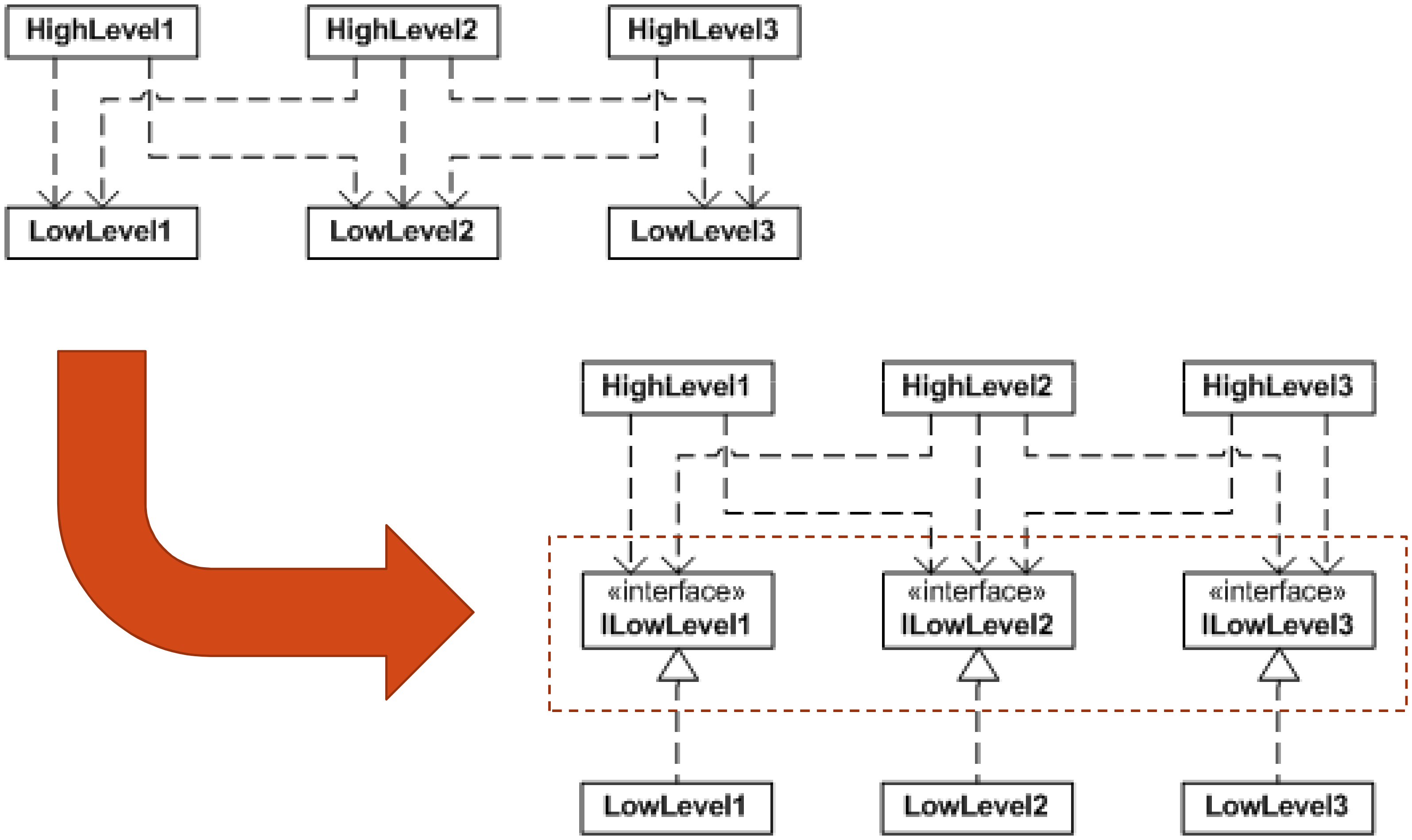
- **I moduli di alto livello (i clienti) non dovrebbero dipendere dai moduli di basso livello (i fornitori di servizi)**
  Entrambi dovrebbero dipendere da astrazioni
- **Le astrazioni non dovrebbero dipendere dai dettagli**
  I dettagli dovrebbero dipendere dalle astrazioni

The Dependency Inversion Principle



- I **moduli di basso livello** contengono la maggior parte del codice e della logica implementativa e quindi **sono i più soggetti a cambiamenti**
- Se **i moduli di alto livello** dipendono dai dettagli dei moduli di basso livello (sono accoppiati in modo troppo stretto), i cambiamenti si propagano e le conseguenze sono:
  - **Rigidità**: bisogna intervenire su un numero elevato di moduli
  - **Fragilità**: si introducono errori in altre parti del sistema
  - **Immobilità**: i moduli di alto livello non si possono riutilizzare perché non si riescono a separare da quelli di basso livello

The Dependency Inversion Principle



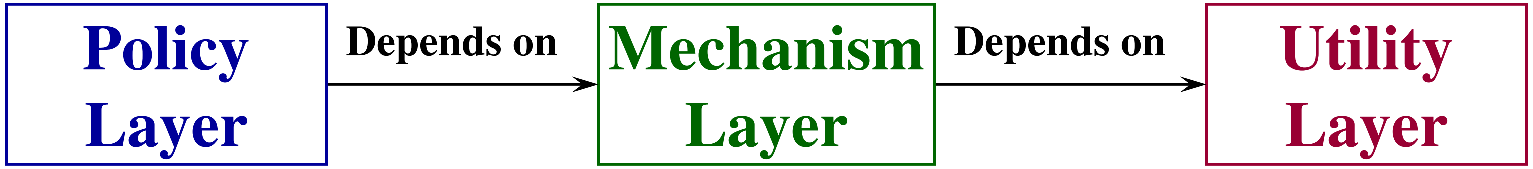
The Dependency Inversion Principle

- Questo principio funziona perché:
  - **le astrazioni** contengono pochissimo codice (in teoria nulla) e quindi **sono poco soggette a cambiamenti**
  - i **moduli non astratti sono soggetti a cambiamenti** ma questi cambiamenti sono sicuri perché nessuno dipende da questi moduli
- I dettagli del sistema sono stati isolati, separati da un **muro di astrazioni stabili**, e questo impedisce ai cambiamenti di propagarsi (**design for change**)
- Nel contempo i singoli moduli sono **maggiormente riusabili** perché sono disaccoppiati fra di loro (**design for reuse**)

The Dependency Inversion Principle
Dipendenze transitive

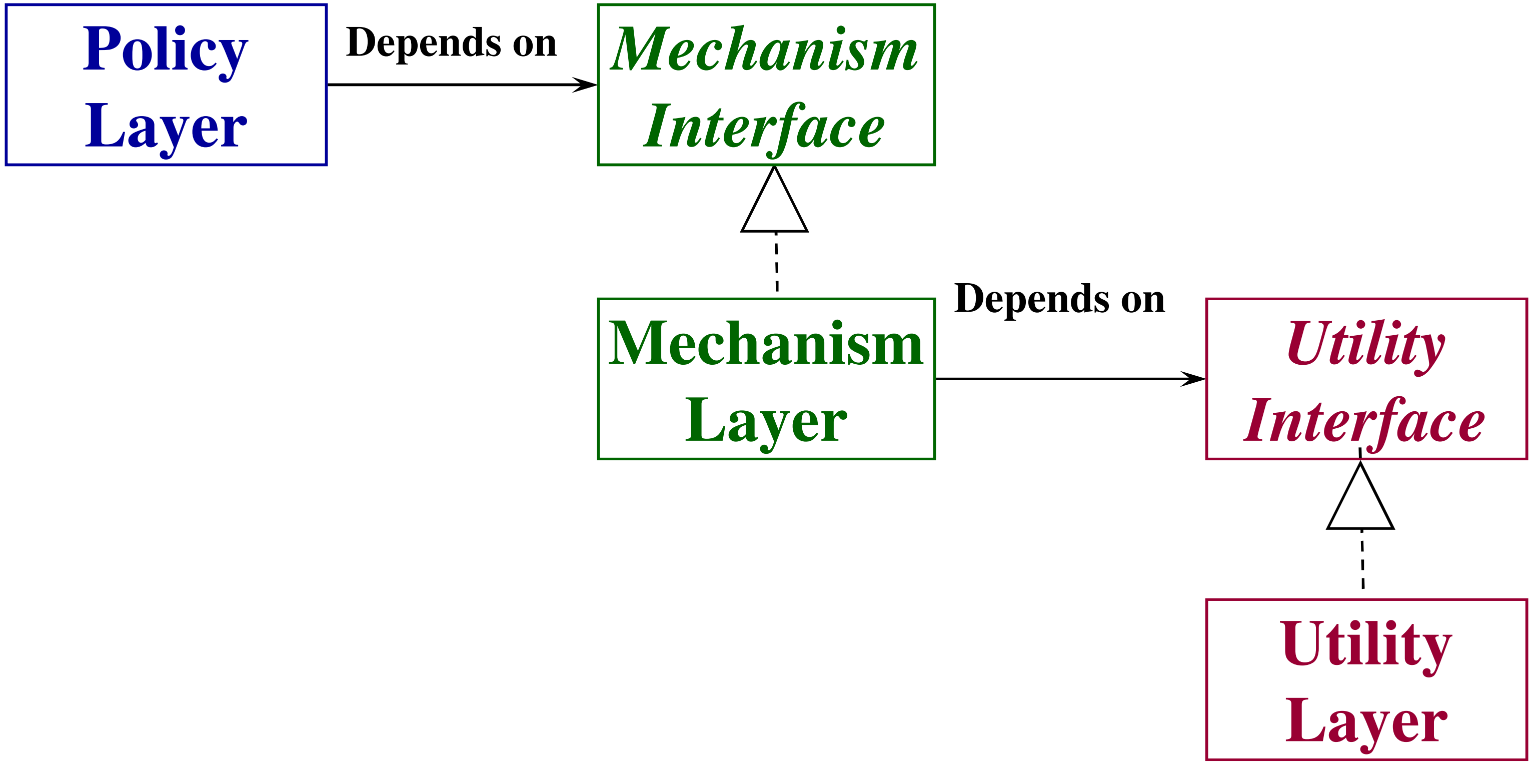
- “***...all well structured object-oriented architectures have clearly-defined layers, with each layer providing some coherent set of services though a well-defined and controlled interface***” (Grady Booch)

