Object design

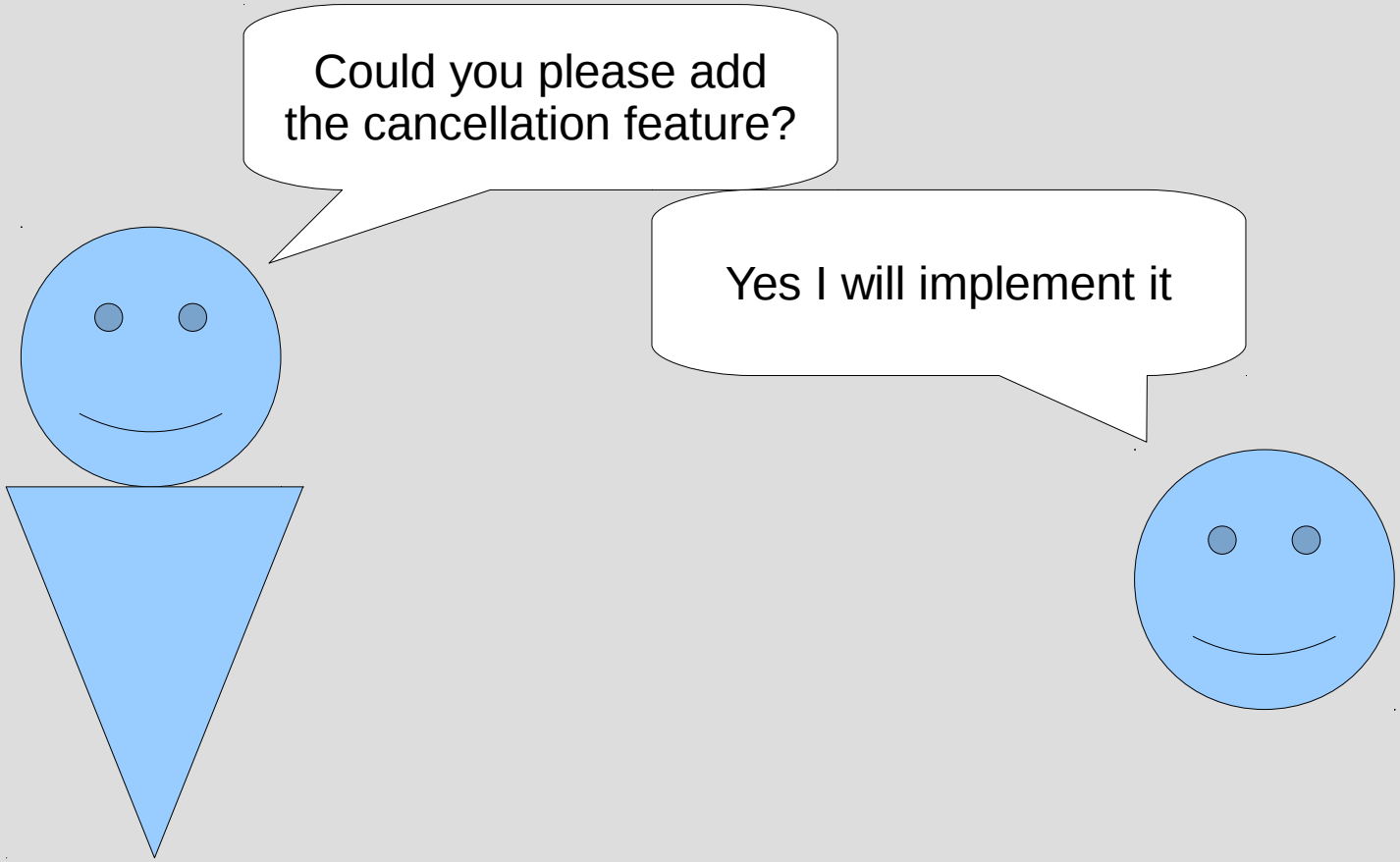
François Schwarzentruber
ENS Cachan – Antenne de Bretagne

Symptoms of rotting systems (according to Robert C. Martin)

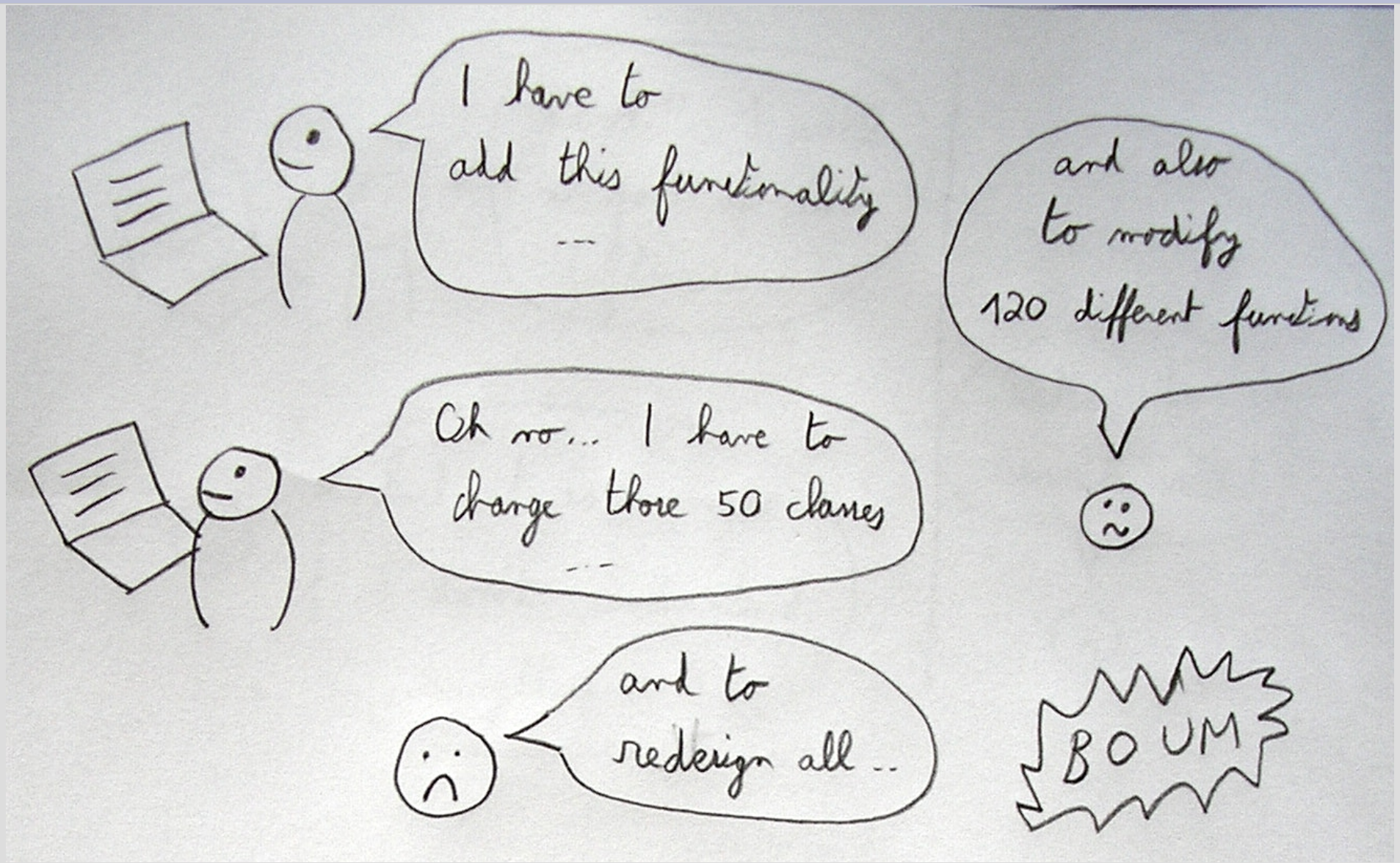
Four behaviors of the developer team:

- Rigidity
- Fragility
- Immobility
- Viscosity

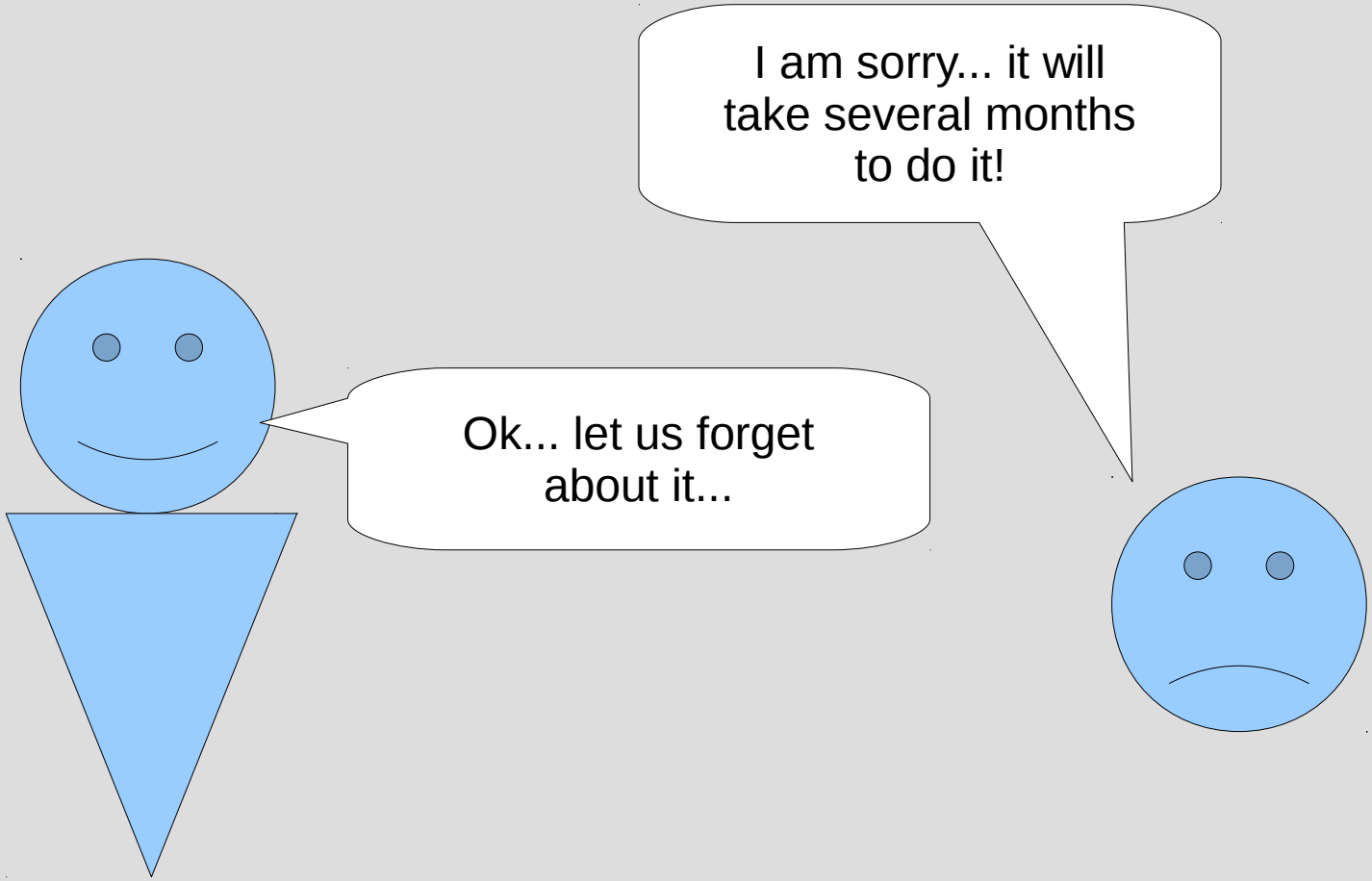
Rigidity


Could you please add the cancellation feature?
Yes I will implement it

Rigidity

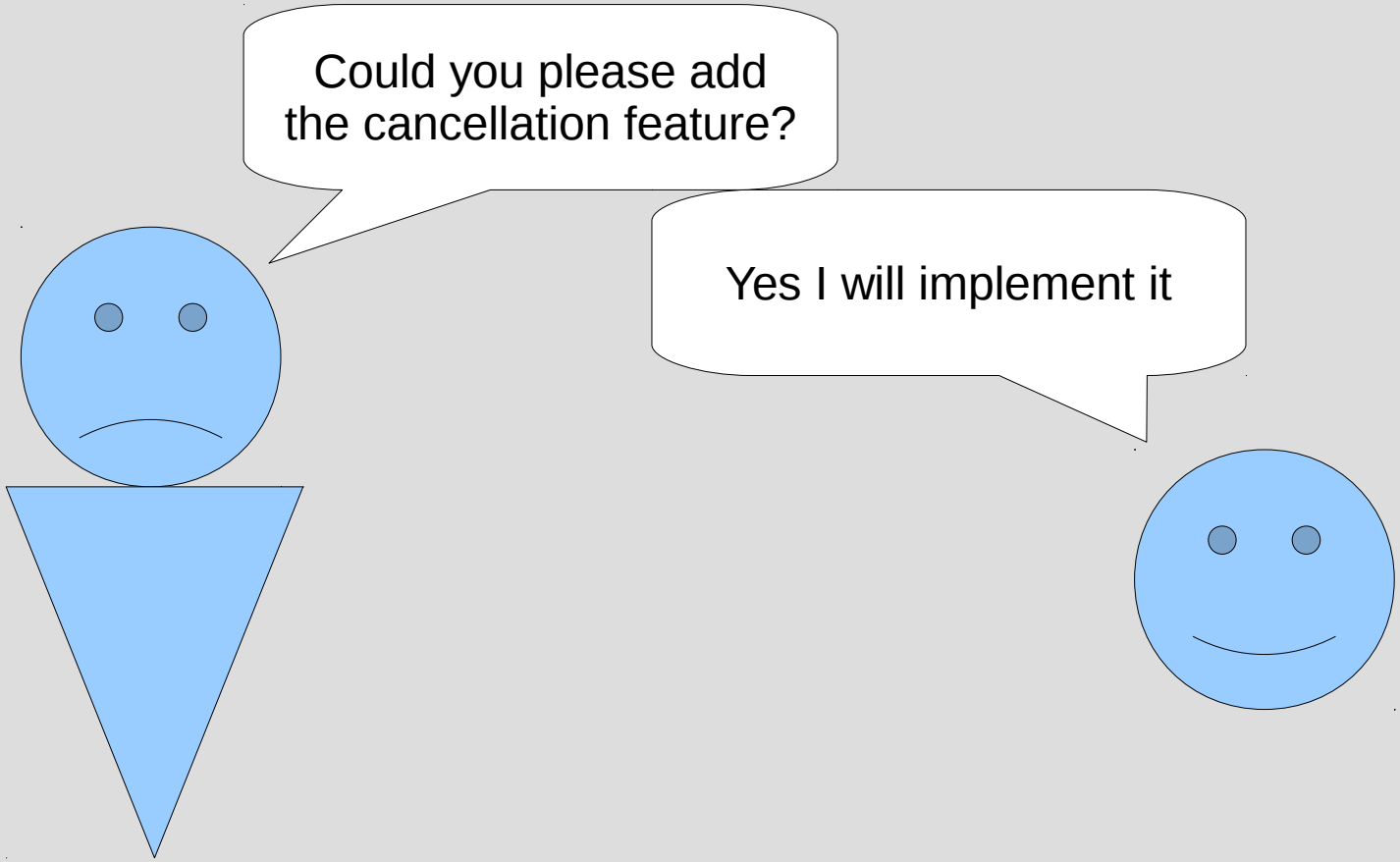

I have to add this functionality ...
and also to modify 120 different functions
Oh no... I have to change those 50 classes ...
and to redesign all ..
BOUM

Rigidity

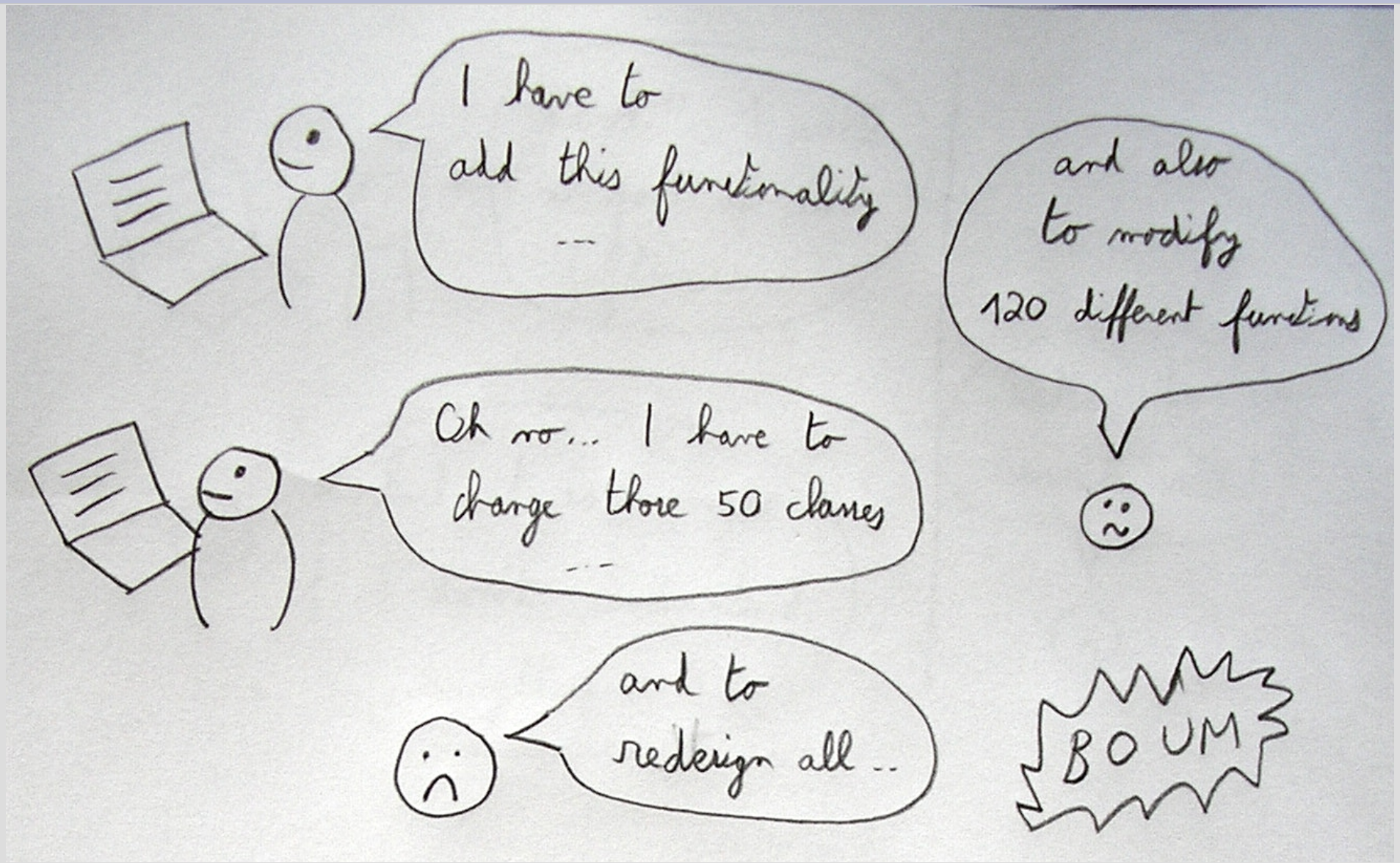


This functionality will never be implemented.