- I sistemi software dovrebbero essere stratificati, cioè organizzati a livelli
- Le **dipendenze transitive** devono essere eliminate



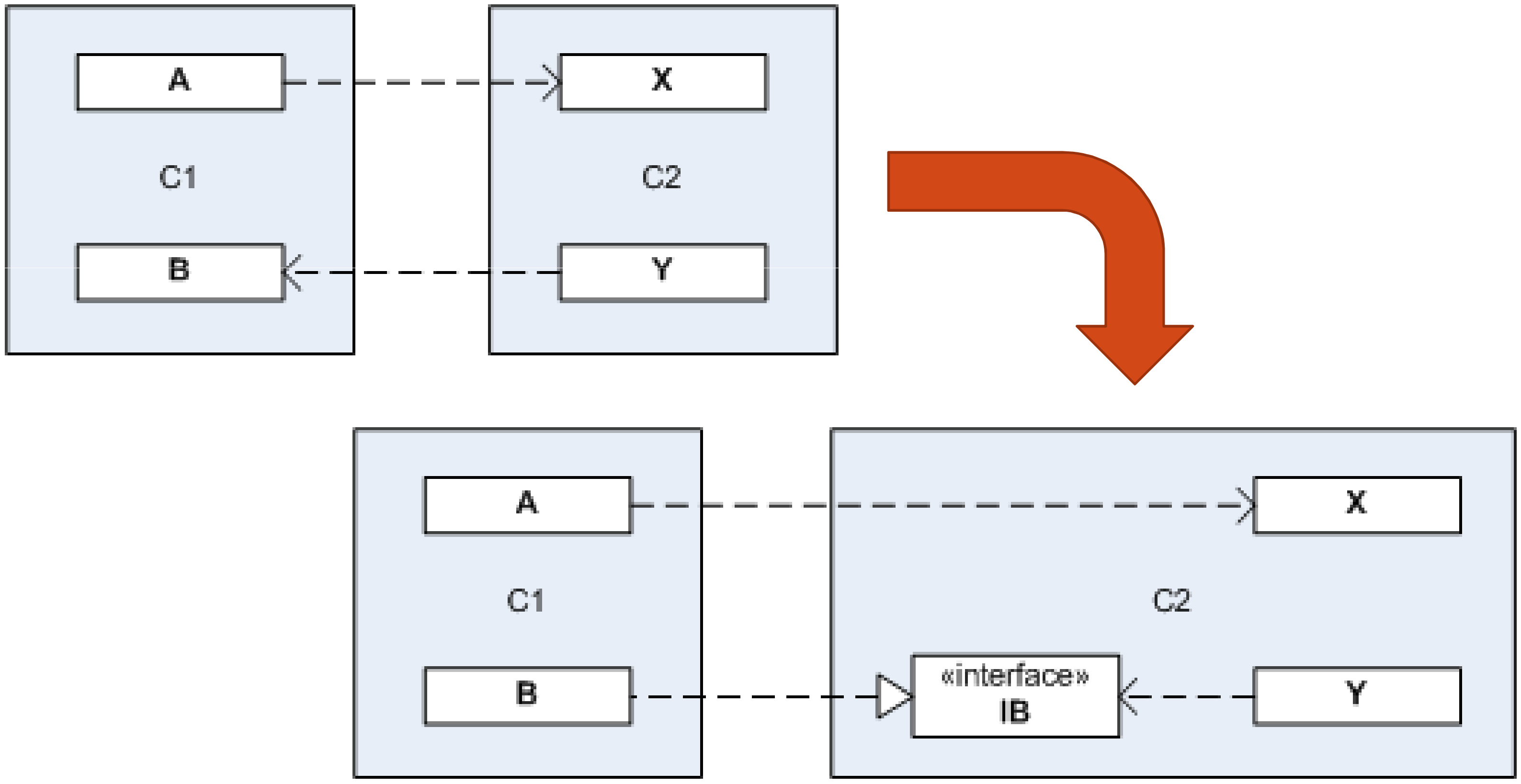
The Dependency Inversion Principle

Dipendenze transitive



The Dependency Inversion Principle
Dipendenze cicliche

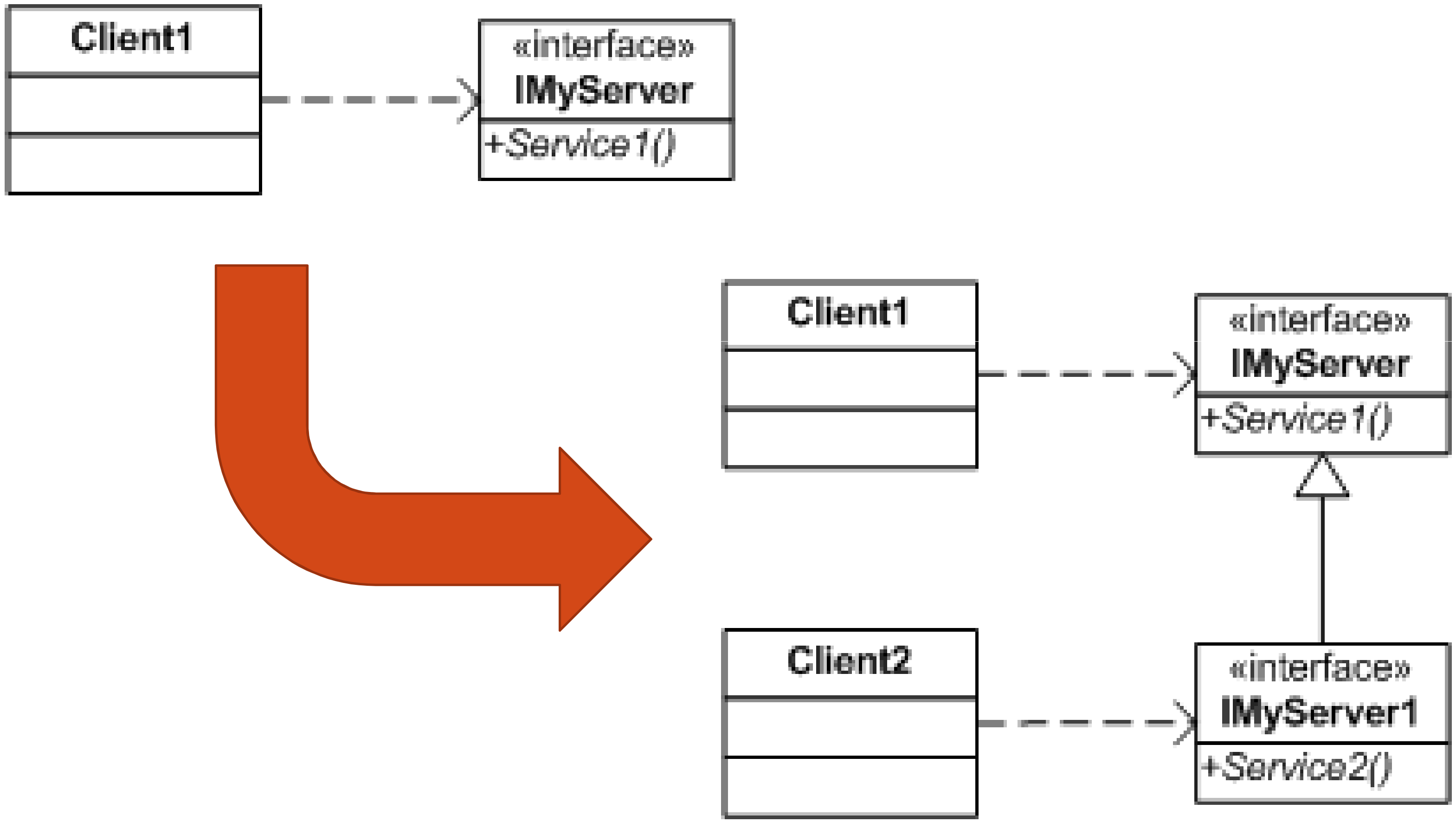
- Le **dipendenze cicliche** devono essere eliminate



The Dependency Inversion Principle
Definizione di un’interfaccia stabile

1. Consider all potential usage of a module
2. Define and document the public interface
   - minimize public behavior (the expected service to be performed) to minimize future change
   - define preconditions e postconditions
   - distribute widely and allow time for debate
3. Never change the published interface
   - Fixes or enhancements must not effect existing users

The Dependency Inversion Principle
Estensione di un'interfaccia in uso



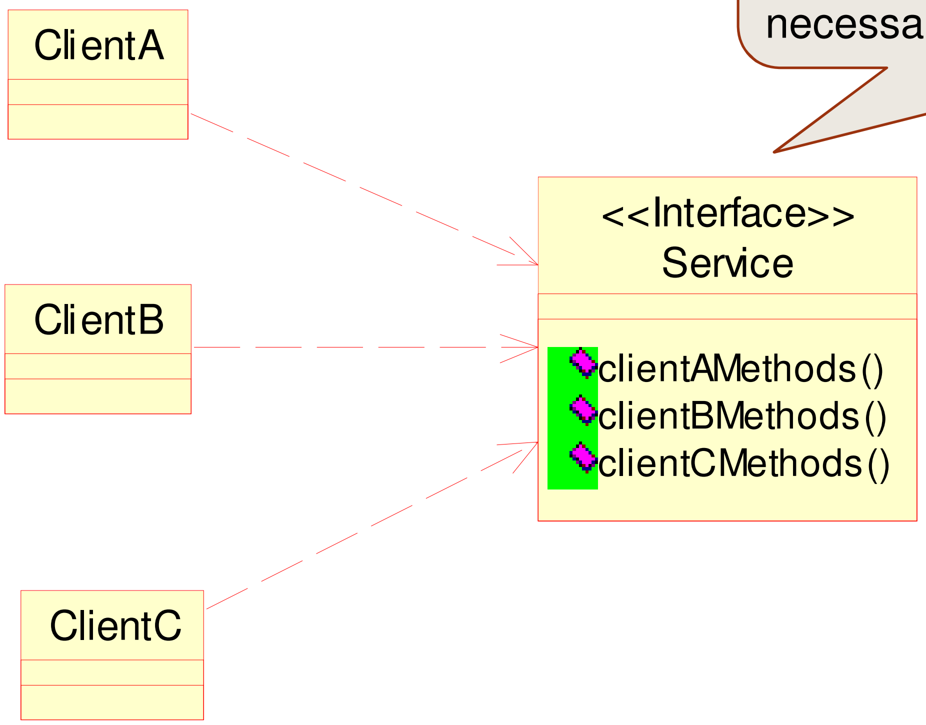
The Interface Segregation Principle

- ***Clients should not be forced to depend upon interfaces that they do not use***
- ***Many client specific interfaces are better than one general purpose interface***

The Interface Segregation Principle

Fat Interface

If a change is made to one of the methods that **ClientA** calls, **ClientB** and **ClientC** may be affected: it may be necessary to recompile and redeploy them



The Interface Segregation Principle

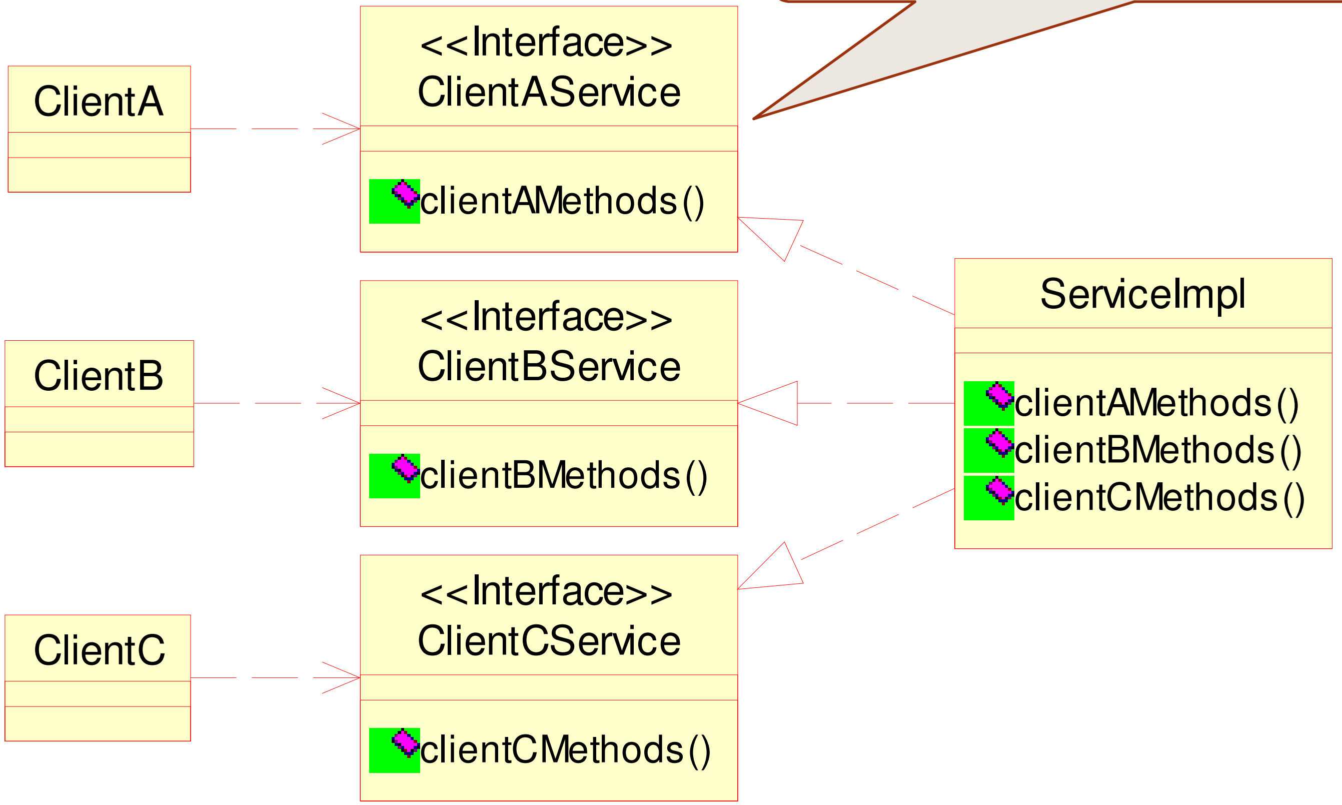
- I clienti non dovrebbero dipendere da servizi che non utilizzano
- Le fat interfaces creano una forma indiretta di accoppiamento (inutile) fra i clienti – se un cliente richiede l’aggiunta di una nuova funzionalità all’interfaccia, ogni altro cliente è costretto a cambiare anche se non è interessato alla nuova funzionalità
- Questo crea un’inutile sforzo di manutenzione e può rendere difficile trovare eventuali errori

↓

- Se i servizi di una classe possono essere suddivisi in gruppi e ogni gruppo viene utilizzato da un diverso insieme di clienti, creare interfacce specifiche per ogni tipo di cliente e implementare tutte le interfacce nella classe