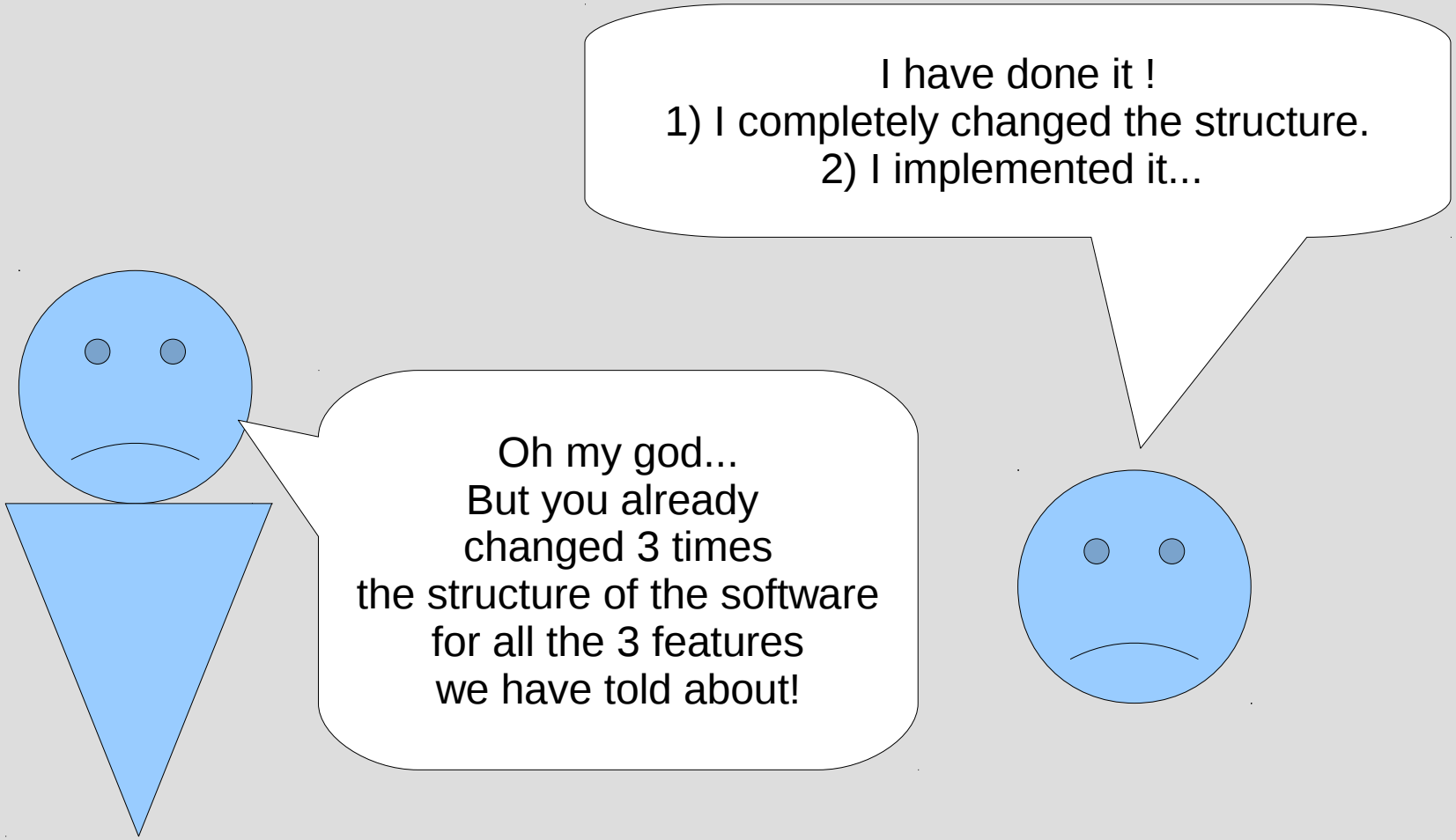
Fragility


Could you please add the cancellation feature?
Yes I will implement it

Fragility

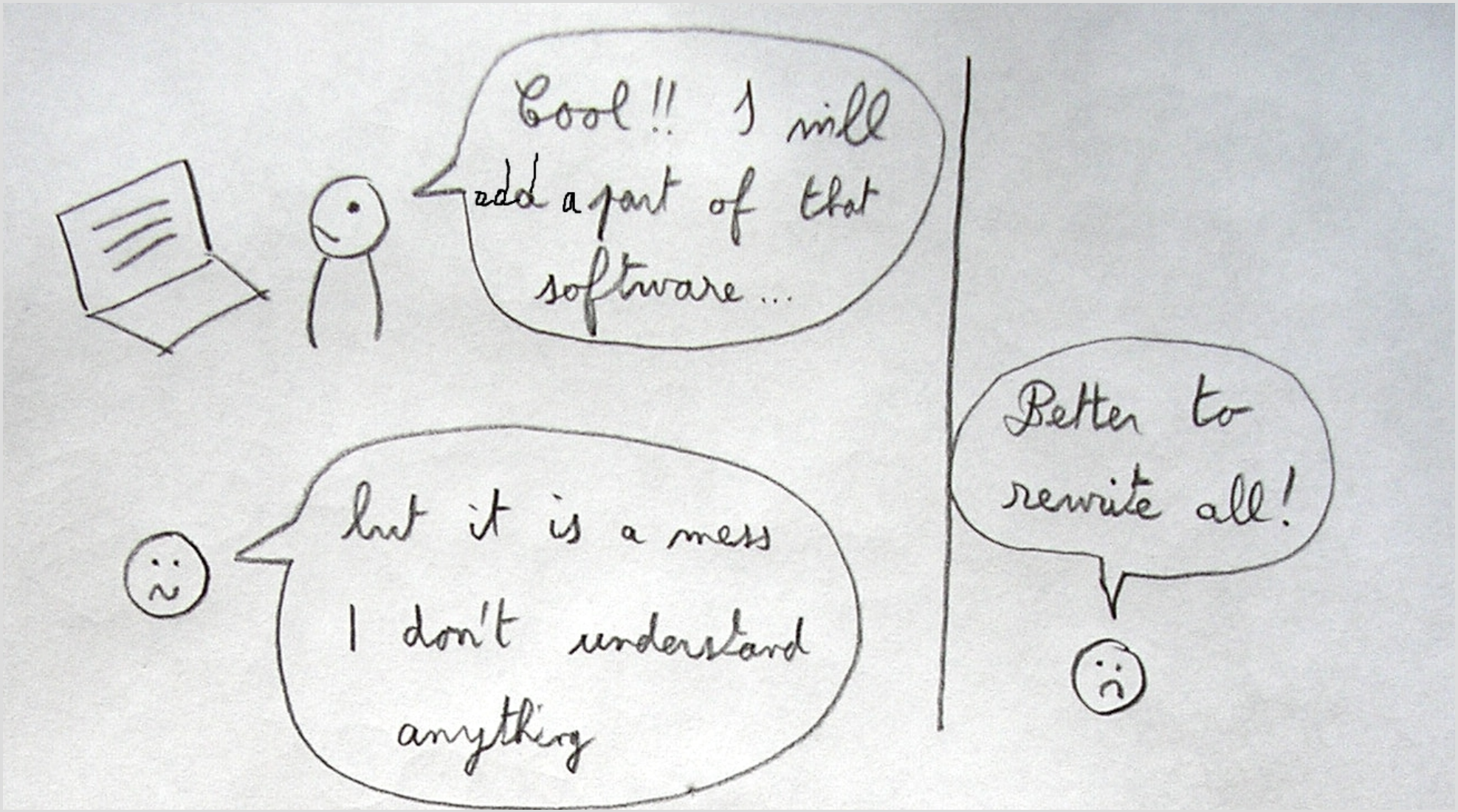

I have to add this functionality ...
and also to modify 120 different functions
Oh no... I have to change those 50 classes ...
and to redesign all ..
BOUM

Fragility

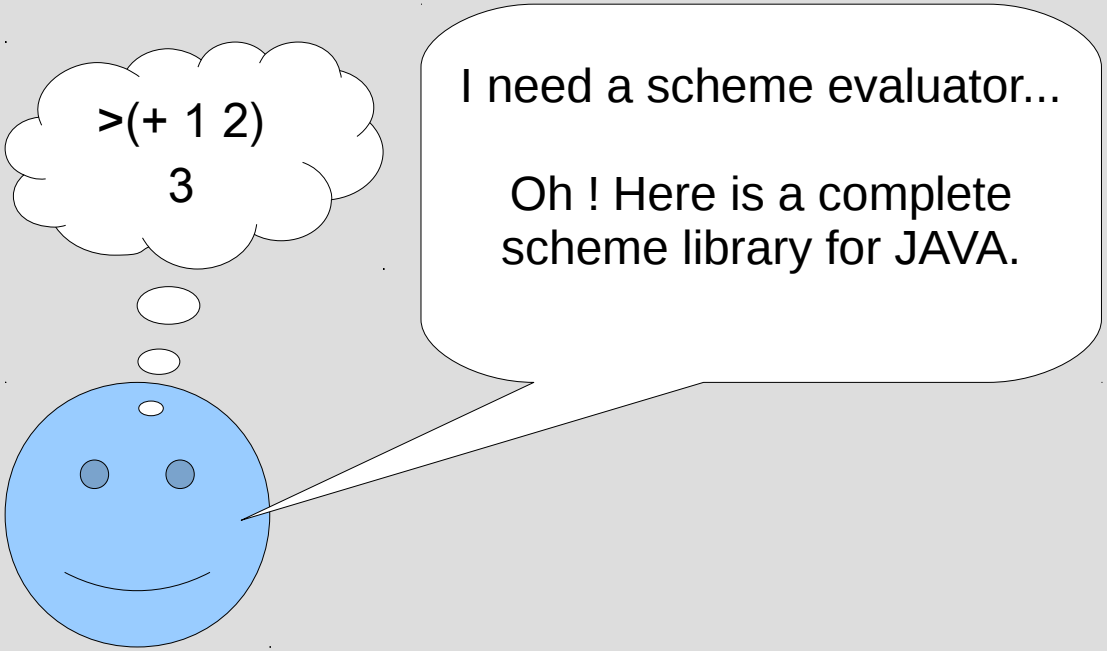


The developer team is not to be trusted.

Immobility


Cool!! I will add a part of that software...
but it is a mess I don't understand anything
Better to rewrite all!

Immobility

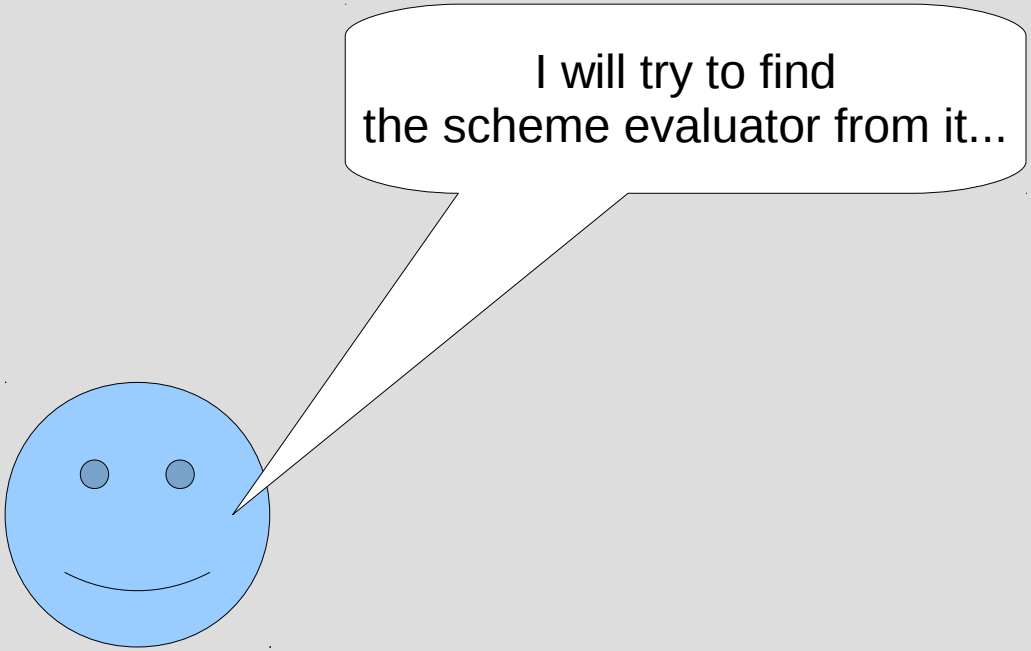


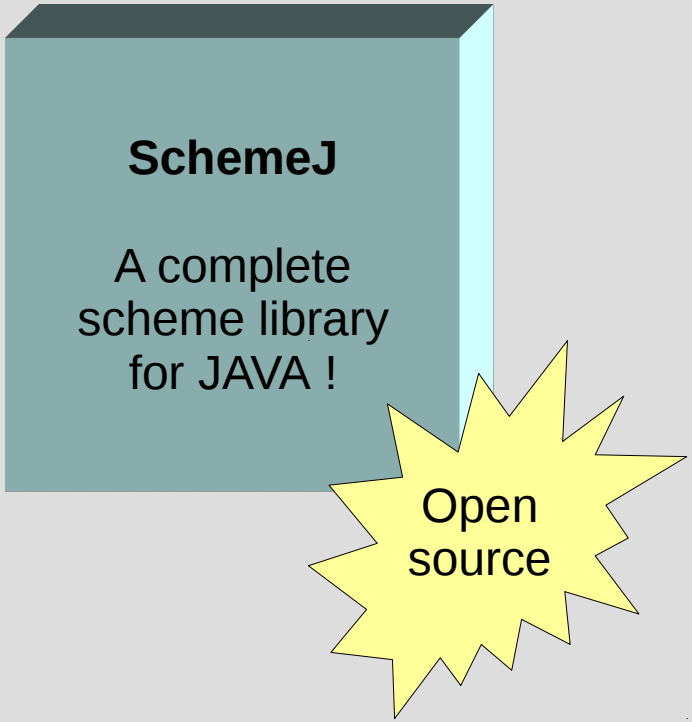
SchemeJ

A complete scheme library for JAVA !

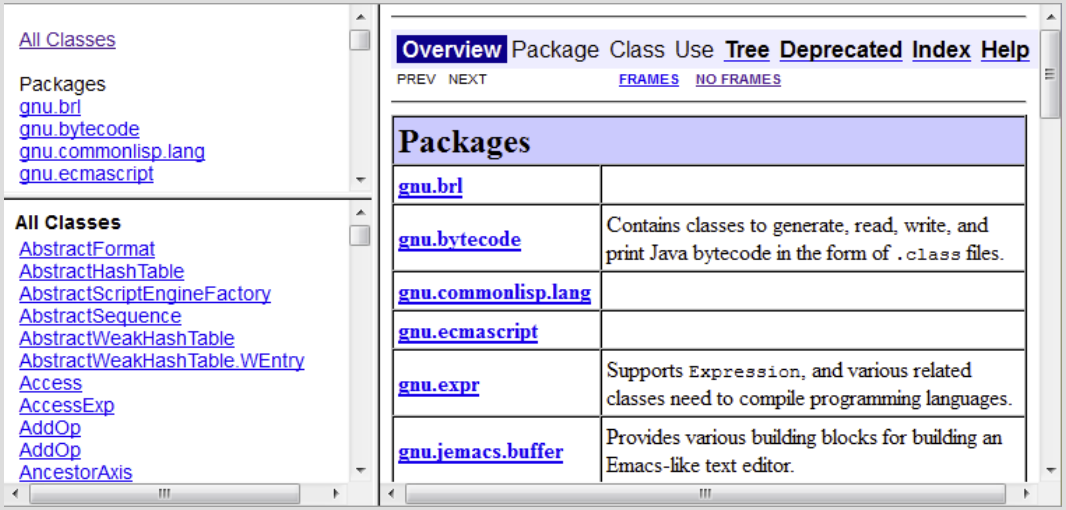
Open source

Immobility


I will try to find
the scheme evaluator from it...


SchemeJ
A complete
scheme library
for JAVA !
Open
source

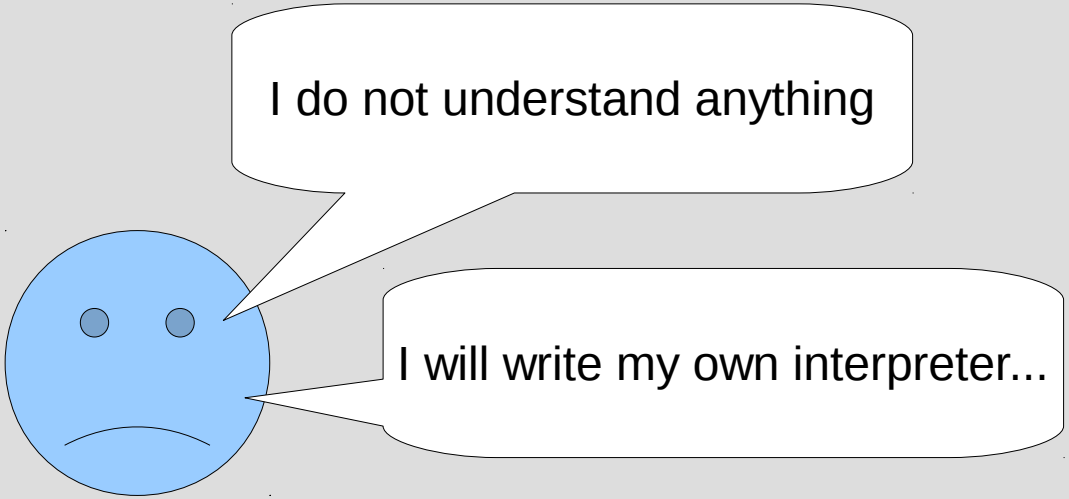
Immobility



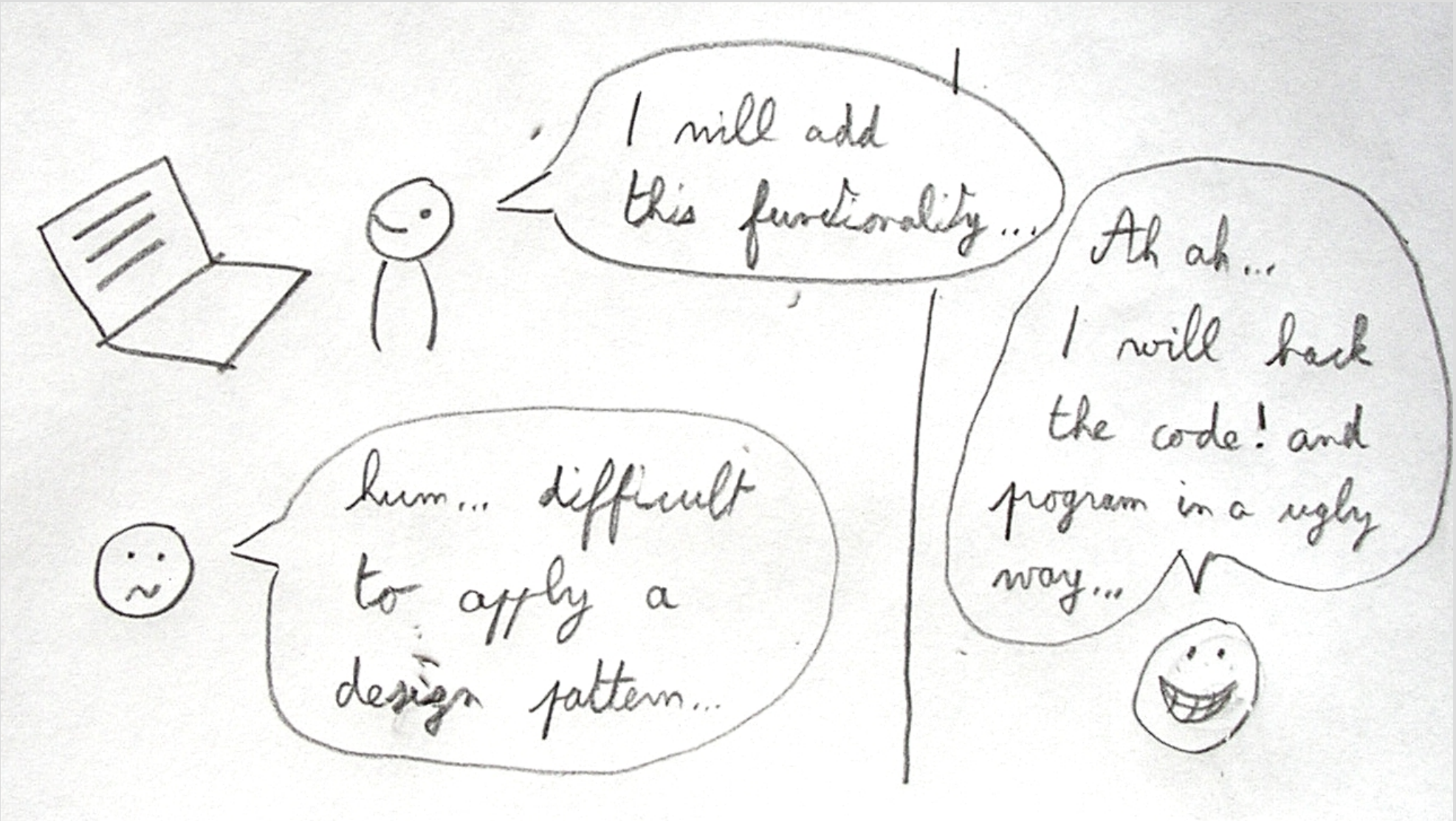
SchemeJ

A complete scheme library for JAVA !