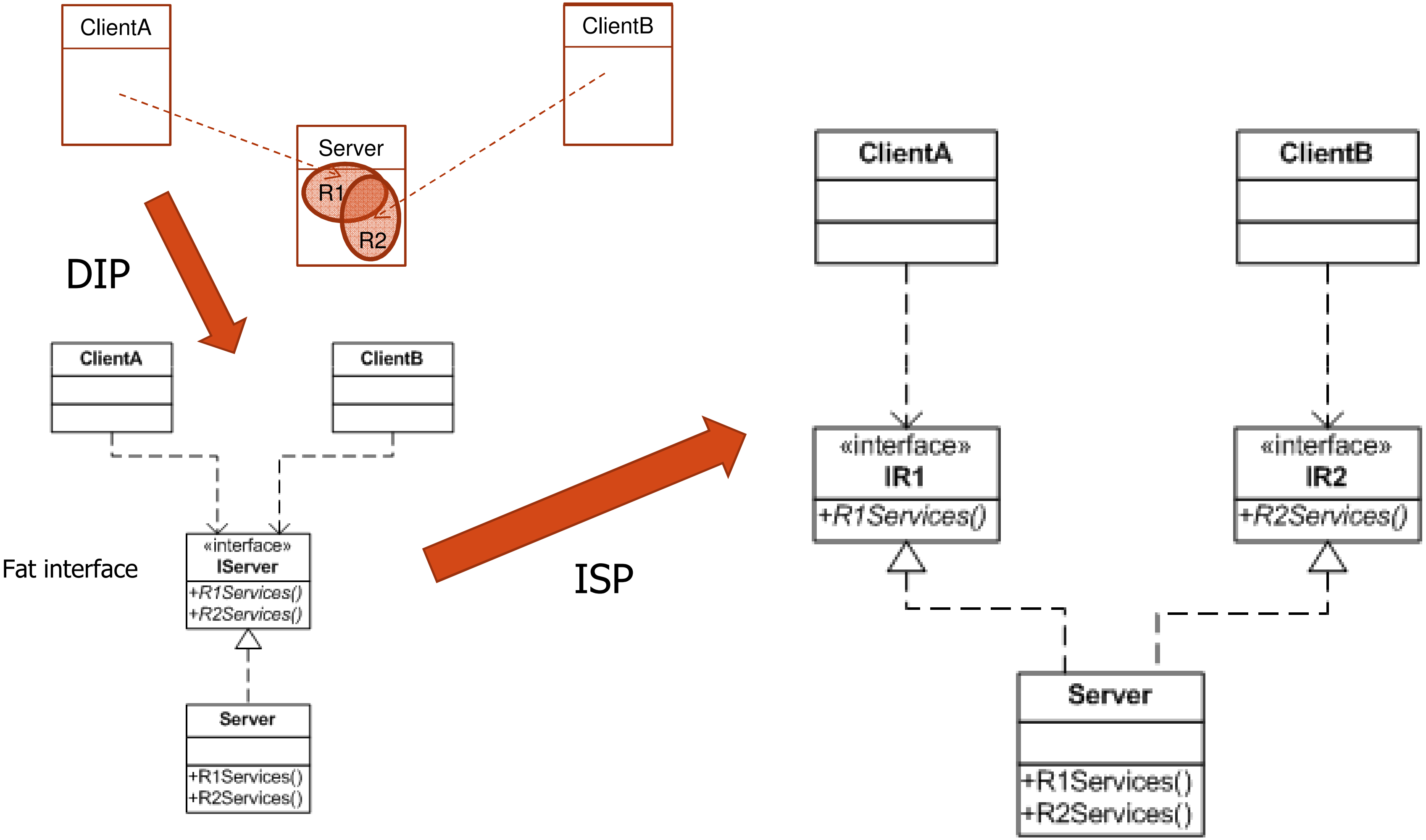
The Interface Segregation Principle

Segregated Interfaces



If the interface for ClientA needs to change, ClientB and ClientC will remain unaffected

The Interface Segregation Principle



The Open/Closed Principle

- **The most important principle** for **designing reusable entities**
- ***Software entities (classes, modules, functions, …) should be open for extension, but closed for modification***
- **Open**:
  - **May be extended** by adding new state or behavioral properties
- **Closed**:
  - Has a well-defined, published and stable interface which **may not be changed**

The Open/Closed Principle

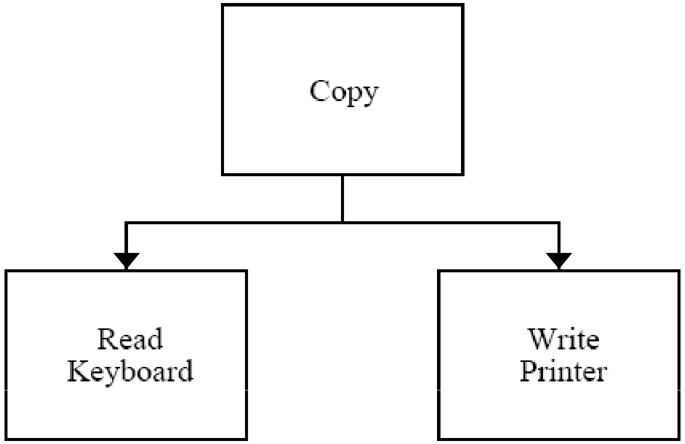
- We should write our modules so that
  - they **can be extended**,
  - **without** requiring them **to be modified**
- In other words, we want to be able
  - to change what the modules do,
  - without changing the source code of the modules
- Apparentemente si tratta di una **contraddizione**:
  come può un modulo immutabile esibire un comportamento che non sia fisso nel tempo?
- La risposta risiede **nell'astrazione**:
  è possibile creare astrazioni che rendono un modulo immutabile,
  ma rappresentano un gruppo illimitato di comportamenti

The Open/Closed Principle

- Il segreto sta nell'utilizzo di **interfacce** (o di **classi astratte**)
- A un'interfaccia **immutabile** possono corrispondere innumerevoli classi concrete che realizzano comportamenti diversi
- Un modulo che utilizza astrazioni
  - non dovrà mai essere modificato, dal momento che le astrazioni sono immutabili (**il modulo è chiuso per le modifiche**)
  - potrà cambiare comportamento, se si utilizzano nuove classi che implementano le astrazioni (**il modulo è aperto per le estensioni**)

The Open/Closed Principle Esempio 1

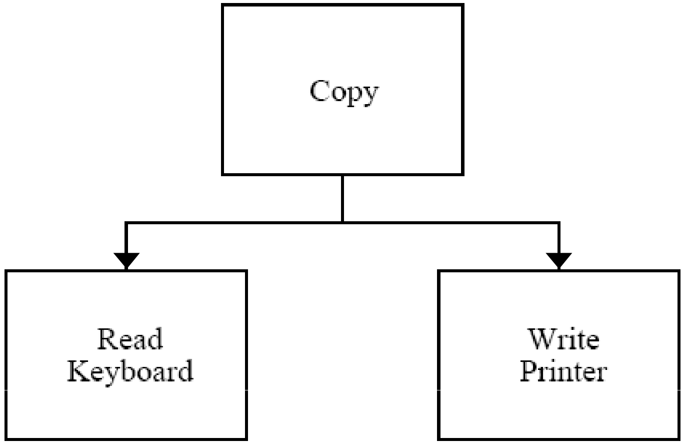
- Consider a simple program that is charged with the task of copying characters typed on a keyboard to a printer
- Assume, furthermore, that the implementation platform does not have an operating system that supports device independence



```
void Copy()
{
int c;
while ((c = ReadKeyboard()) != EOF)
  WritePrinter(c);
}
```

The Open/Closed Principle Esempio 1

- **The two low level modules are reusable**: they can be used in many other programs to gain access to the keyboard and the printer – this is the same kind of reusability that we gain from subroutine libraries
- **The "Copy" module is not reusable** in any context which does not involve a keyboard or a printer
- This is a shame since the intelligence of the system is maintained in this module – it is the "Copy" module that encapsulates a very interesting policy that we would like to reuse



```
void Copy()
{
int c;
while ((c = ReadKeyboard()) != EOF)
  WritePrinter(c);
}
```

The Open/Closed Principle
Esempio 1

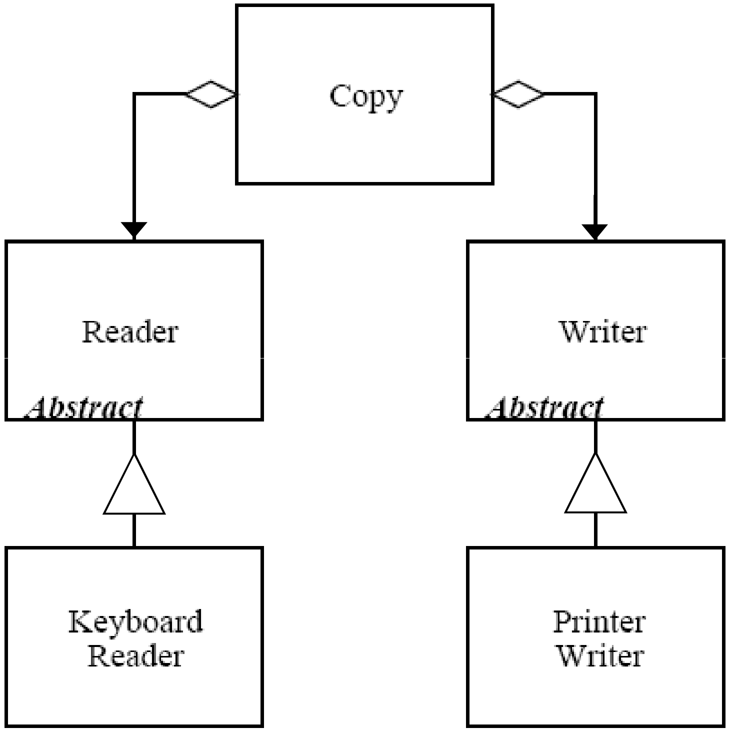
- Consider a new program that copies keyboard characters to a disk file
- We could modify the “Copy” module to give it the new desired functionality
- As time goes on, and more and more devices must participate in the copy program, the “Copy” module will be littered with if/else statements and **will be dependent upon many lower level modules**
  ▶ it will eventually become **rigid** and **fragile**

```
enum OutputDevice
{
  Printer,
  Disk
};

void Copy(OutputDevice dev)
{
int c;
while ((c = ReadKeyboard())
   != EOF)
   {
   if (dev == Printer)
     WritePrinter(c);
   else
     WriteDisk(c);
   }
}
```

The Open/Closed Principle
Esempio 1

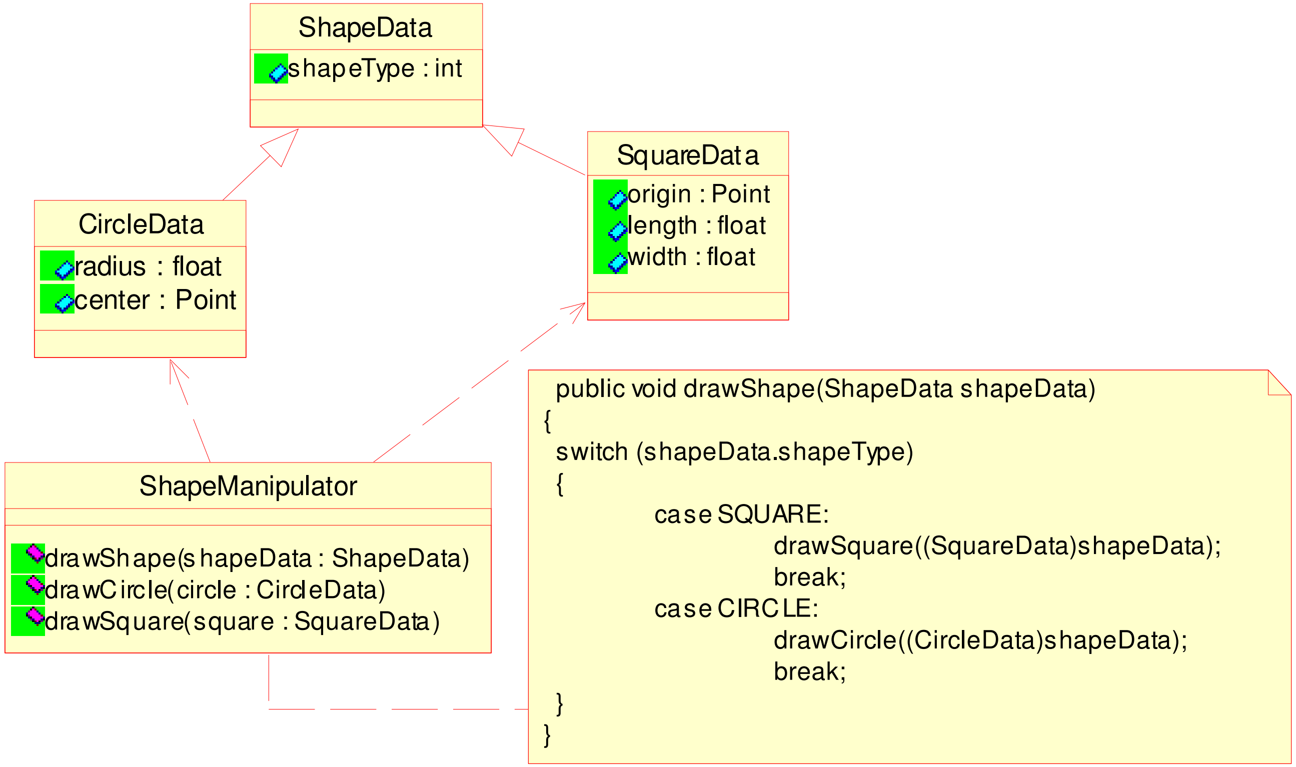
- One way to characterize the problem above is to notice that the module containing the high level policy (Copy) is dependent upon the low level detailed modules that it controls (WritePrinter and ReadKeyboard)
- If we could find a way to make the Copy module independent of the details that it controls, then
  - we could **reuse** it freely
  - we could produce other programs which used this module to **copy characters from any input device to any output device**

The Open/Closed Principle
Esempio 1



```
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);
}
```

The Open/Closed Principle
Esempio 2

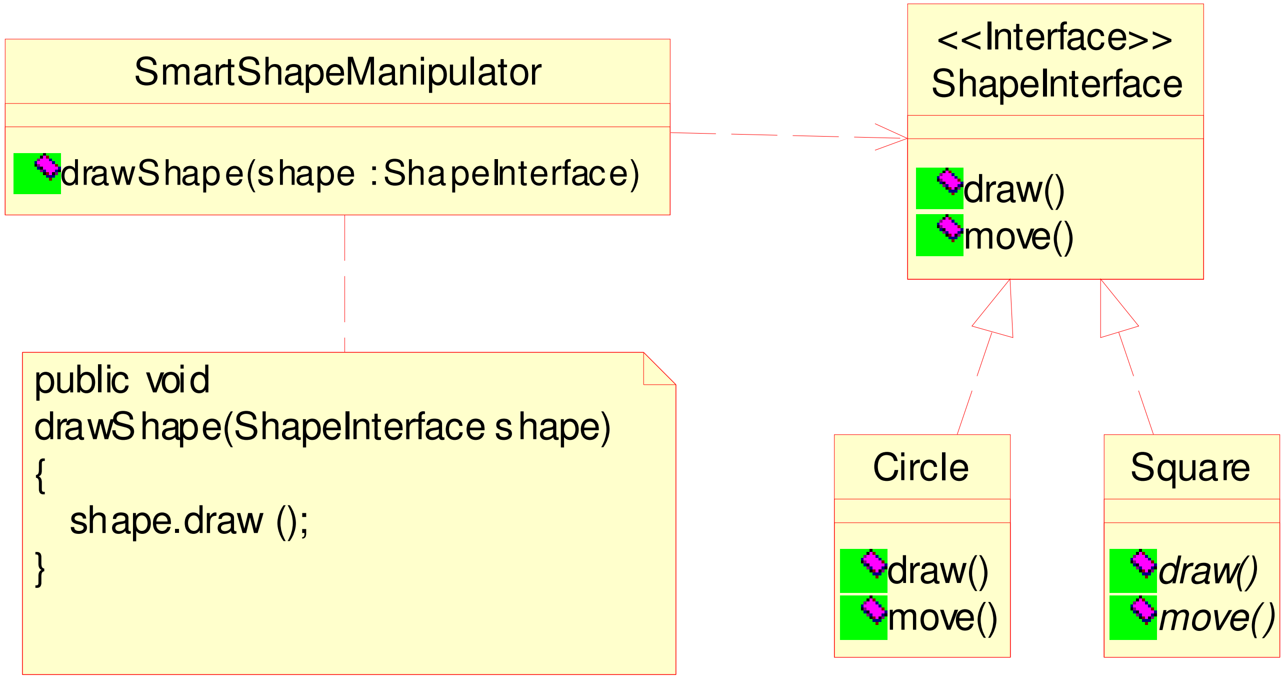


```
 public void drawShape(ShapeData shapeData)
{
 switch (shapeData.shapeType)
 {
          case SQUARE:
                    drawSquare((SquareData)shapeData);
                    break;
          case CIRCLE:
                    drawCircle((CircleData)shapeData);
                    break;
 }
}
```

The Open/Closed Principle Esempio 2

- **If I need to create a new** shape, such as a **`Triangle`**, **I must modify** **`drawShape`**
- In a complex application the **`switch/case`** statement above is repeated over and over again for every kind of operation that can be performed on a shape
- Worse, every module that contains such a **`switch/case`** statement **retains a dependency upon every possible shape** that can be drawn, thus, whenever one of the shapes is modified in any way, the modules all need recompilation, and possibly modification

The Open/Closed Principle
Esempio 2



SmartShapeManipulator

drawShape(shape :ShapeInterface)

```
public void
drawShape(ShapeInterface shape)
{
   shape.draw ();
}
```

<<Interface>>
ShapeInterface

draw()
move()

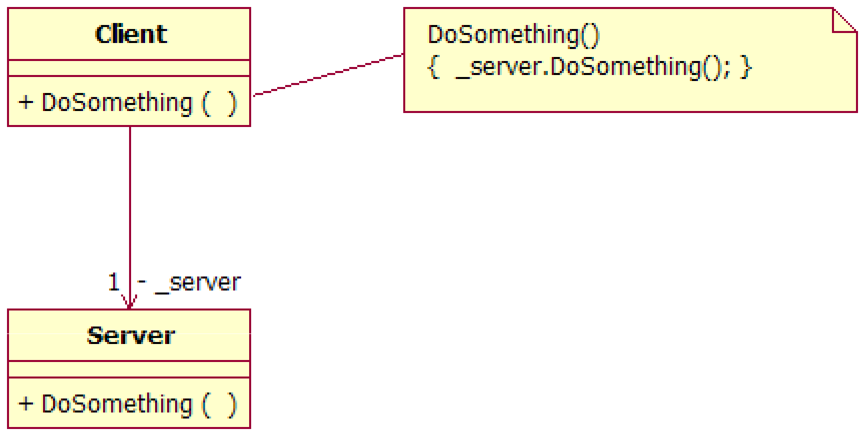
Circle

draw()
move()

Square

draw()
move()

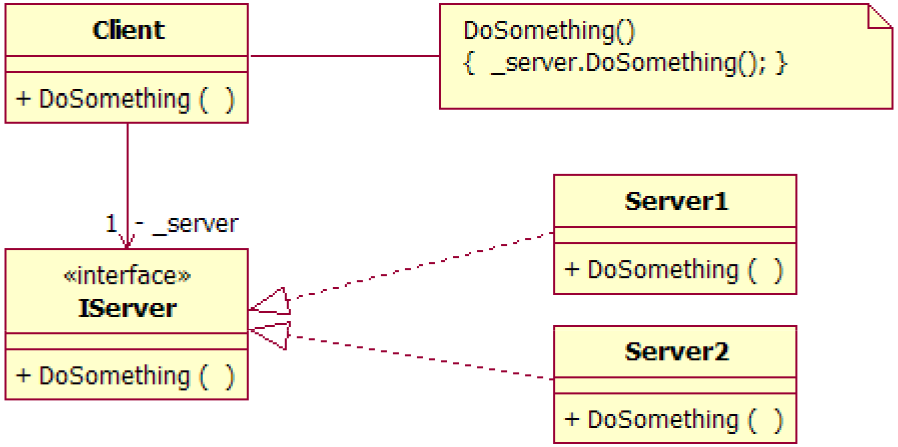
The Open/Closed Principle
Esempio 3



```
DoSomething()
{  _server.DoSomething(); }
```