Open source



Viscosity


I will add this functionality...
hum... difficult to apply a design pattern...
Ah ah... I will hack the code! and program in a ugly way...

Viscosity

Another difficulty, on the implementation side this time, is that a lot of the code used in the query part of [redacted] is shared with the [redacted] module, which we can not use. This precise point has made the implementation far more difficult that I expected at the beginning of the internship. The decision for this internship was to duplicate functionality, but a far better approach would be to rewrite a significant part of [redacted] back-end to have a proper separation for every concept. This would have costed far too much time for the duration of the internship.

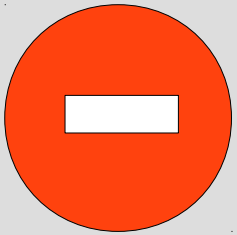
from a MIT2 internship report...

SOLID

- 5 principles of object oriented class design
- Introduced by Robert C. Martin

S : Single responsibility principle

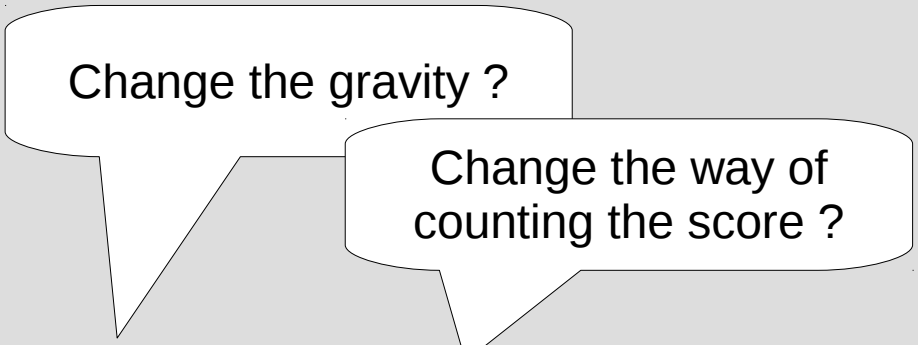
There should never be more than one reason for a class to change.



- Class of a game :

  - that computes the position of enemies

  - that computes the score



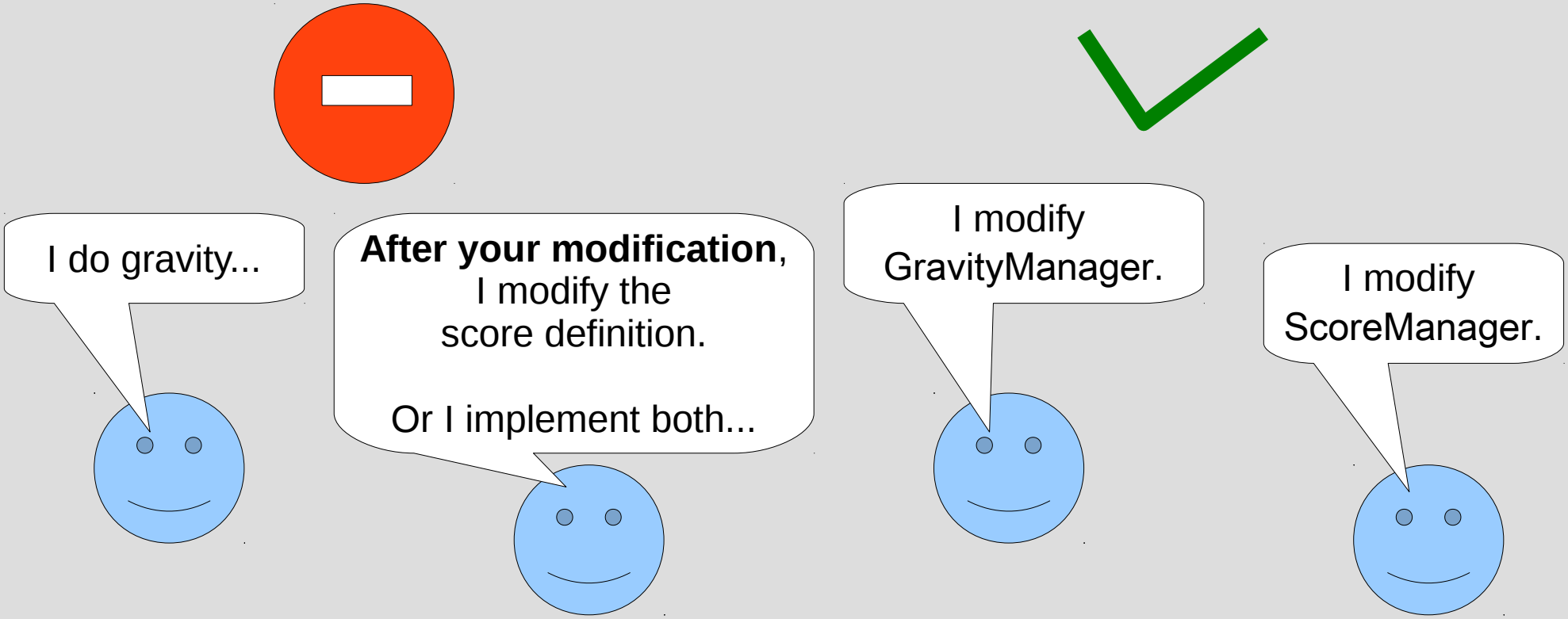
- Class of a game that uses two objects :

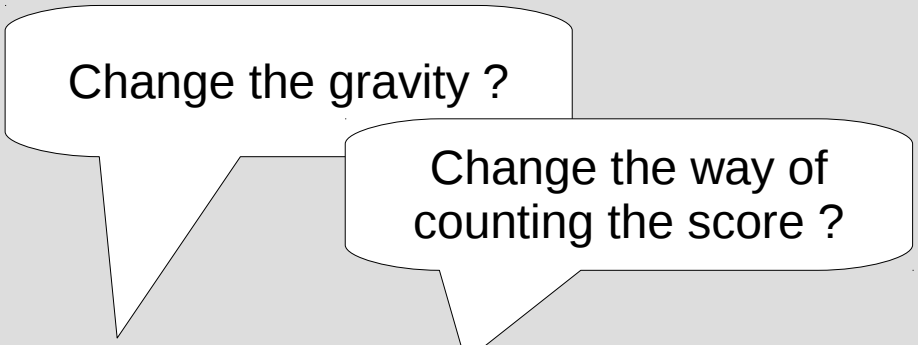
  - one that computes the position of enemies

  - another that computes the score

S : Single responsibility principle

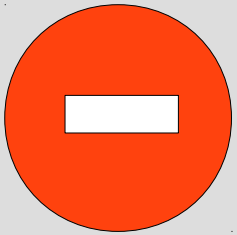
There should never be more than one reason for a class to change.

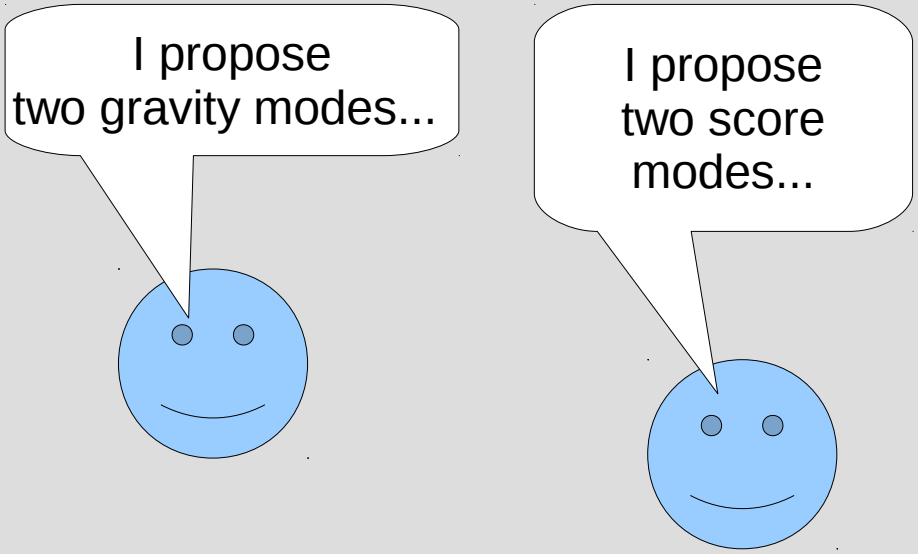


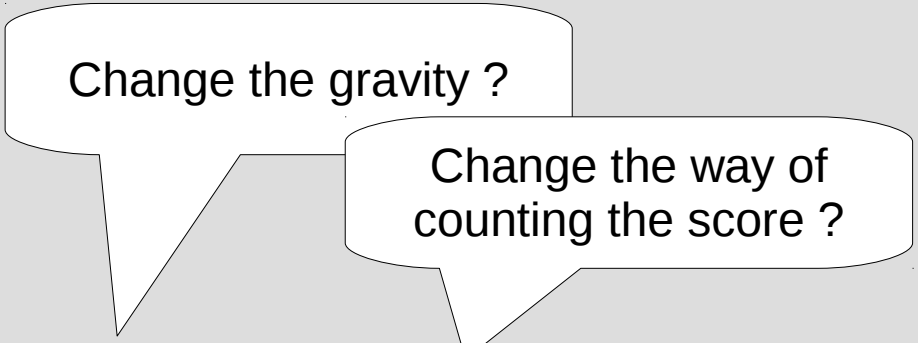


S : Single responsibility principle

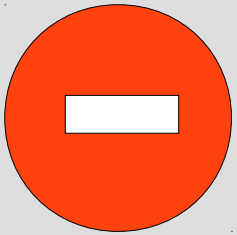
There should never be more than one reason for a class to change.



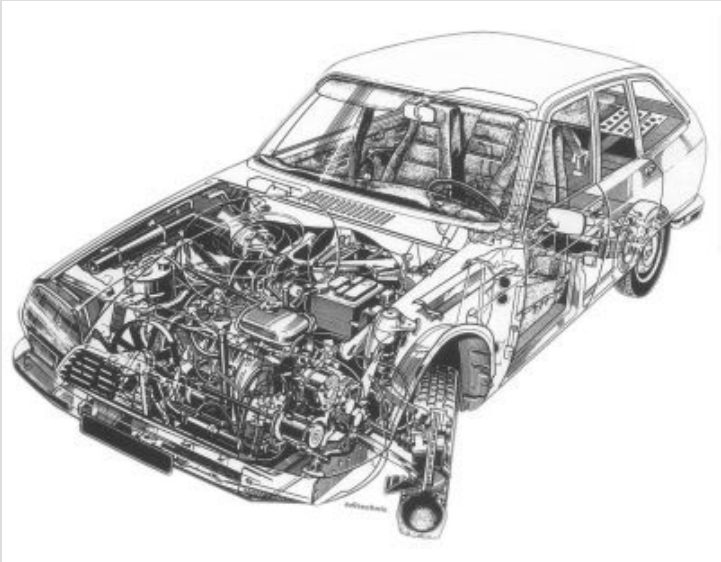




O : Open Closed Principle



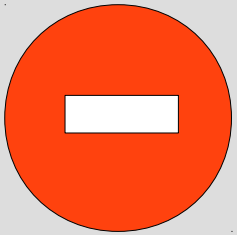
- Change the code source of module to add functionnality



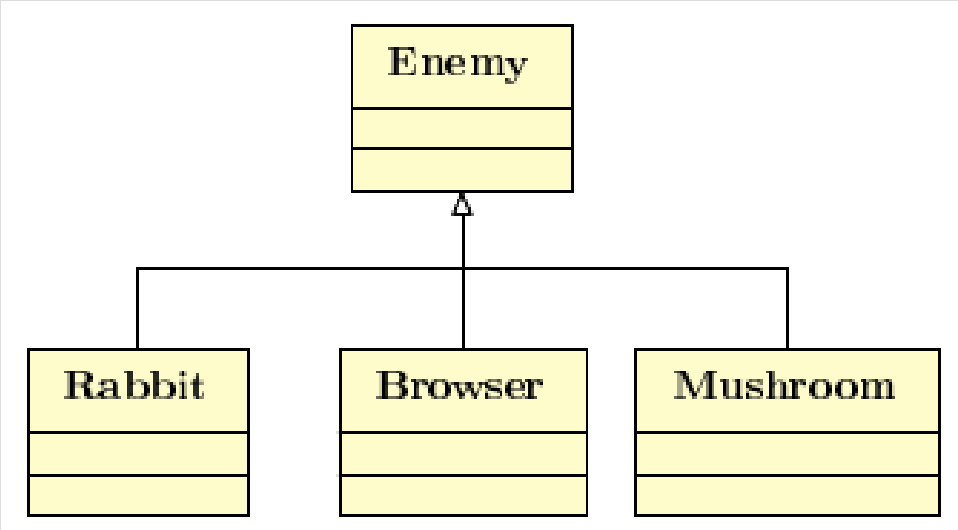
- Being able to extend modules without changing the code source

→ abstraction

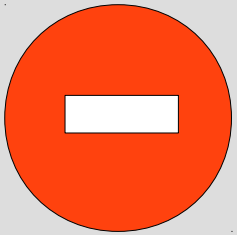
O : Open Closed Principle



```
Class Enemy
{
    void move()
    {
        if(type == RABBIT)
            ...
        else if(type == BROWSER)
            ...
        else if(type == MUSHROOM)
            ...
    }
}
```

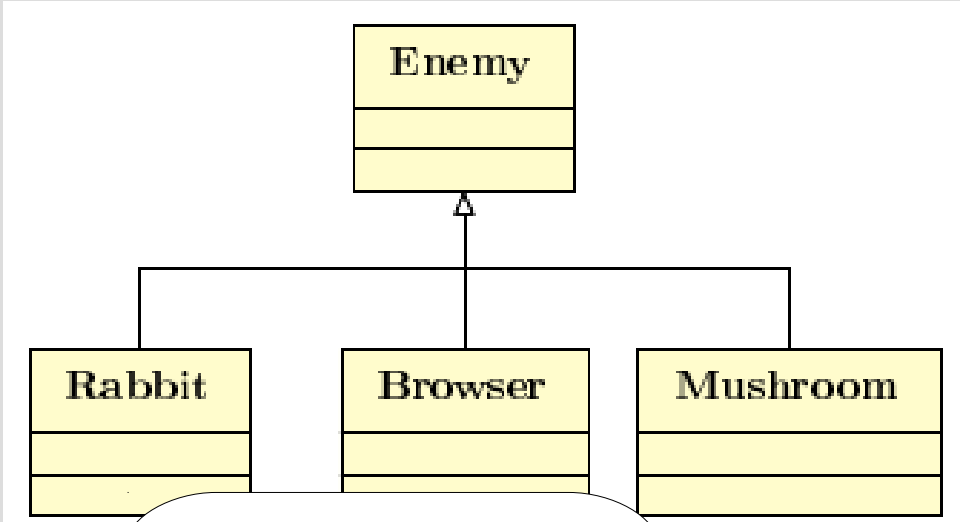


O : Open Closed Principle



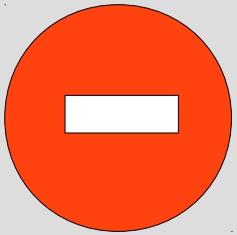
```
Class Enemy
{
    void move()
    {
        if(type == RABBIT)
            ...
        else if(type == BROWSER)
            ...
        else if(type == MUSHROOM)
            ...
    }
}
```

For each change of
a type of enemy,
I recompile all
the class Enemy...