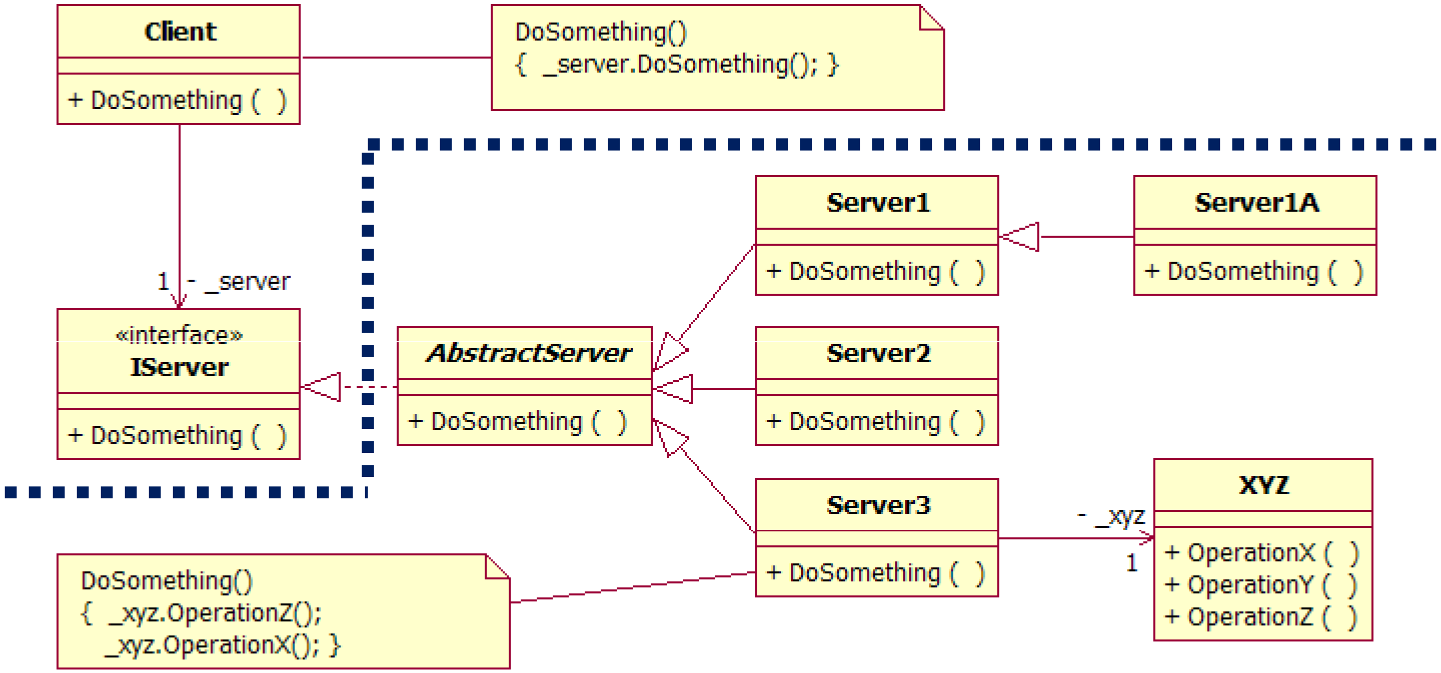
- Supponiamo di dover utilizzare un nuovo tipo di server!

The Open/Closed Principle
Esempio 3

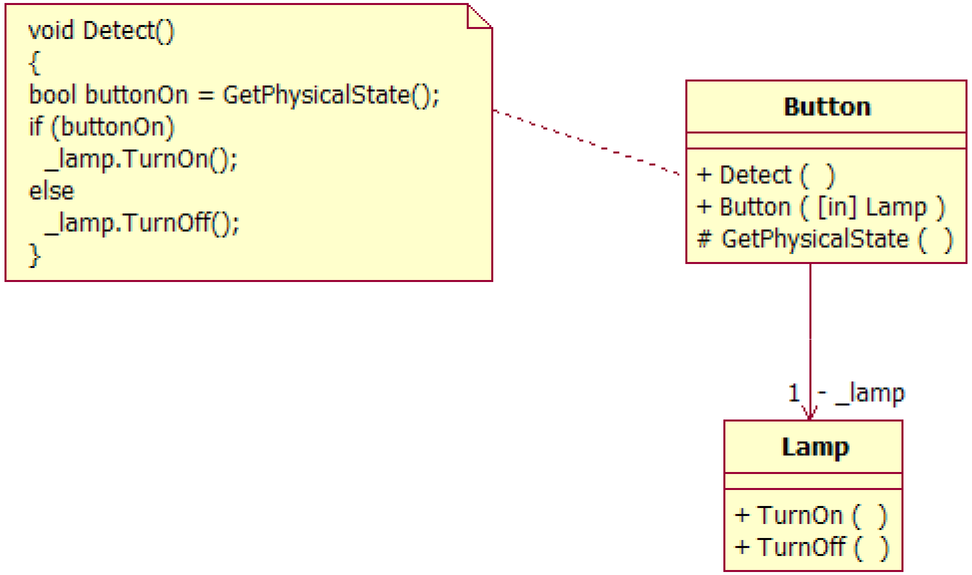


- **`Client`** is closed to changes in implementation of **`IServer`**
- **`Client`** is open for extension through new **`IServer`** implementations
- Without **`IServer`** the **`Client`** is open to changes in **`Server`**

The Open/Closed Principle
Esempio 3



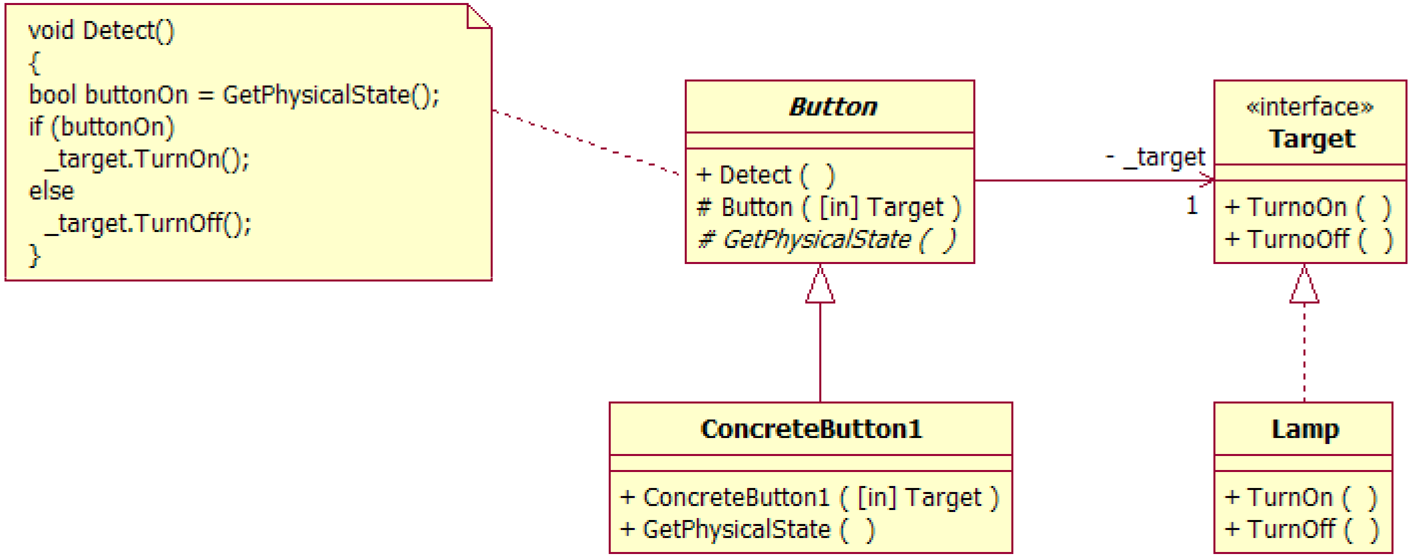
The Open/Closed Principle
Esempio 4



```
void Detect()
{
bool buttonOn = GetPhysicalState();
if (buttonOn)
  _lamp.TurnOn();
else
  _lamp.TurnOff();
}
```

- E se volessimo accendere un motore?