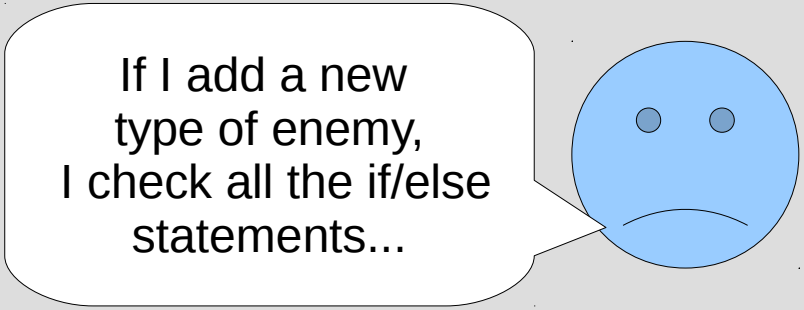


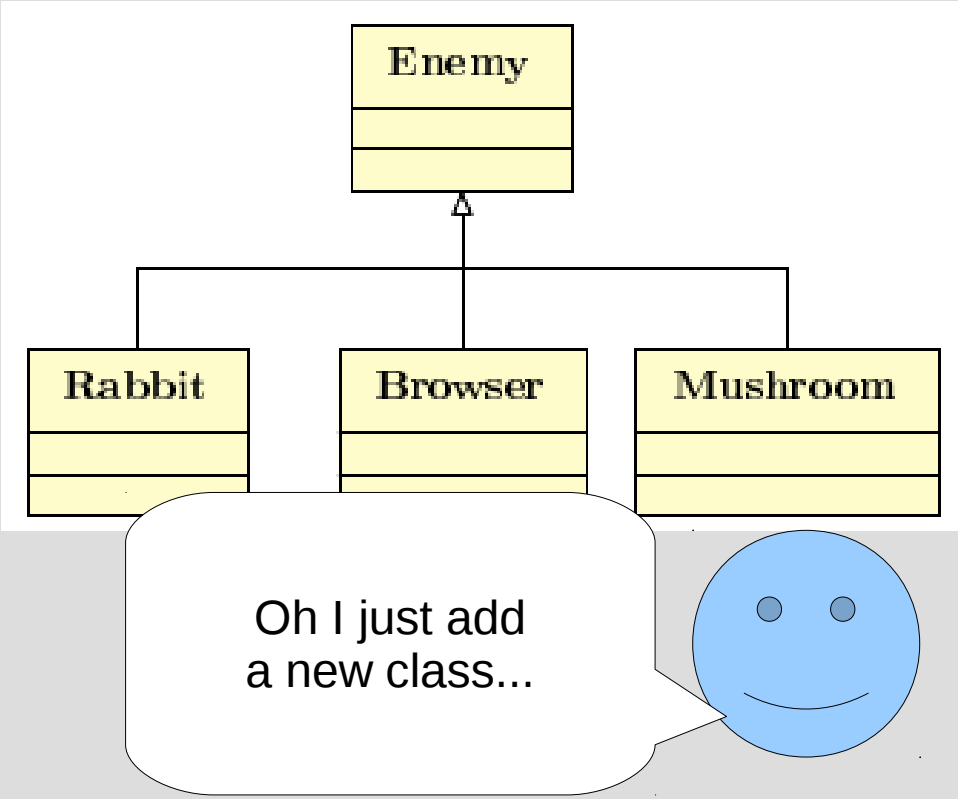
Oh I just recompile
one class...

O : Open Closed Principle

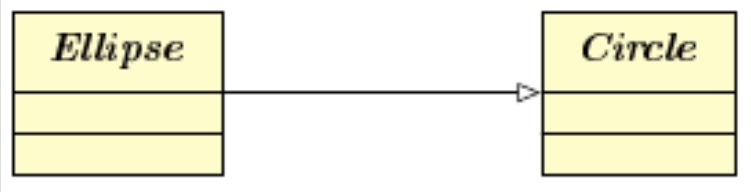


```
Class Enemy
{
     void move()
     {
          if(type == RABBIT)
               ...
          else if(type == BROWSER)
               ...
          else if(type == MUSHROOM)
               ...
     }

     void jump()
     {
          if(type == RABBIT)
               ...
          else if(type == BROWSER)
               ...
          else if(type == MUSHROOM)

     }
     .
     .
     .
}
```



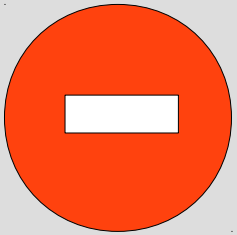


L : Liskov Substitution Principle

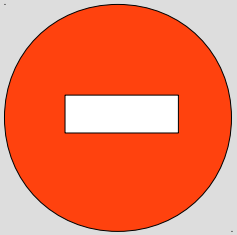


Barbara Liskov
Turing award 2008

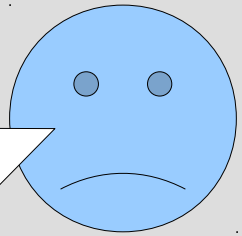
L : Liskov Substitution Principle



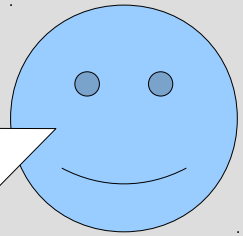
L : Liskov Substitution Principle



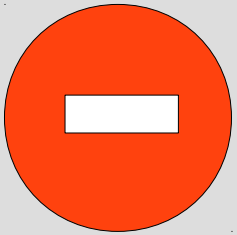
But a circle is simplier...
And we extend it to
make a ellipse...

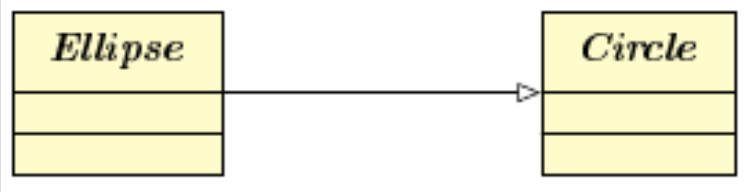


It seems reasonable...

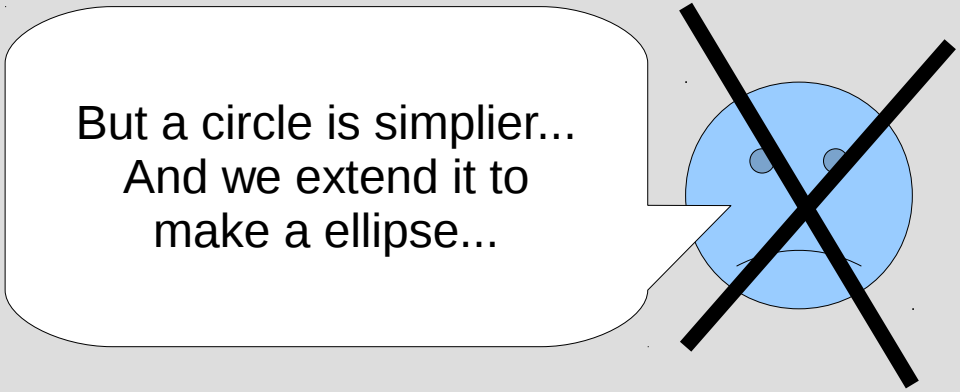


L : Liskov Substitution Principle





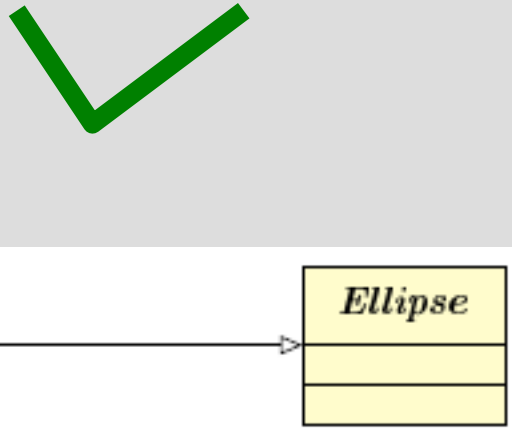

```
Class Circle
{
    public float getR();
    public float getArea();

}

Class Ellipse extends Circle
{
    ...
}
```



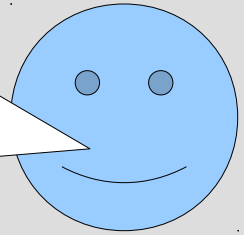

```
Circle c;
c = new Ellipse(...);

/* Here we expect that
the area of c is
c.getR()^2 * PI
*/
```

L : Liskov Substitution Principle and design by contract

```
Class Ellipse
{
    void setFocus(Point p1, p2)
    {
        this.p1 = p1;
        this.p2 = p2;
    }
    :
}

Class Circle extends Ellipse
{
    void setFocus(Point p1, p2)
    {
        this.p1 = p1;
        this.p2 = p1;
    }
    :
}

:
```

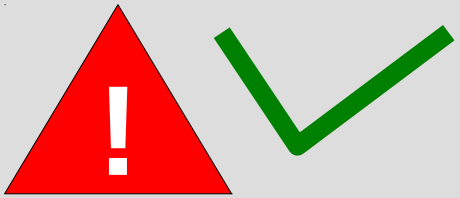



```
Ellipse e = new Circle();
e.setFocus(p1, p2);

//and e is an ellipse!
```

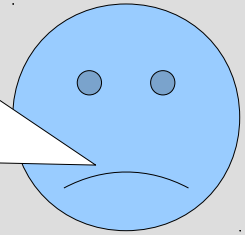


L : Liskov Substitution Principle and design by contract

```
Class Ellipse
{
    postcondition:
        this.p1 == p1 & this.p2 == p2
    void setFocus(Point p1, p2)
    {
        this.p1 = p1;
        this.p2 = p2;
    }
    :
}

Class Circle extends Ellipse
{
    void setFocus(Point p1, p2)
    {
        this.p1 = p1;
        this.p2 = p1;
    }
    :
}

:
```

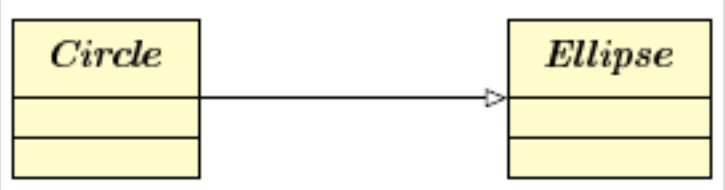


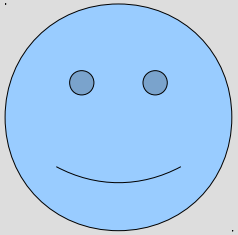

```
Ellipse e = new Circle();
e.setFocus(p1, p2);

assert(e.getP1() == p1);
assert(e.getP2() == p2);
```



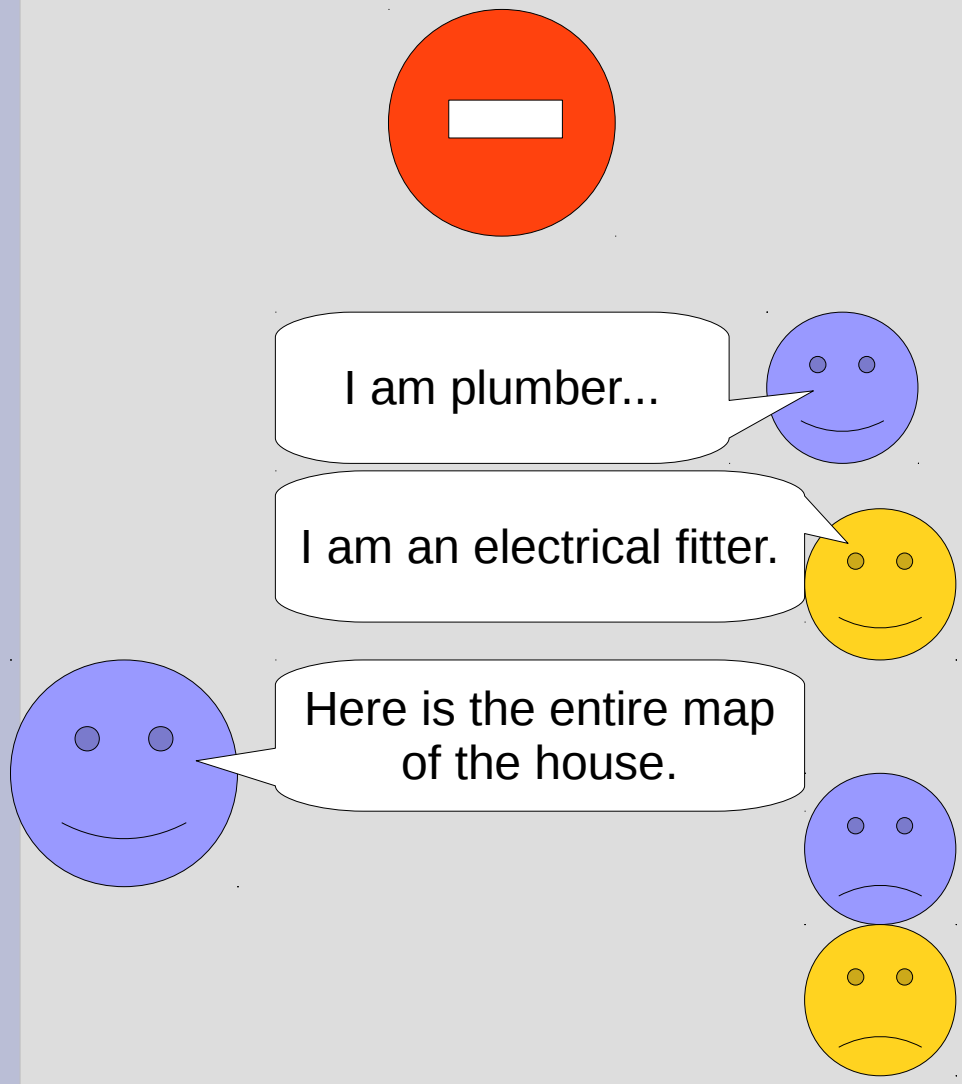
L : Liskov Substitution Principle and design by contract

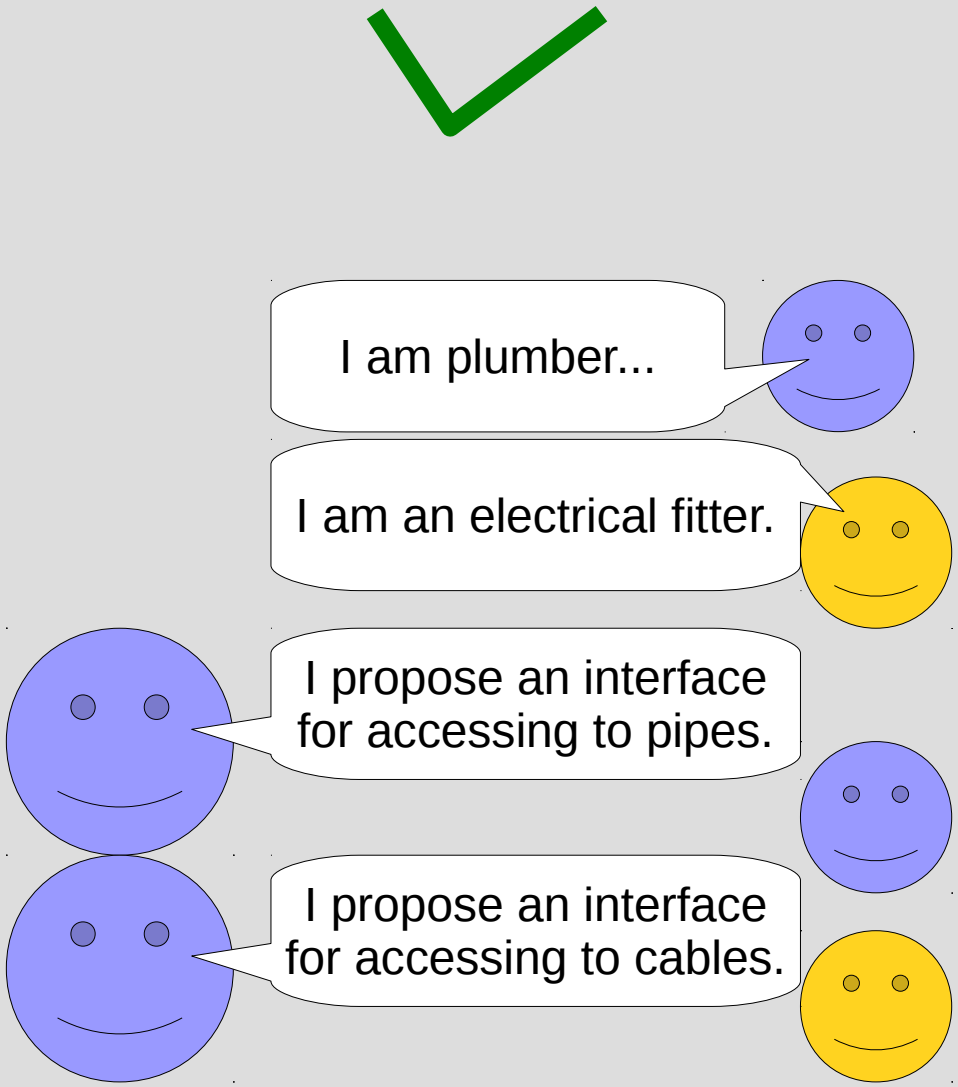
```
Class Ellipse
{
    invariant: inv

    precondition: pre
    postcondition: pos
    void setFocus(Point p1, p2)
    {
    }
    :
}

Class Circle extends Ellipse
{
    invariant: stronger than inv

    precondition: weaker than pre
    postcondition: stronger than pos
    void setFocus(Point p1, p2)
    {
    }
    :
}
```



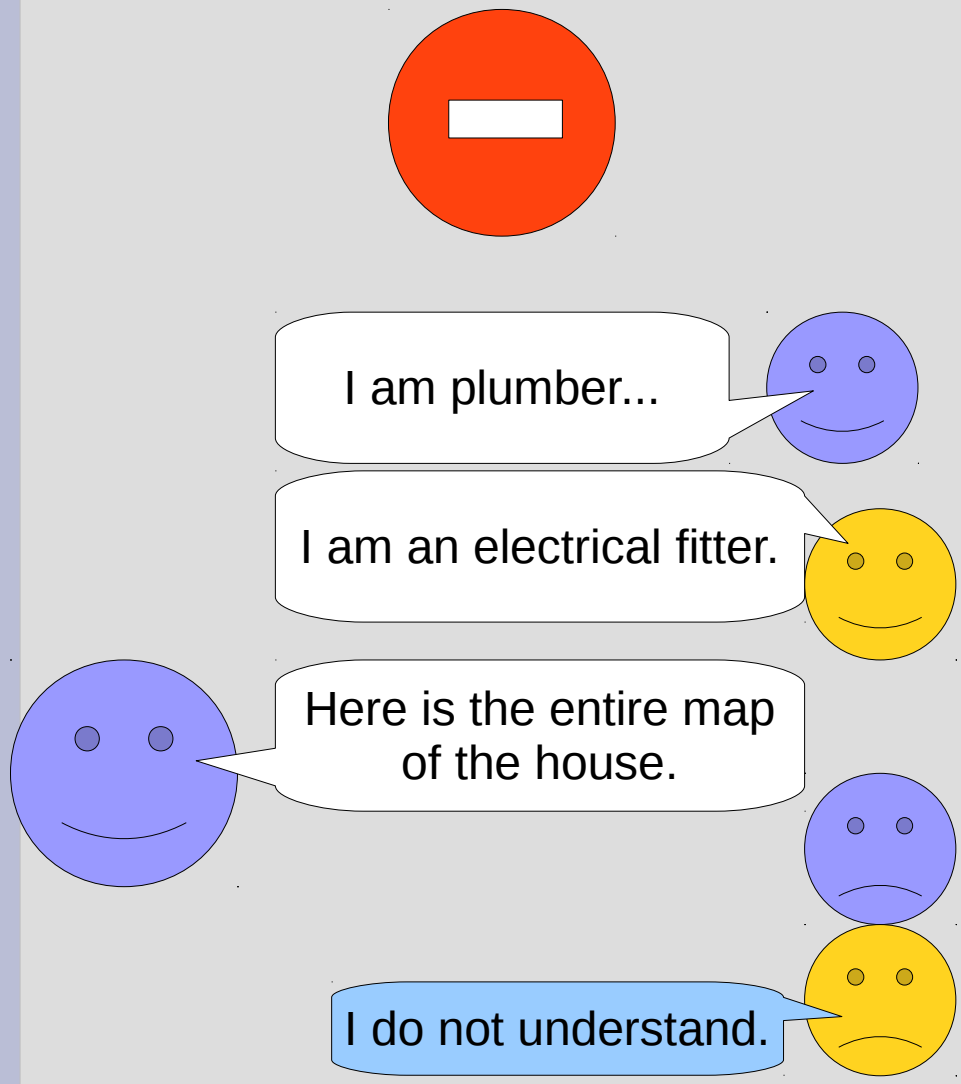


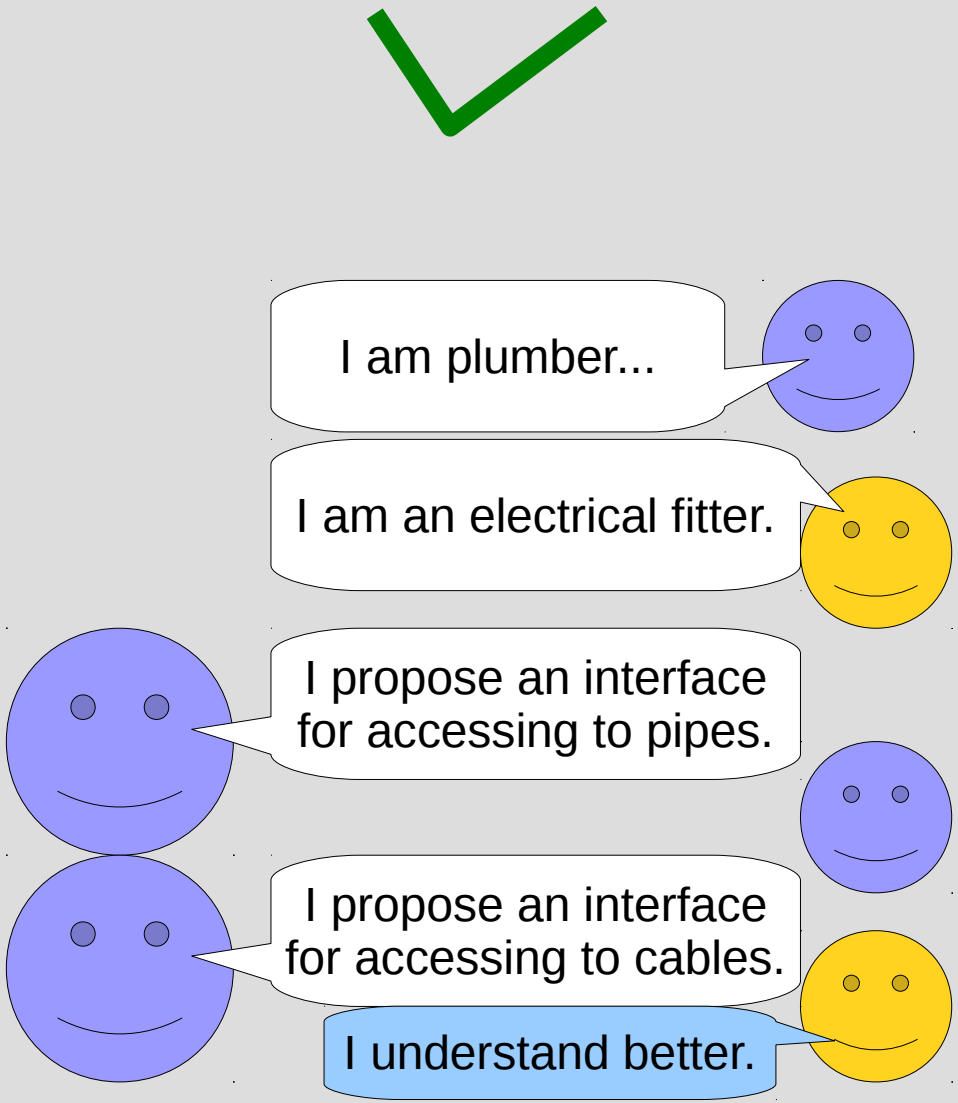
I : Interface Segregation Principle



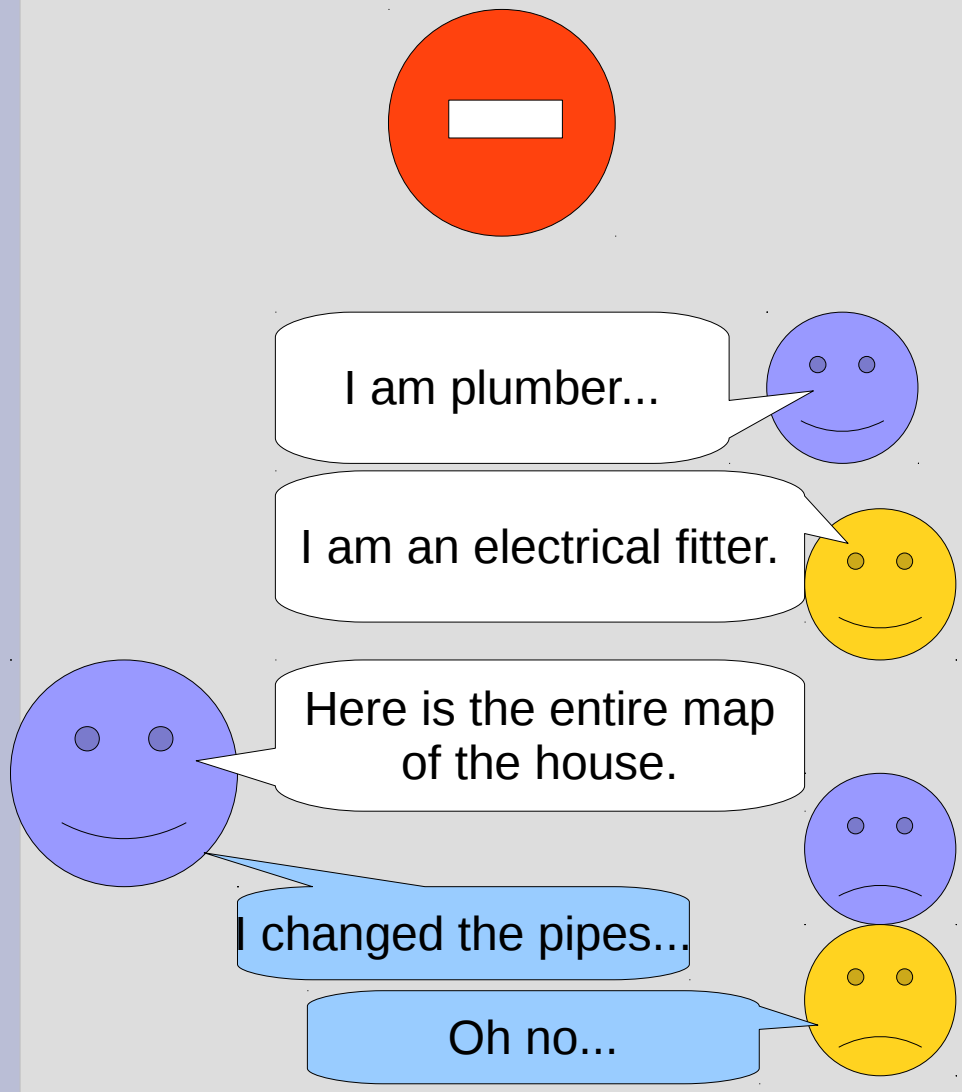


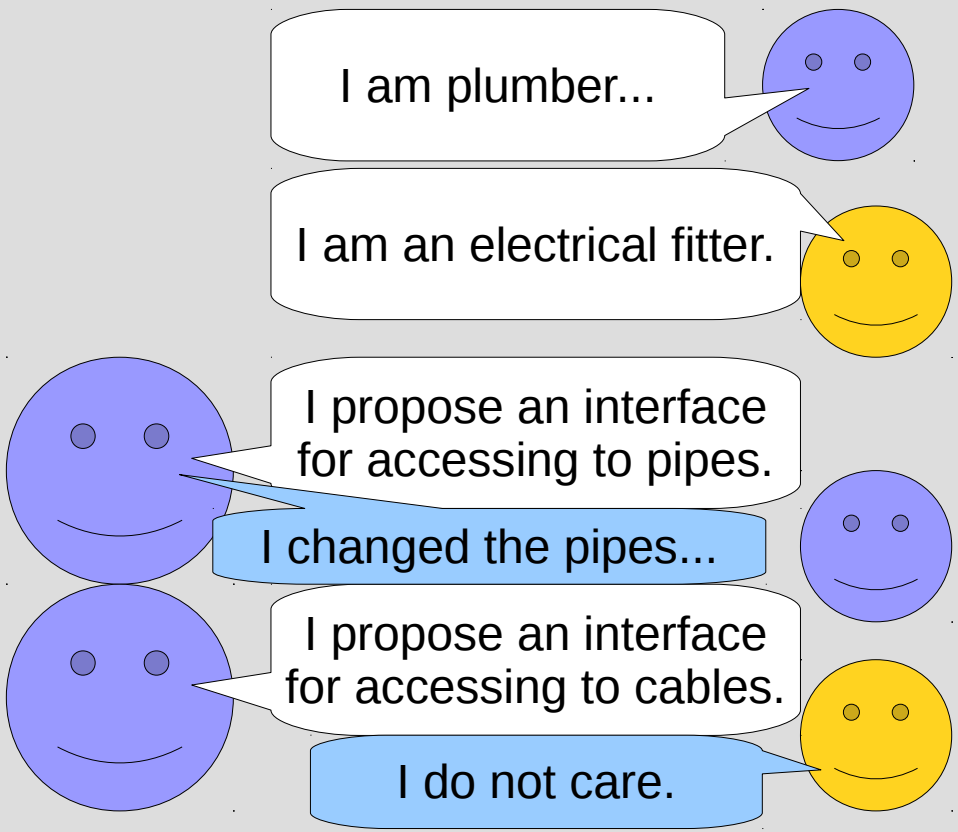
I : Interface Segregation Principle



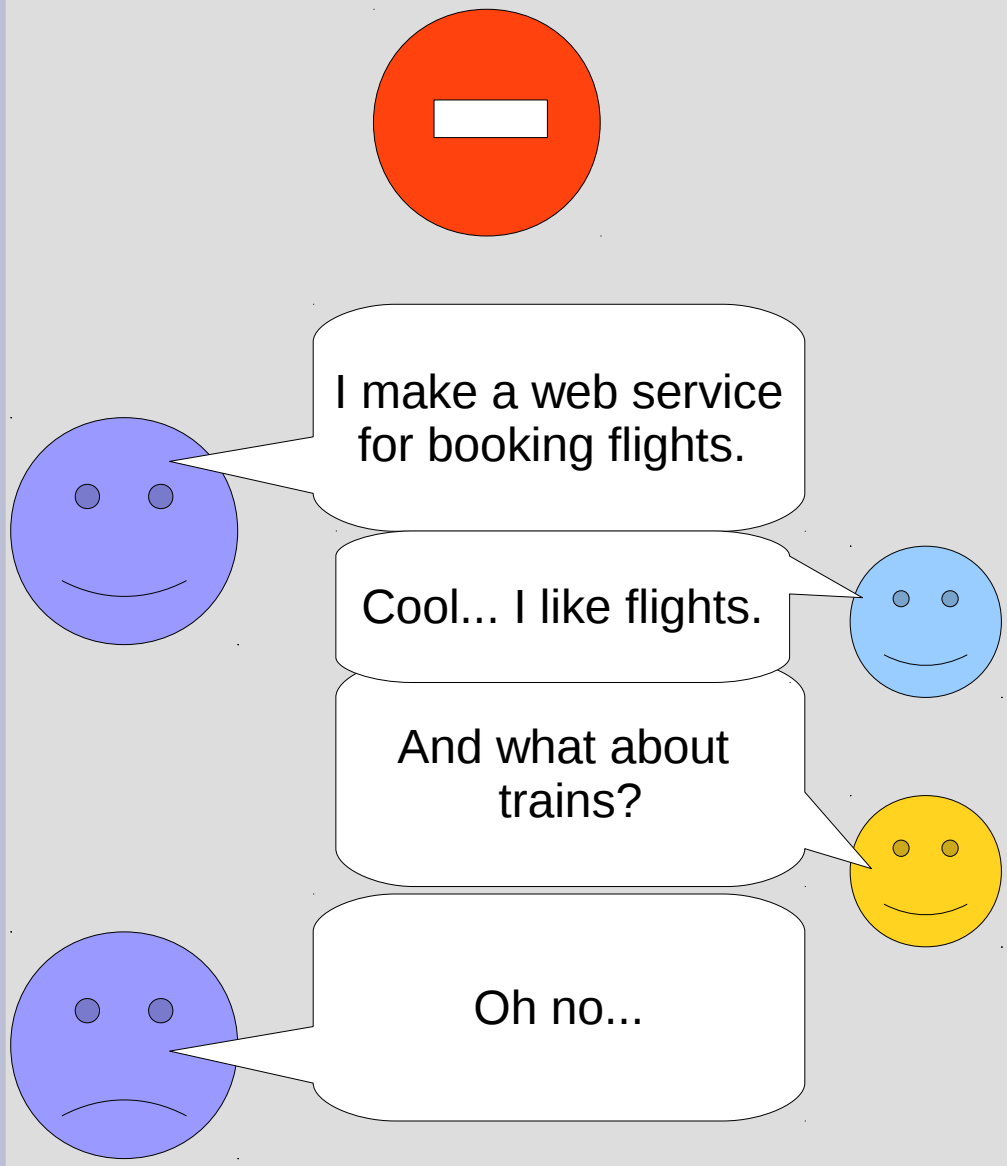


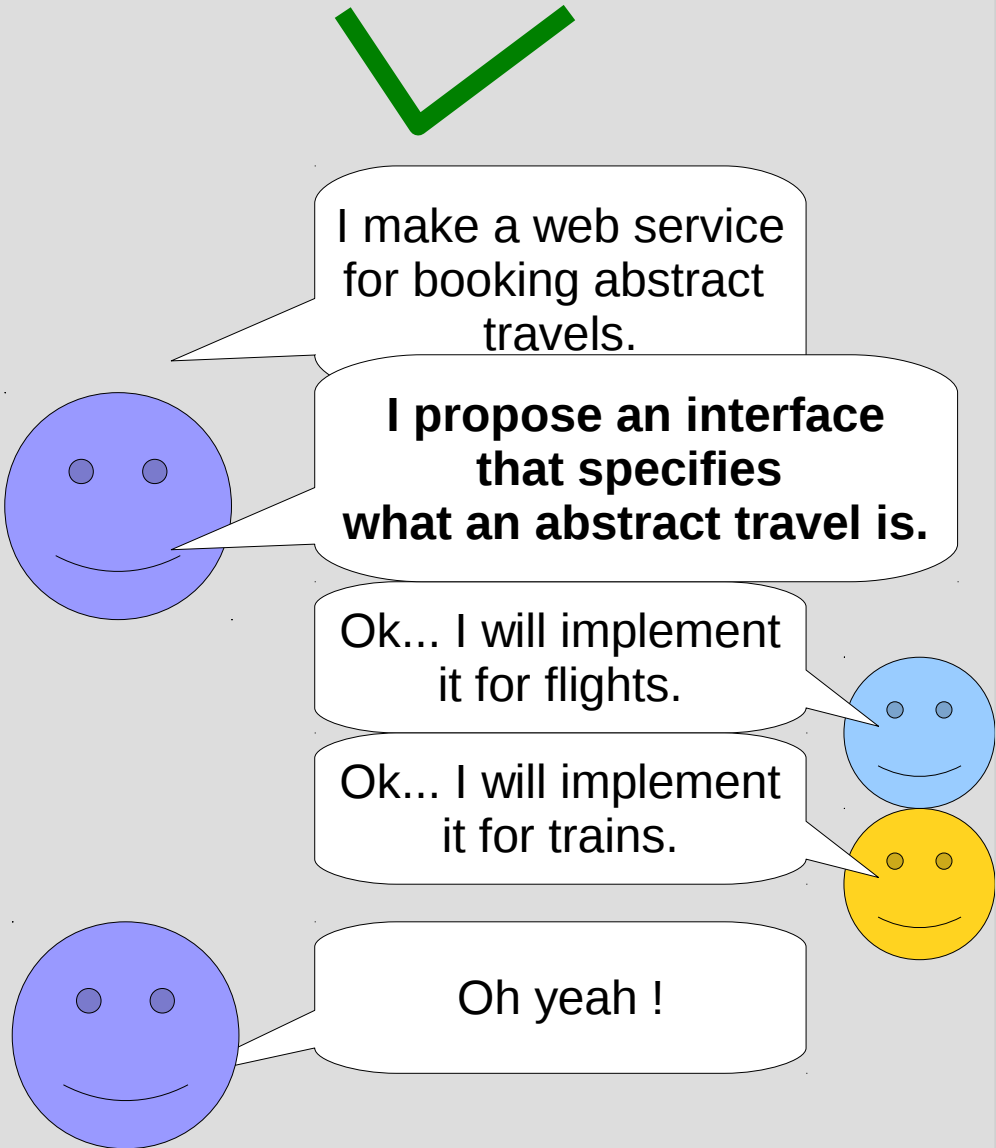
I : Interface Segregation Principle





D: Dependency Inversion Principle

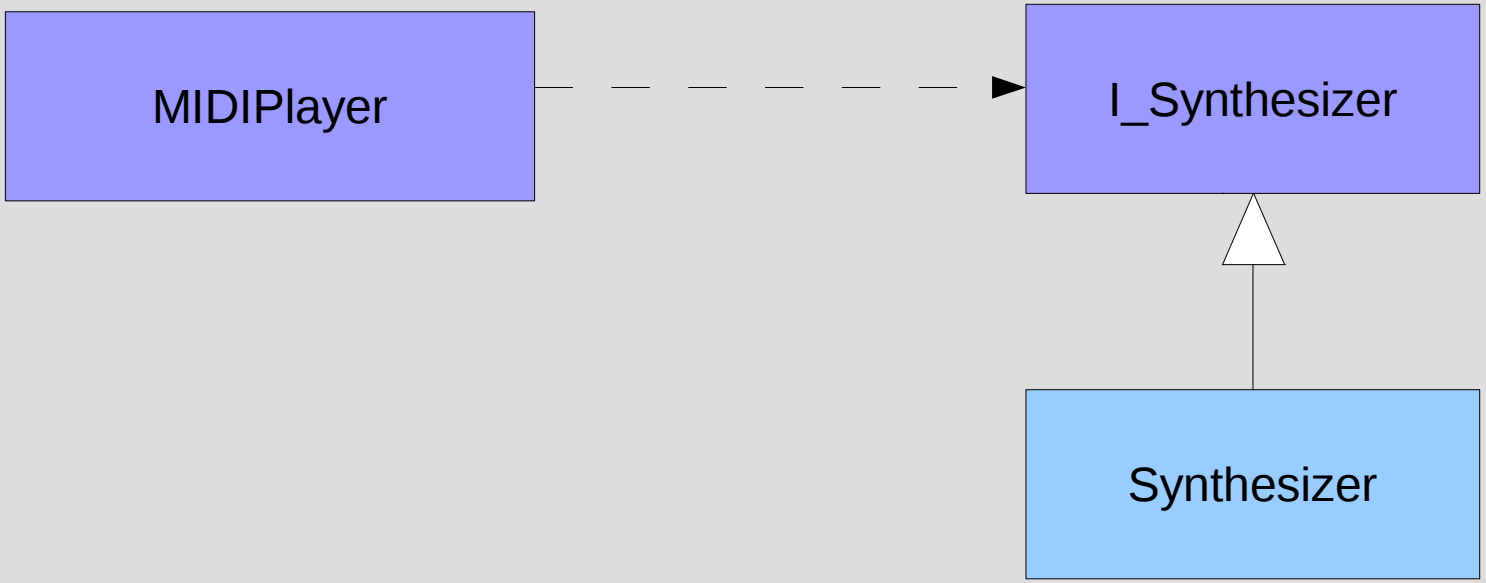




D: Dependency Inversion Principle

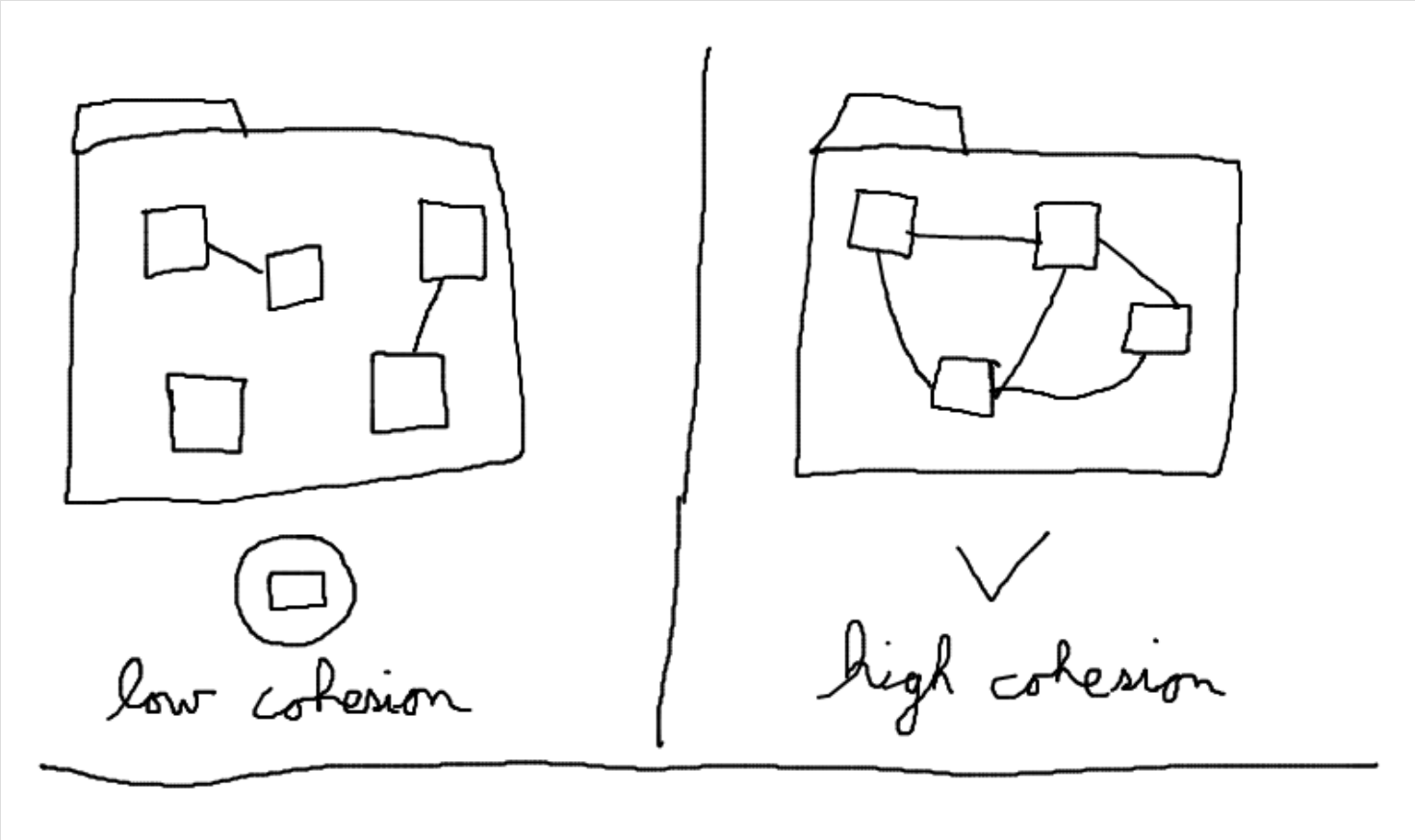
Example:

- JAVA MidiSound



Principles of Package Architecture

Package
Cohesion Principles



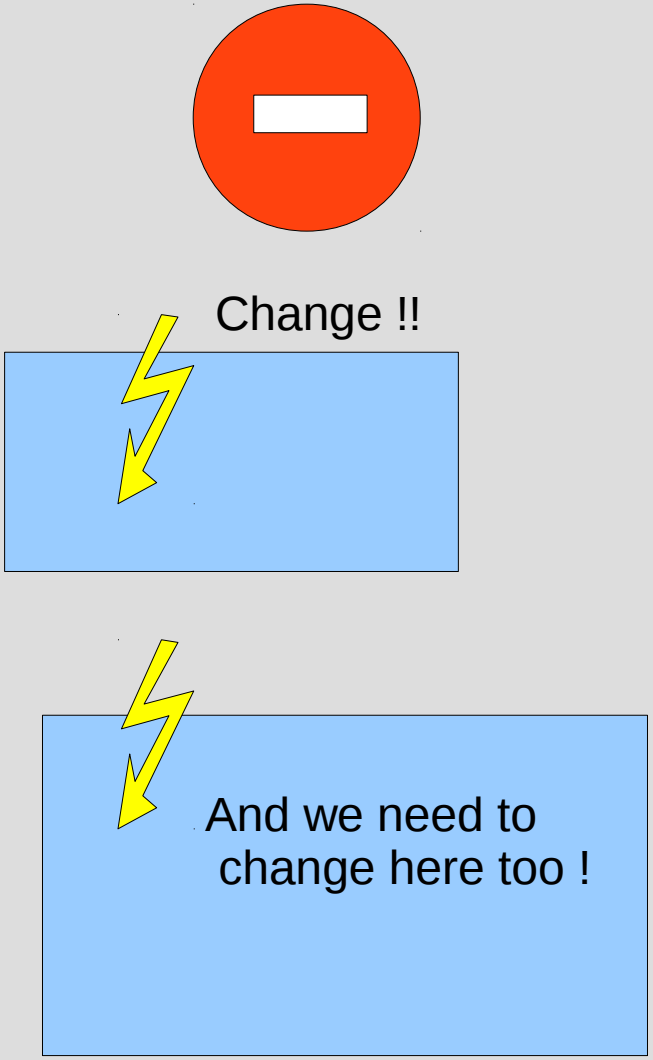
Remark