The Open/Closed Principle
Esempio 4



```
void Detect()
{
bool buttonOn = GetPhysicalState();
if (buttonOn)
  _target.TurnOn();
else
  _target.TurnOff();
}
```

The Open/Closed Principle

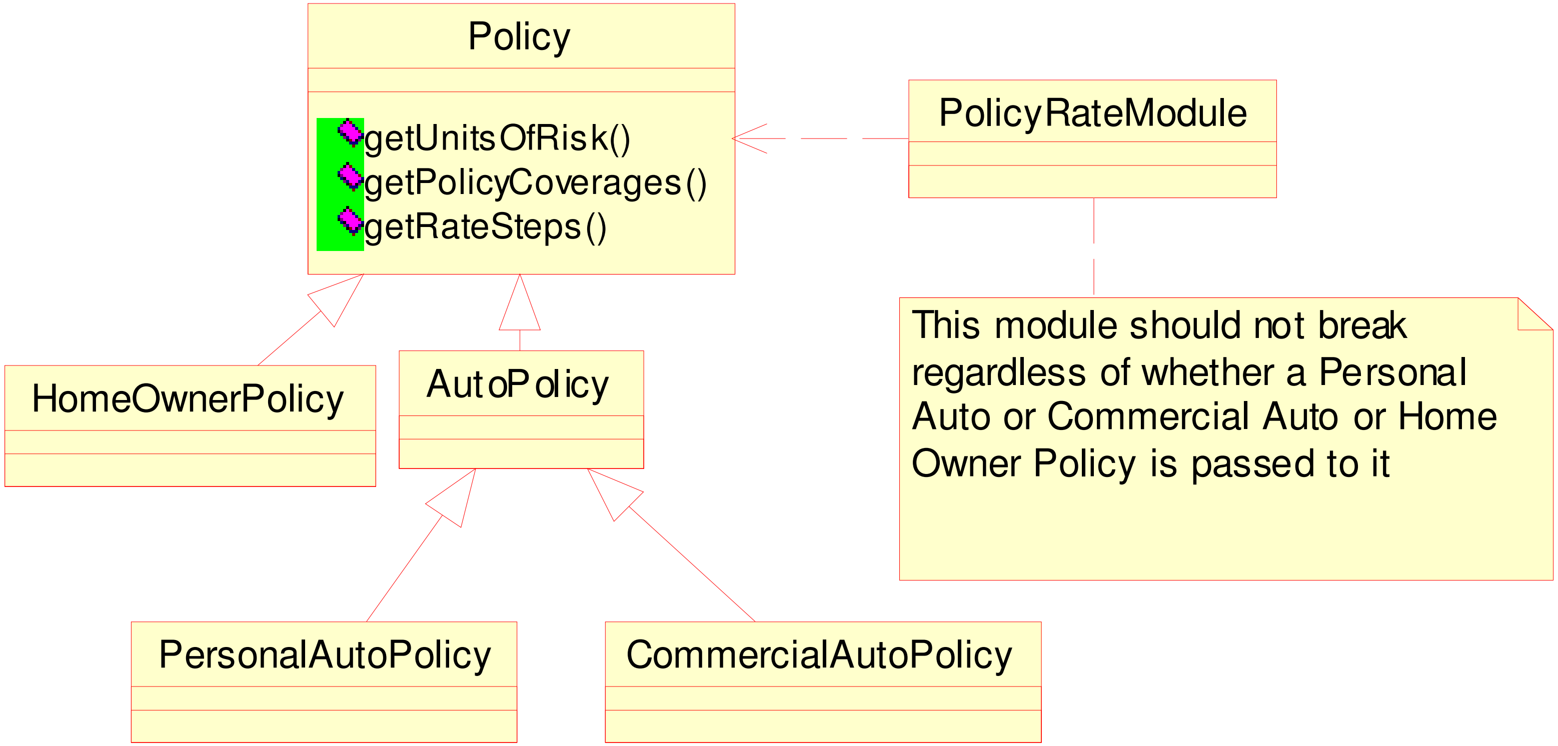
- When the majority of modules in an application conform to the OCP, then
  - new features can be added to the application
    - by **adding new code**
    - rather than by **changing working code**
  - the working code is not exposed to breakage
- Even if the OCP cannot be fully achieved, even partial OCP compliance can make dramatic improvements in the structure of an application

The Liskov Substitution Principle

- ***Subclasses should be substitutable for their base classes*** (Barbara Liskov)
- ***All derived classes must honor the contracts of their base classes*** (Design by Contract – Bertrand Meyer)

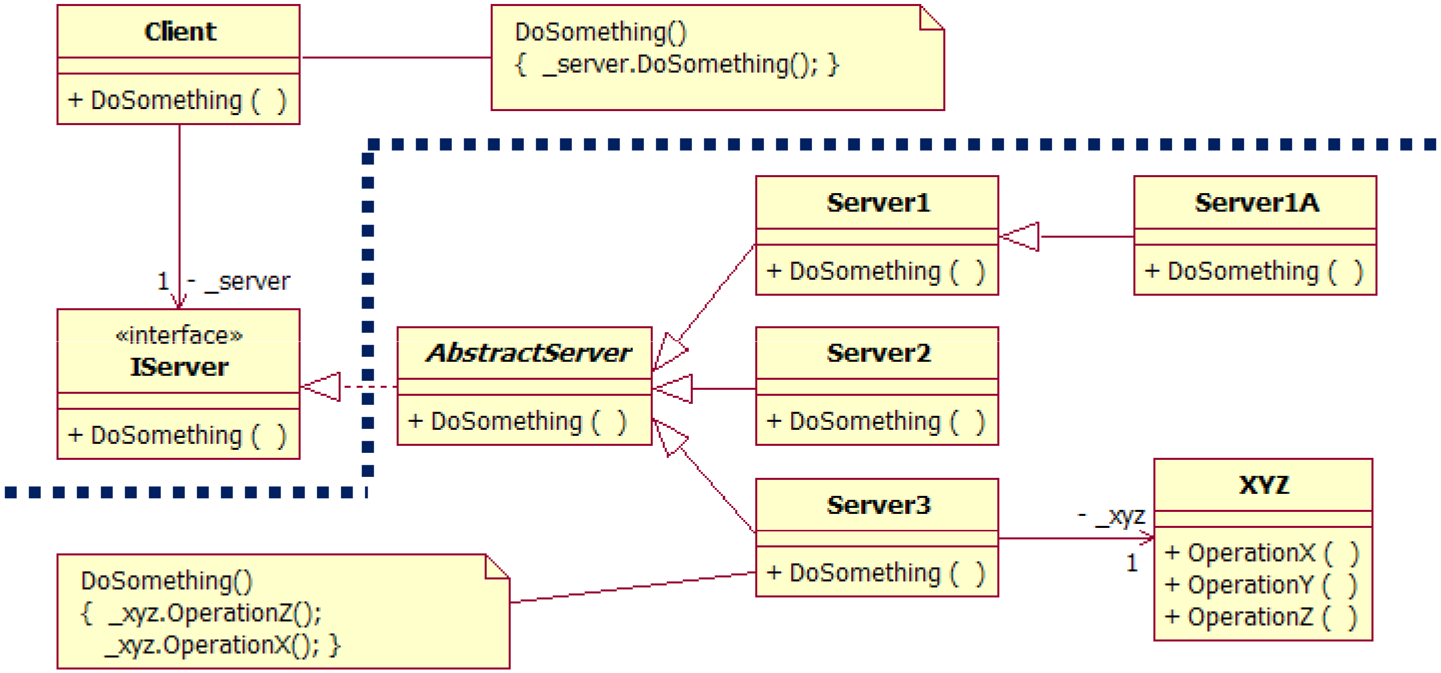
- A client of a base class should continue to function properly if a derivative of that base class is passed to it
- In altre parole: un cliente che usa istanze di una classe A deve poter usare istanze di una qualsiasi sottoclasse di A **senza accorgersi della differenza**

The Liskov Substitution Principle Example



This module should not break regardless of whether a Personal Auto or Commercial Auto or Home Owner Policy is passed to it

The Liskov Substitution Principle Example



The Liskov Substitution Principle

- OCP si basa sull’uso di classi concrete derivate da un astrazione (interfaccia o classe astratta)
- LSP costituisce una guida per creare queste classi concrete mediante l’ereditarietà
- La principale causa di violazioni al principio di Liskov è dato dalla **ridefinizione di metodi virtuali** nelle classi derivate:
  è qui che bisogna riporre la massima attenzione
- La chiave per evitare le violazioni al principio di Liskov risiede nel **Design by Contract** (B. Meyer)

Design by Contract

- Nel Design by Contract ogni metodo ha
  - un insieme di **pre-condizioni** – requisiti minimi che devono essere soddisfatti dal chiamante perché il metodo possa essere eseguito correttamente
  - un insieme di **post-condizioni** – requisiti che devono essere soddisfatti dal metodo nel caso di esecuzione corretta
- Questi due insiemi di condizioni costituiscono **un contratto tra** chi invoca il metodo (**cliente**) **e** il **metodo** stesso (e quindi la classe a cui appartiene)
  - Le **pre-condizioni vincolano il chiamante**
  - Le **post-condizioni vincolano il metodo**
  - Se il chiamante garantisce il verificarsi delle pre-condizioni, il metodo garantisce il verificarsi delle post-condizioni