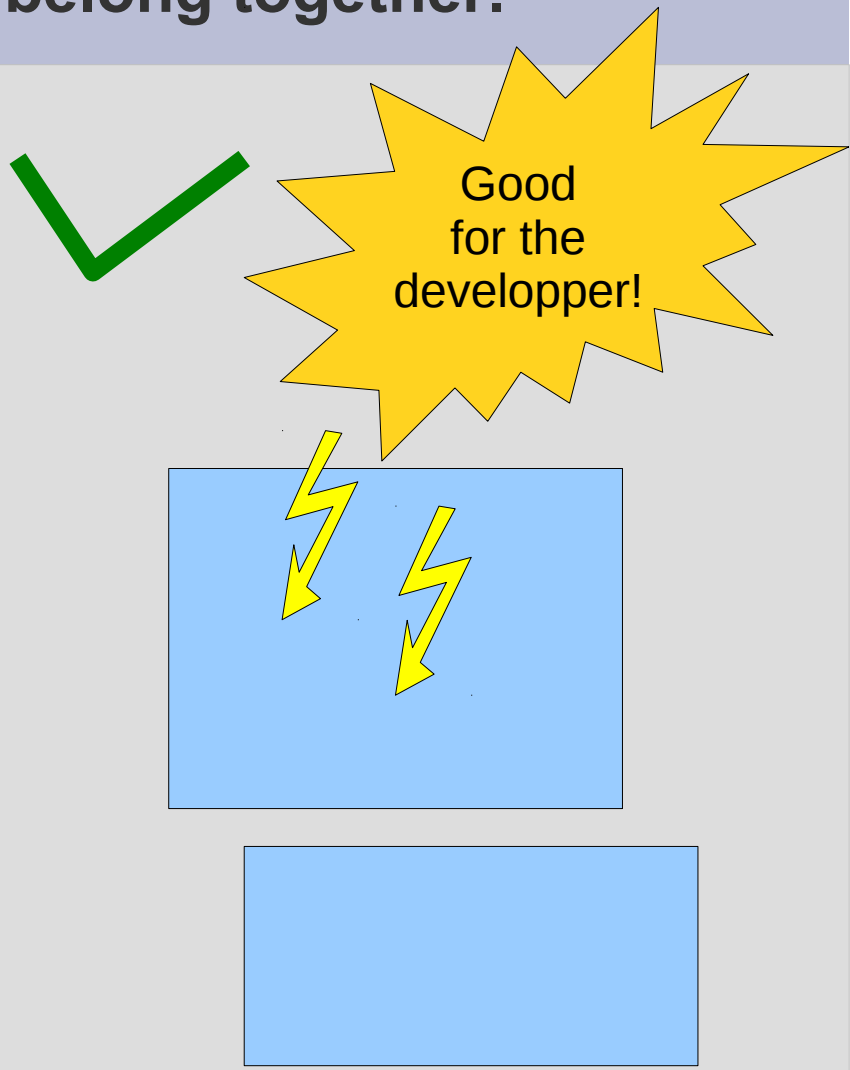
We refactor the packages during the development:

- At the begining stage, we favor the developper.
- At the end, we favor the clients.

The Common Closure Principle

Classes that change together, belong together.





Packages tend to be large.

The Release Reuse Equivalency Principle

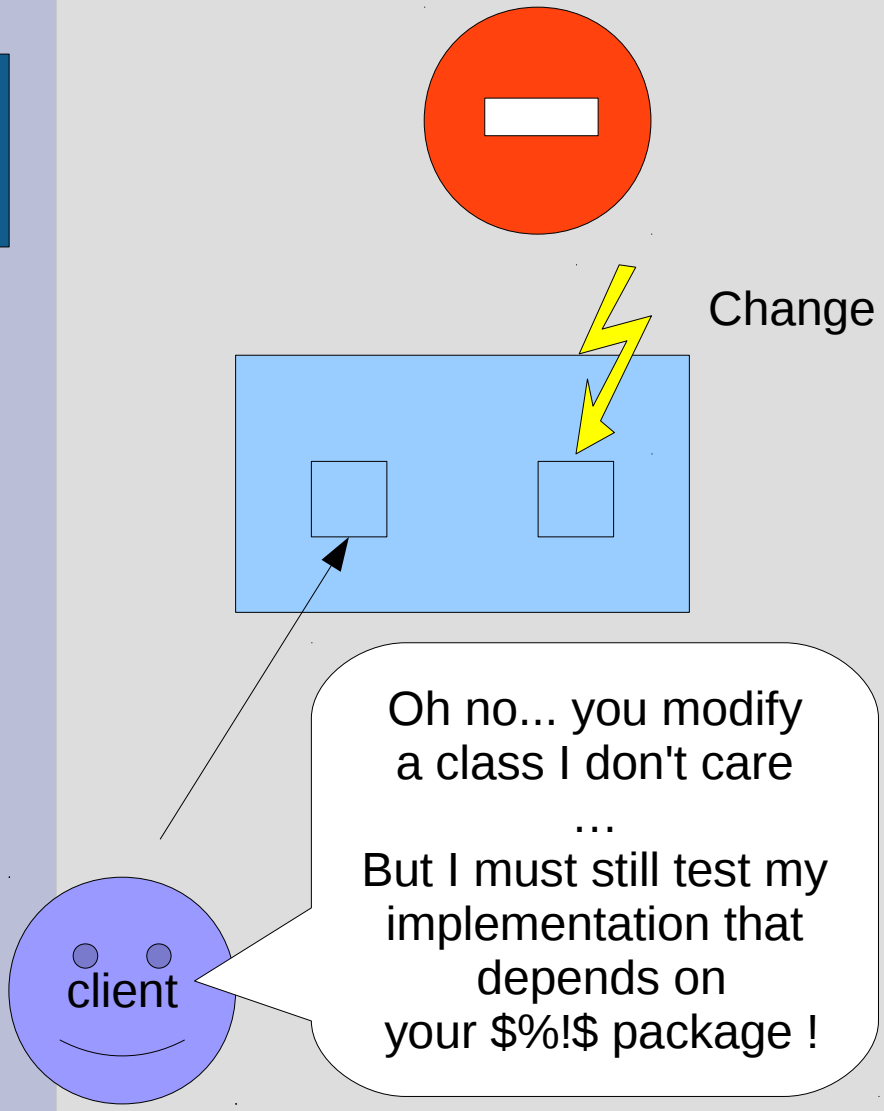
A package

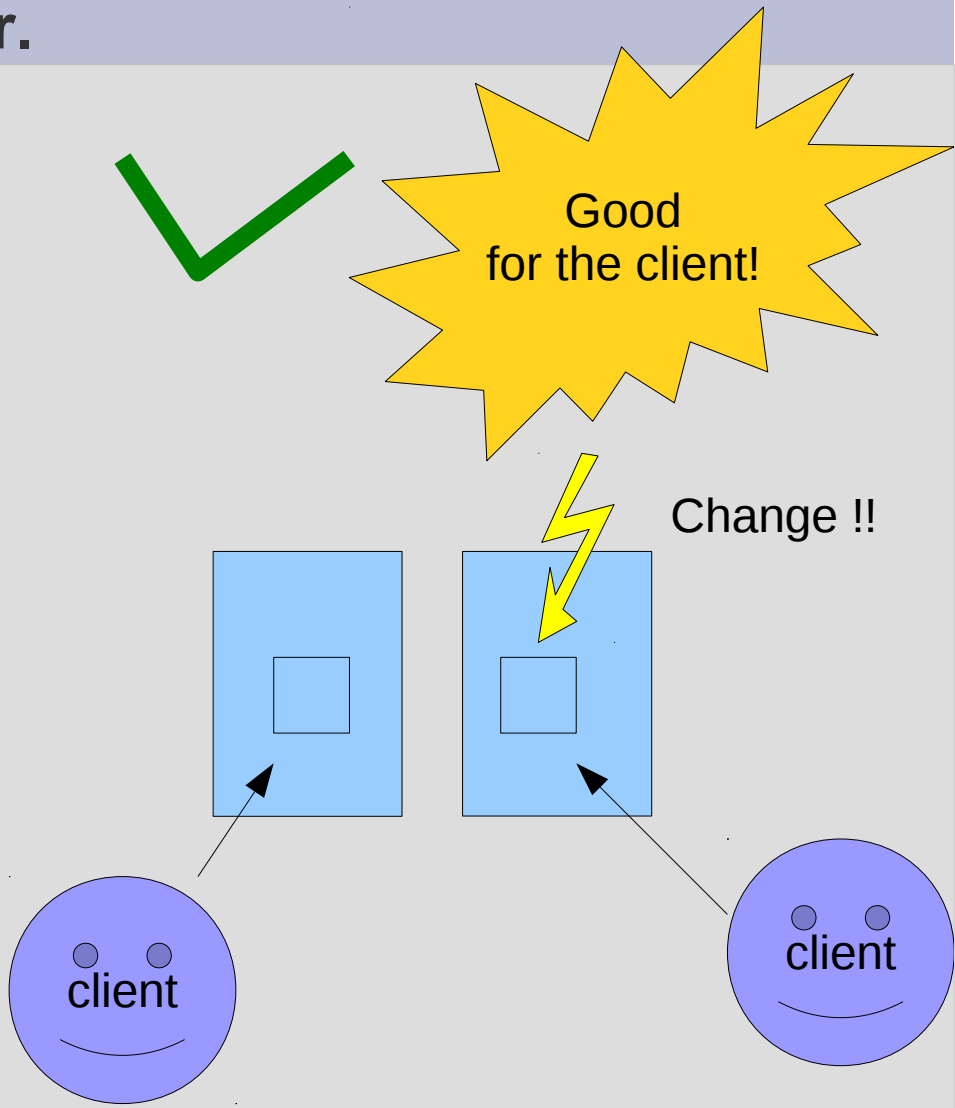
- the granule of reuse
- The granule of release
- Number of versions
- Should support and maintain older versions

Packages tend to be small.

The Common Reuse Principle

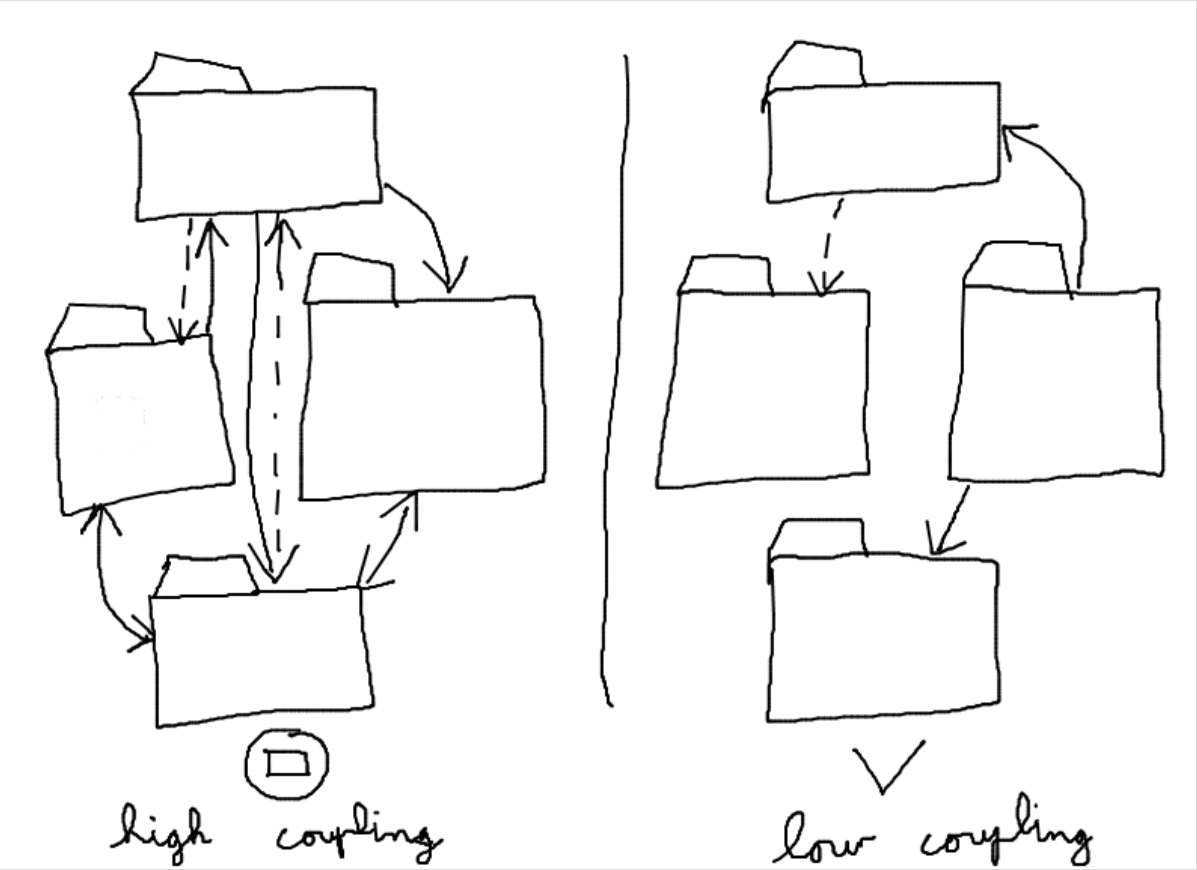
Classes that aren’t reused together should not be grouped together.