Design by Contract

- Quando un metodo viene ridefinito in una sottoclasse
  - le **pre-condizioni** devono essere **identiche o meno stringenti**
  - le **post-condizioni** devono essere **identiche o più stringenti**
- Questo perché un cliente che invoca il metodo conosce il contratto definito a livello della classe base, quindi non è in grado:
  - di soddisfare pre-condizioni più stringenti o
  - di accettare post-condizioni meno stringenti
- In caso contrario, il cliente dovrebbe conoscere informazioni sulla classe derivata e questo porterebbe a una violazione del principio di Liskov

Design by Contract

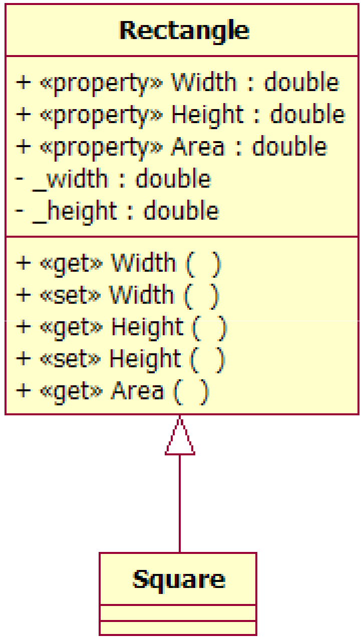
```
public class BaseClass
{
  public virtual int Calculate(int val)
  {
    Precondition(-10000 <= val && val <= 10000);
    int result = val / 100;
    Postcondition(-100 <= result && result <= 100);
    return result;
  }
}

public class SubClass : BaseClass
{
  public override int Calculate(int val)
  {
    Precondition(-20000 <= val && val <= 20000);
    int result = Math.Abs(val) / 200;
    Postcondition(0 <= result && result <= 100);
    return result;
  }
}
```

Il Quadrato è un Rettangolo?

```
public class Rectangle
{
  private double _width;
  private double _height;

  public double Width
  {
    get { return _width; }
    set { _width = value; }
  }

  public double Height
  {
    get { return _height; }
    set { _height = value; }
  }

  public double Area
  {
    get { return Width * Height; }
  }
}
```

- Imagine that this application works well, and is installed in many sites
- As is the case with all successful software, as its users' needs change, new functions are needed
- Imagine that one day the users demand the ability to manipulate squares in addition to rectangles

Il Quadrato è un Rettangolo?



- Inheritance is the IsA relationship
- In other words, if a new kind of object can be said to fulfill the IsA relationship with an old kind of object, then the class of the new object should be derived from the class of the old object
- Clearly, a square is a rectangle for all normal intents and purposes
- Since the IsA relationship holds, it is logical to model the **`Square`** class as being derived from **`Rectangle`**

Il Quadrato è un Rettangolo?

- This use of the IsA relationship is considered by many to be one of the fundamental techniques of Object Oriented Analysis
- A square is a rectangle, and so the **`Square`** class should be derived from the **`Rectangle`** class
- However this kind of thinking can lead to some subtle, yet significant, problems
- Generally these problem are not foreseen until we actually try to code the application

Il Quadrato è un Rettangolo?

```
public class Square : Rectangle
{
  public new double Width
  {
    get { return base.Width; }
    set
    {
         base.Width = value;
         base.Height = value;
    }
  }

  public new double Height
  {
    get { return base.Height; }
    set
    {
         base.Height = value;
         base.Width = value;
    }
  }
}
```

- È necessario ridefinire le proprietà **`Width`** e **`Height`**…
- Notevoli differenze tra Java e C++ / C#
  - In Java tutti i metodi sono virtuali
  - In C++ / C# è possibile definire
    - sia metodi virtuali,
    - sia **metodi non virtuali** (non polimorfici)

Il Quadrato è un Rettangolo?

```
…
Square s = new Square();
s.Width = 5;  //  5 x 5
…
… Method1(s);  //  ?
…

void Method1(Rectangle r)
{
  r.Width = 10;
}
```

- If we pass a reference to a `Square` object into `Method1`, the `Square` object will be corrupted because the height won't be changed (!)
- This is a clear violation of LSP
- `Method1` does not work for derivatives of its arguments
- **The reason for the failure is that `Width` and `Height` were not declared virtual in `Rectangle`**

Il Quadrato è un Rettangolo?

```
public class Rectangle
{
  …
  public virtual double Width
  { … }
  public virtual double Height
  { … }
  …
}

public class Square : Rectangle
{
  public override double Width
  { … }
  public override double Height
  { … }
}
```

- We can fix this easily
- However, **when the creation of a derived class causes us to make changes to the base class, it often implies that the design is faulty**
- Indeed, it violates the OCP
- We might counter this with argument that forgetting to make **`Width`** and **`Height`** virtual was the real design flaw, and we are just fixing it now
- However, this is hard to justify since setting the height and width of a rectangle are exceedingly primitive operations – by what reasoning would we make them virtual if we did not anticipate the existence of square

Il Quadrato è un Rettangolo?

- At this point in time we have two classes, `Square` and `Rectangle`, that appear to work
- No matter what you do to a `Square` object, it will remain consistent with a mathematical square
- And regardless of what you do to a `Rectangle` object, it will remain a mathematical rectangle
- Moreover, you can pass a `Square` into a function that accepts a reference to a `Rectangle`, and the `Square` will still act like a square and will remain consistent
- Thus, we might conclude that the model is now self consistent, and correct
- However, **a model that is self consistent is not necessarily consistent with all its users**!

Il Quadrato è un Rettangolo?

```
public void Scale(Rectangle rectangle)
{
  rectangle.Width = rectangle.Width * ScalingFactor;
  rectangle.Height = rectangle.Height * ScalingFactor;
}
```

- **`Scale`** invokes members of what it believes to be a **`Rectangle`**
- Substituing a **`Square`** into this will result in the square being scaled twice!
- So here is **the real problem**:
  was the programmer who wrote **`Scale`** justified in assuming that changing the width of a **`Rectangle`** leaves its height unchanged?

Il Quadrato è un Rettangolo?

- Clearly, the programmer of **`Scale`** made this very reasonable assumption
- Passing a **`Square`** to functions whose programmers made this assumption will result in problems
- Therefore, there exist functions that take references to **`Rectangle`** objects, but cannot operate properly upon **`Square`** objects
- These functions expose a violation of the LSP
- The addition of the **`Square`** derivative of **`Rectangle`** has broken these function ▶ the OCP has been violated

Il Quadrato è un Rettangolo?

- Why did the model of the `Square` and `Rectangle` go bad
  - After all, isn't a `Square` a `Rectangle`?
  - Doesn't the IsA relationship hold?
- No! A square might be a rectangle, but a `Square` object is definitely not a `Rectangle` object
- Why? Because **the behavior of a `Square` object is not consistent with the behavior of a `Rectangle` object**
- Behaviorally, a `Square` is not a `Rectangle`! And it is behavior that software is really all about

Il Quadrato è un Rettangolo?

- The LSP makes clear that **in OOD the IsA relationship pertains to behavior**. Not intrinsic private behavior, but **extrinsic public behavior**; behavior that clients depend upon
- For example, the author of **`Scale`** depended on the fact that rectangles behave such that **their height and width vary independently of one another**.
  That independence of the two variables is an **extrinsic public behavior** that other programmers are likely to depend upon
- In order for the LSP to hold, and with it the OCP, **all derivatives must conform to the behavior that clients expect of the base classes that they use**

Il Quadrato è un Rettangolo?

- The rule for the preconditions and postconditions for derivatives, is: “when redefining a routine, you may only replace **its precondition by a weaker one**, and **its postcondition by a stronger one**”
- In other words, when using an object through its base class interface, **the user knows only the preconditions and postconditions of the base class**
- Thus, derived classes must not expect such users to obey preconditions that are stronger then those required by the base class
  ▶ they must accept anything that the base class could accept
- Also, derived classes must conform to all the postconditions of the base class
  ▶ their behaviors and outputs must not violate any of the constraints established for the base class

Il Quadrato è un Rettangolo?

- The contract of **`Rectangle`**
  - height and width are independent, can set one while the other remains unchanged
- **`Square`** violates the contract of the base class

Il Quadrato è un Rettangolo?

- If we look at the test fixture for our **`Rectangle`** we get some idea of the contract for a **`Rectangle`**:

```
[TestFixture]
public class RectangleFixture
{
  [Test]
  public void SetHeightAndWidth()
  {
    Rectangle rectangle = new Rectangle();
    int expectedWidth = 3, expectedHeight = 7;
    rectangle.Width = expectedWidth;
    rectangle.Height = expectedHeight;
    Assertion.AssertEquals(expectedWidth, rectangle.Width);
    Assertion.AssertEquals(expectedHeight, rectangle.Height);
  }
}
```

Il Quadrato è un Rettangolo?

```
[TestFixture] public class RectangleFixture
{
  [Test] public void SetHeightAndWidth()
  {
   Rectangle rectangle = GetShape();
   ...
  }
 protected virtual Rectangle GetShape()
  { return new Rectangle(); }
}

[TestFixture] public class SquareFixture : RectangleFixture
{
 protected override Rectangle GetShape()
  { return new Square(); }
}
```