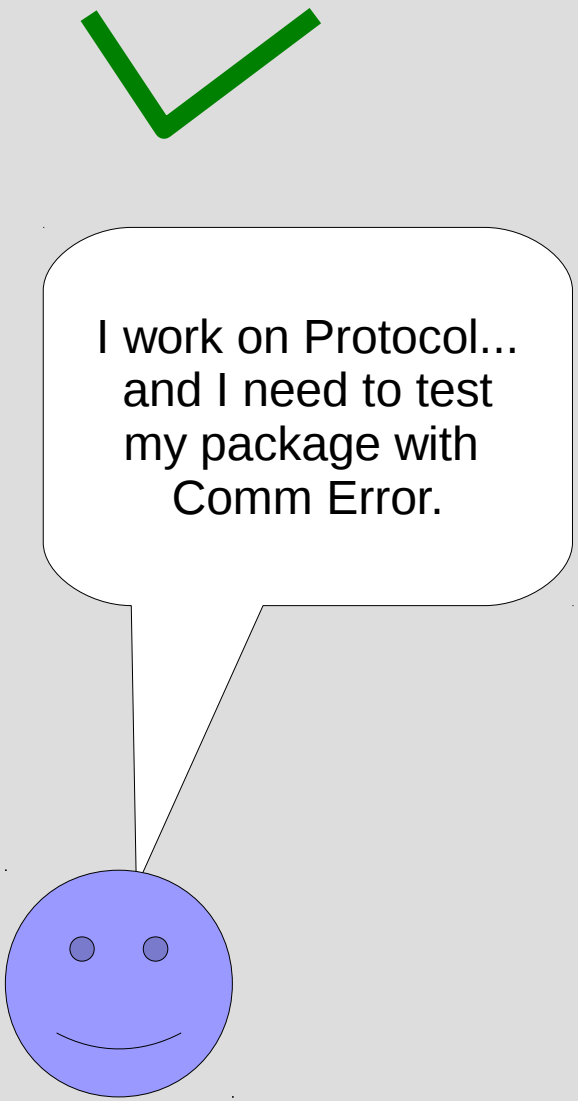
Packages tend to be small.

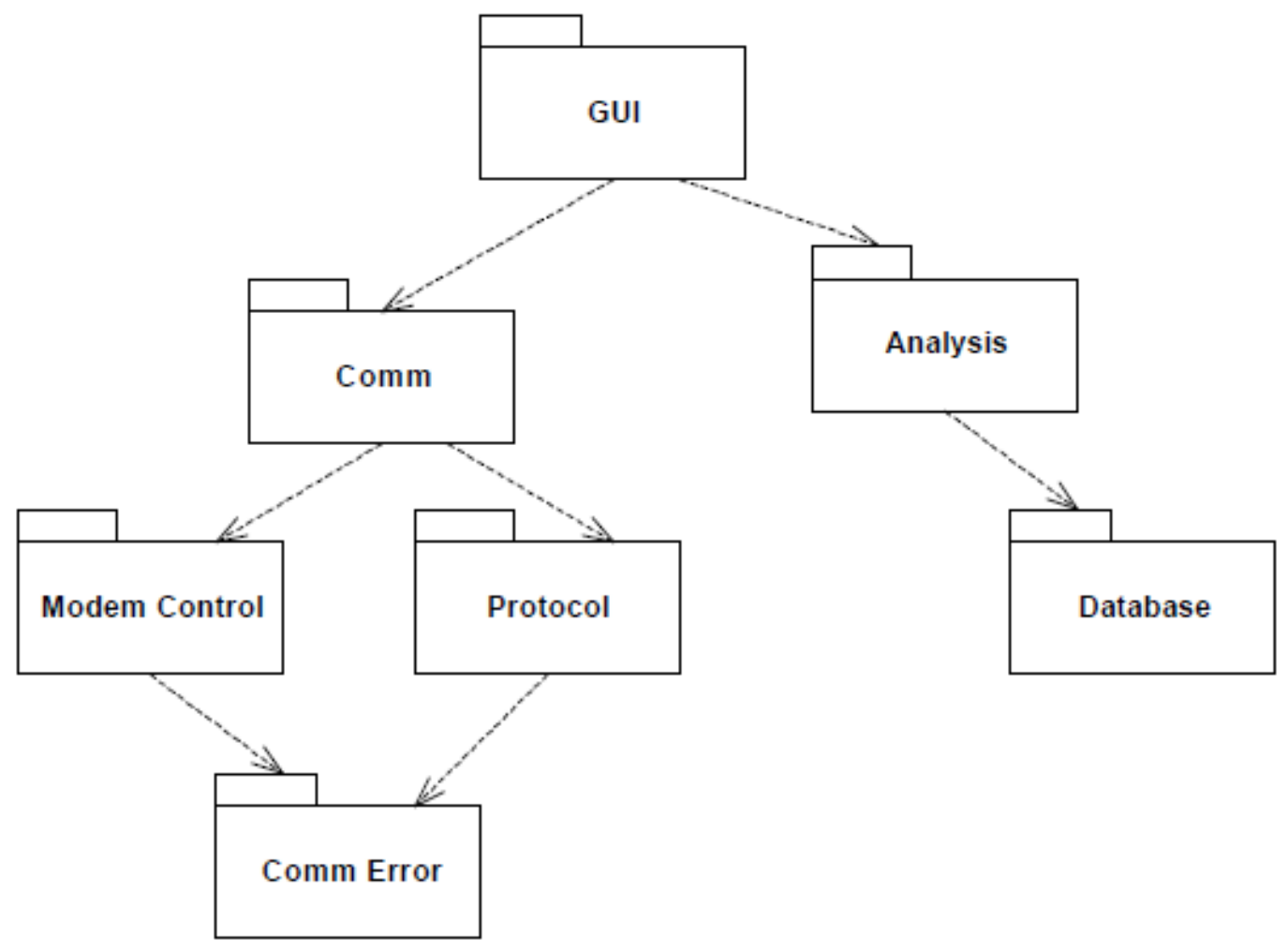
The Package Coupling Principles.



The Acyclic Dependencies Principle

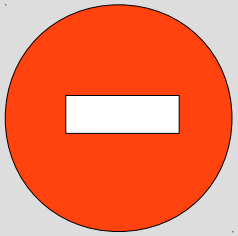
The dependencies betwen packages must not form cycles.



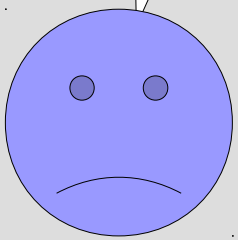


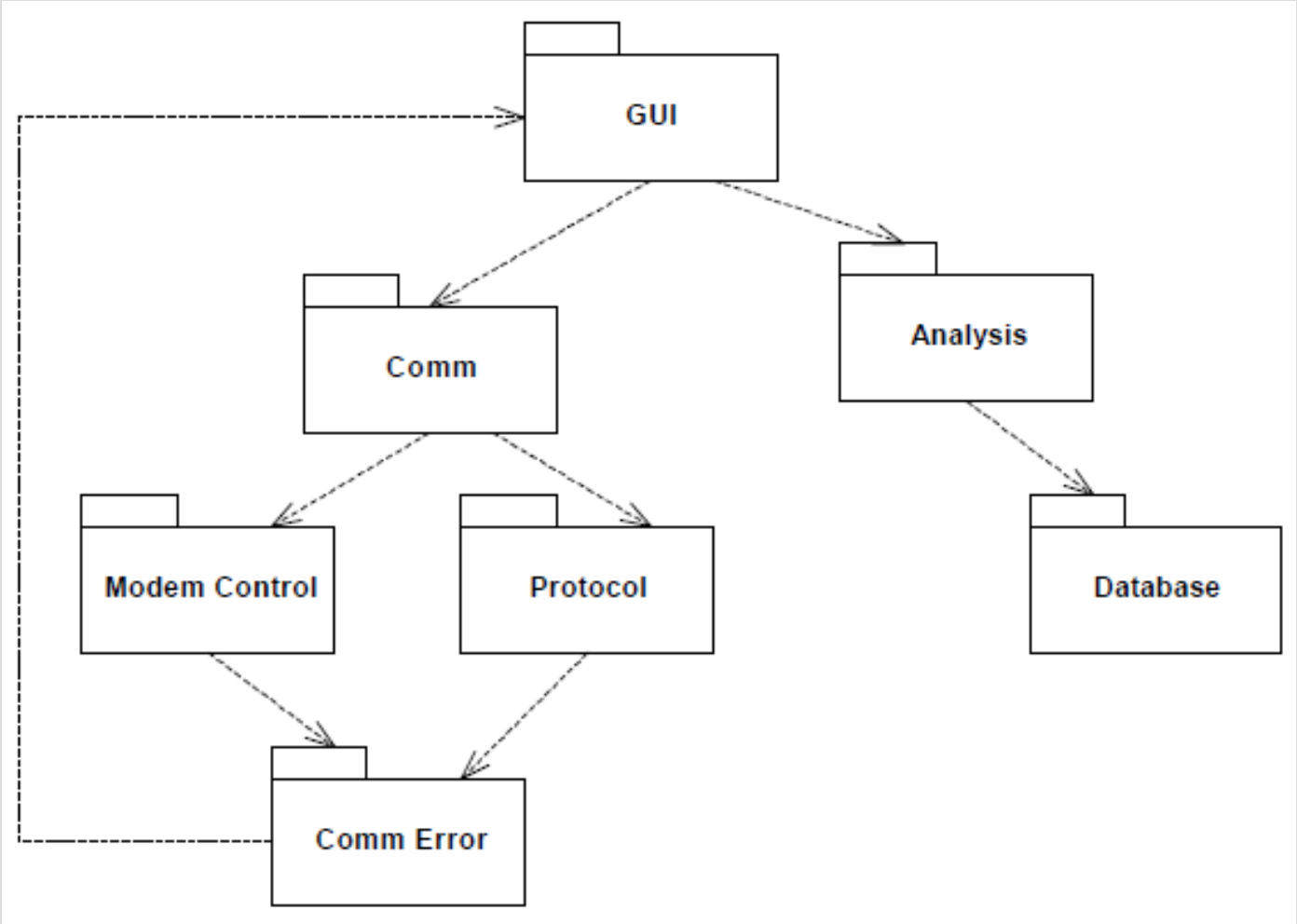
The Acyclic Dependencies Principle

The dependencies betwen packages must not form cycles.

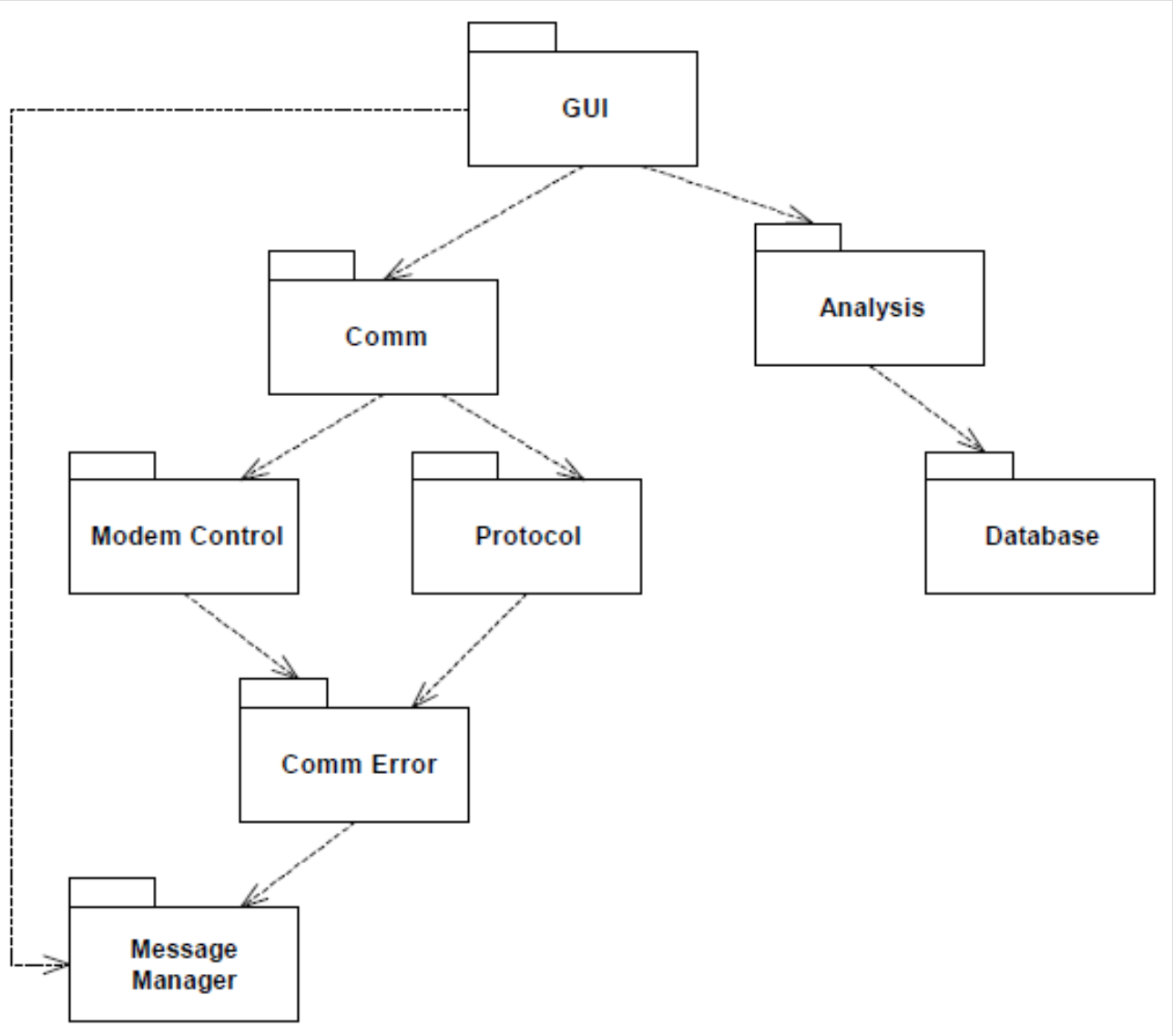


I work on Protocol... and I need to test my package with **all** the packages !



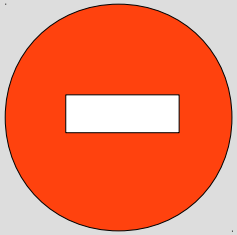


Solution: Dependency Inversion Principle



The Stable Dependencies Principle

Depend in the direction of stability



My work depends on package X !

I need to modify X.. because it is related to other packages... because it is a difficult part of the project...

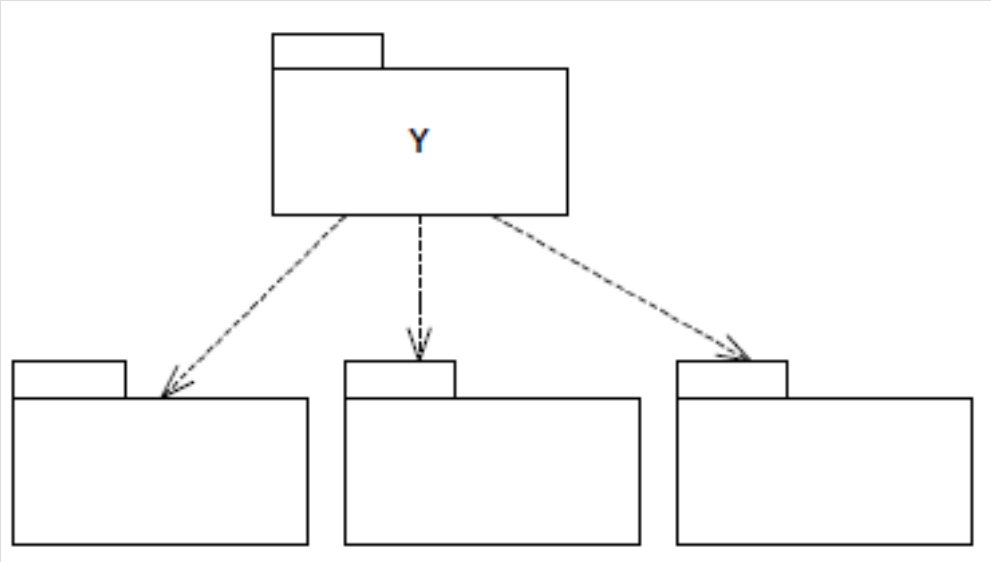
Oh no... X is not stable...

My work depends on package X !

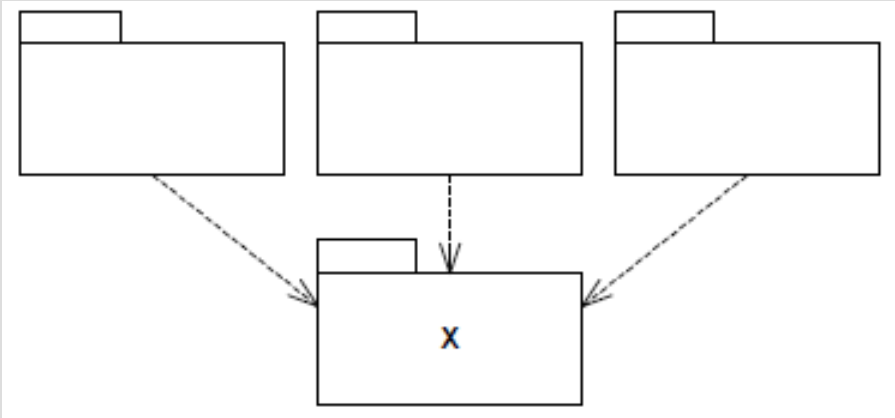
Good point. X will not Change anymore.

X is stable !

Stable / instable



Y instable

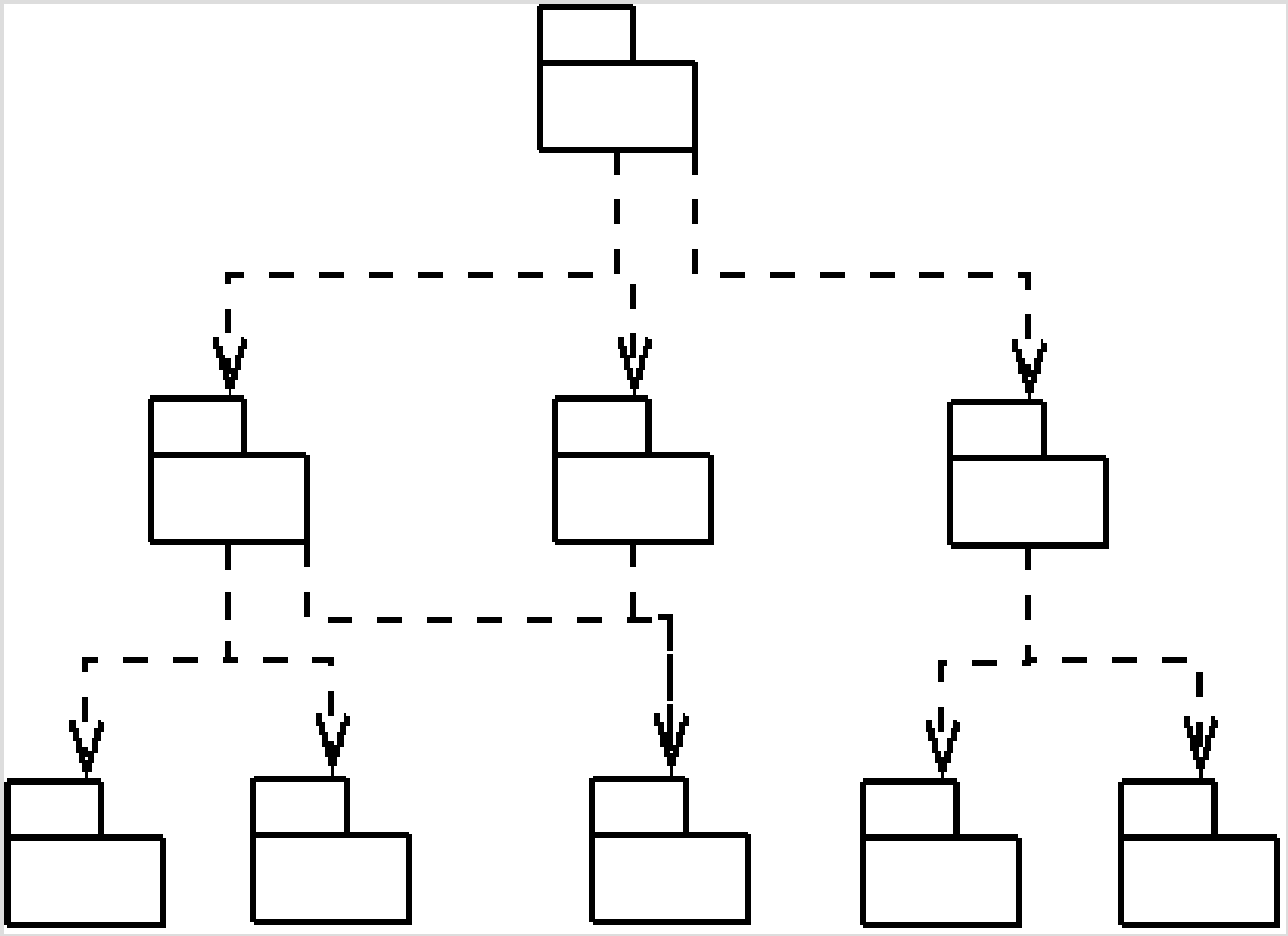


X stable

The stable abstractions principle

Stable packages should be abstract packages.

Flexible / instable

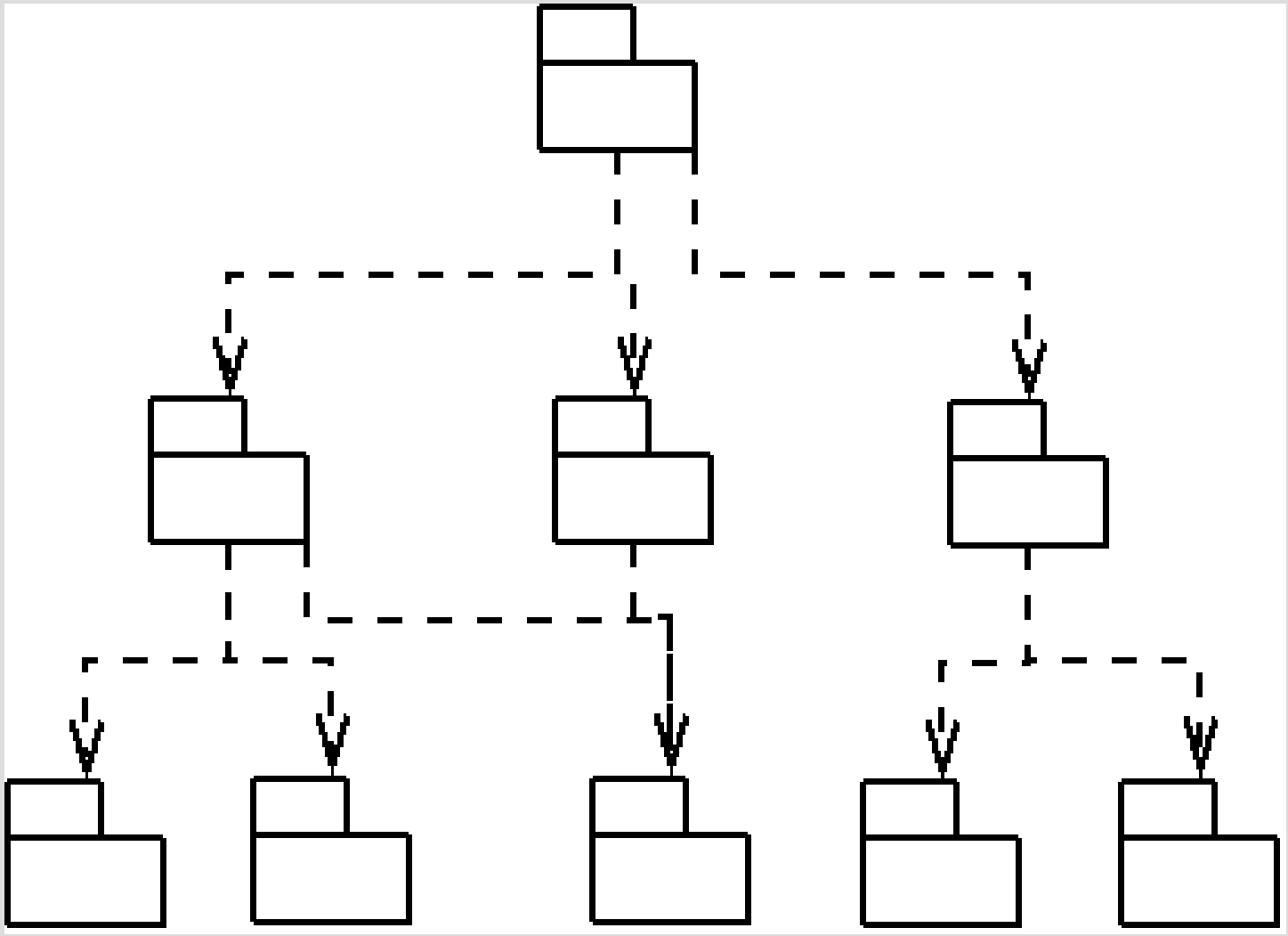


Stable

The stable abstractions principle

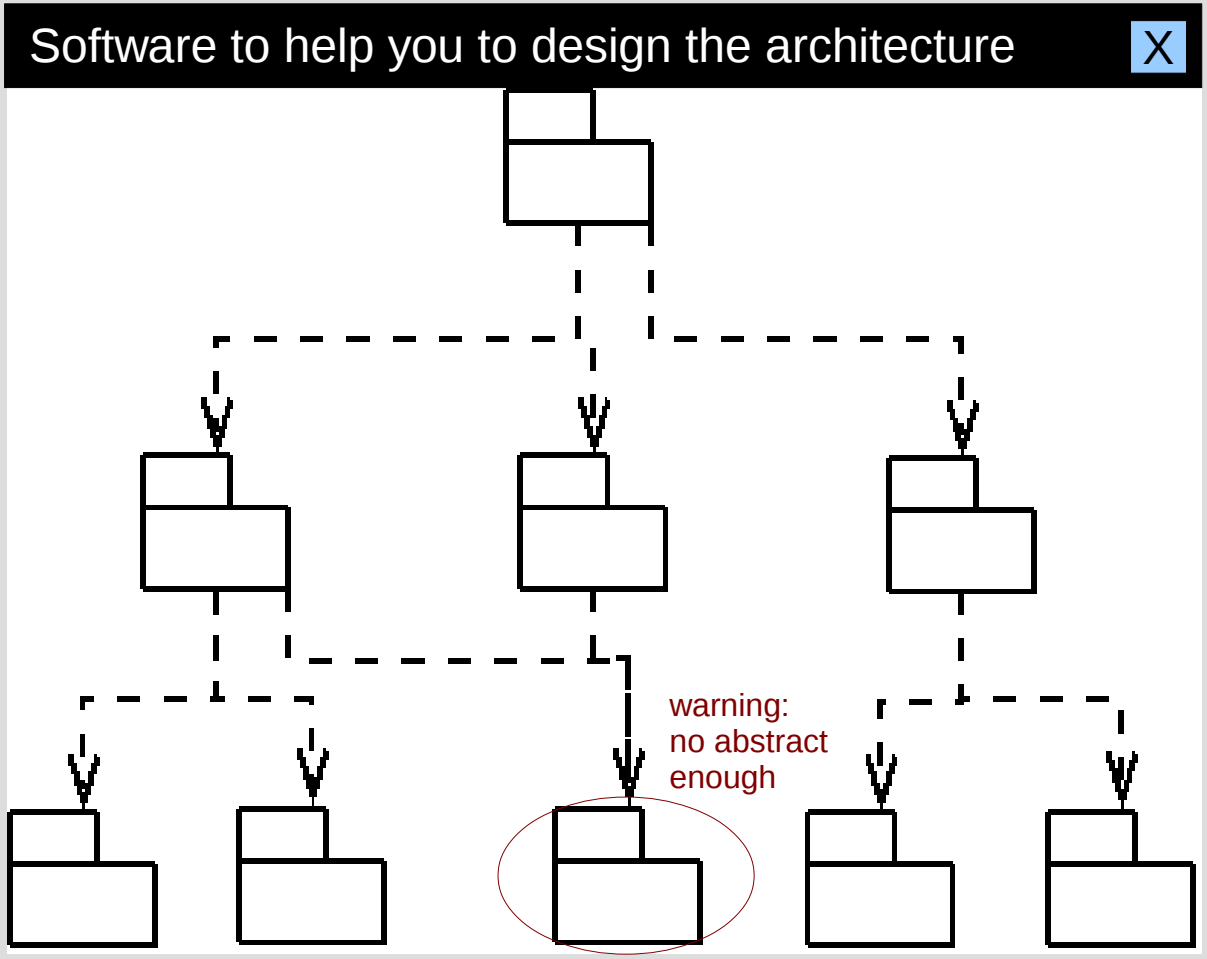
Stable packages should be abstract packages.

Flexible / instable



Stable
→ but we want them flexible
→ should be abstract in order to be extended!

Dream 1: Automated assistance



Measuring instability

Instability:

$$I_{\mathfrak{P}} = \frac{o_{\mathfrak{P}}}{i_{\mathfrak{P}} + o_{\mathfrak{P}}}$$

where

- $o_{\mathfrak{P}}$ (outgoing dependencies) is the number classes outside $\mathfrak{P}$ classes inside $\mathfrak{P}$ depend on;
- $i_{\mathfrak{P}}$ (incoming dependencies) is the number classes outside $\mathfrak{P}$ that depend on classes inside.

Measuring abstractness

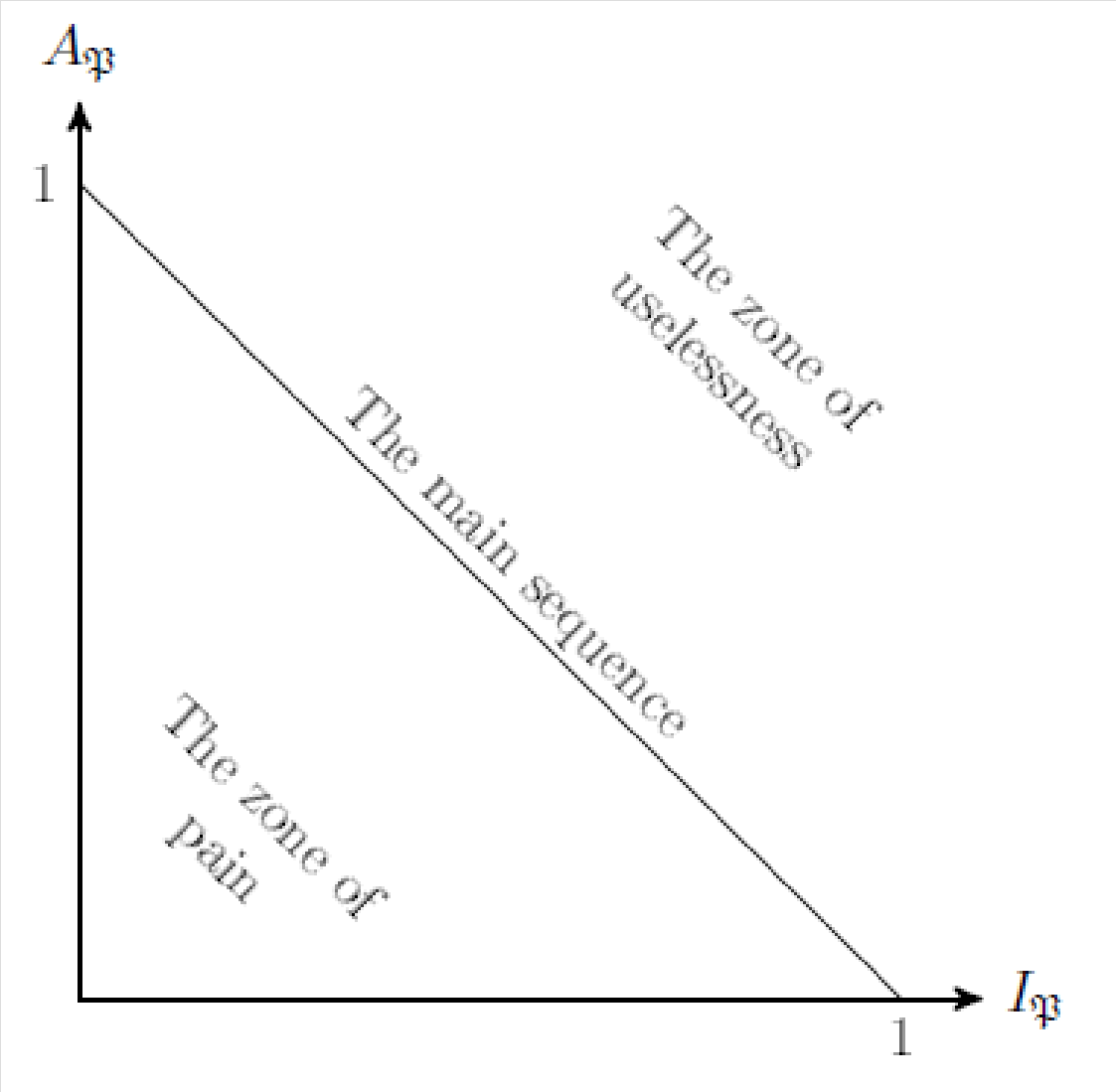
Abstractness:

$$A_{\mathfrak{P}} = \frac{a_{\mathfrak{P}}}{card(\mathfrak{P})}$$

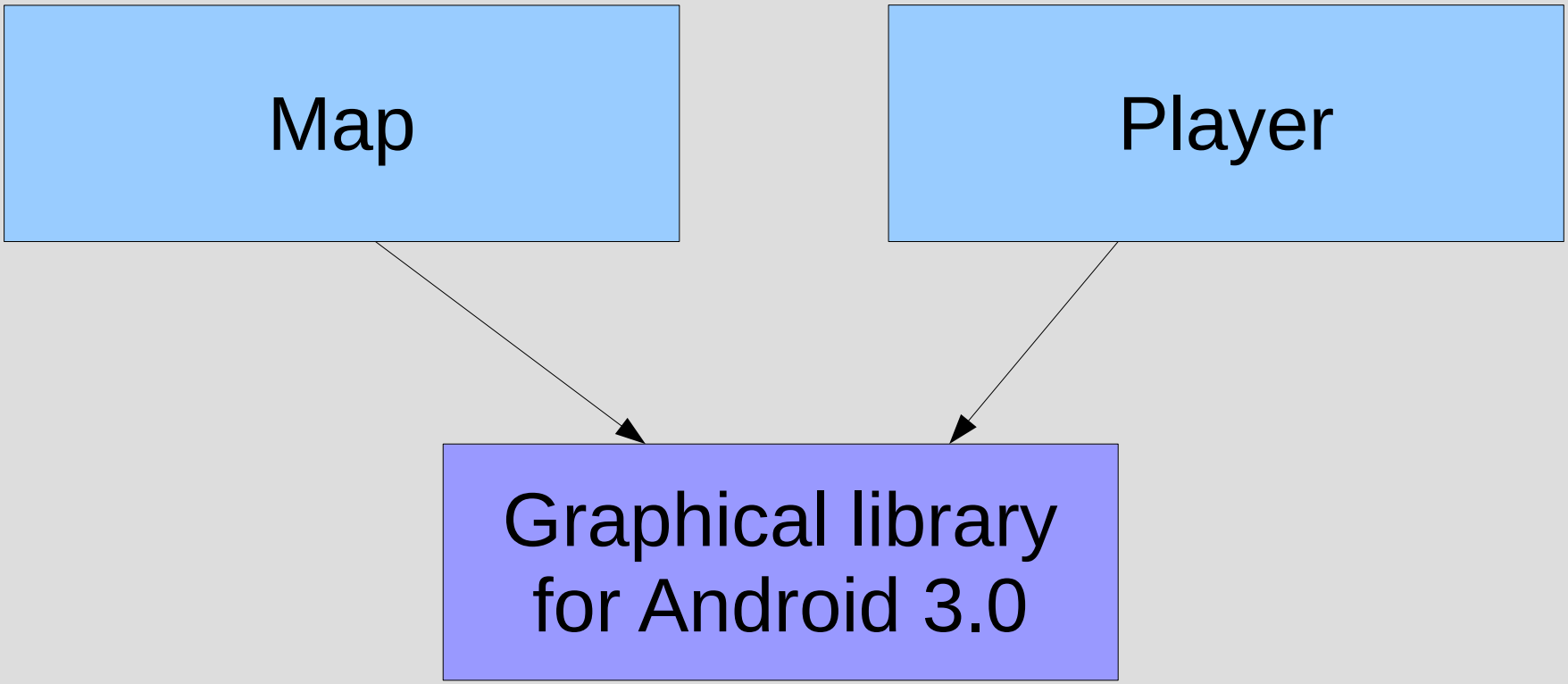
where where

- $a_{\mathfrak{P}}$ is the number of abstract classes in $\mathfrak{P}$;
- $card(\mathfrak{P})$ is the cardinality of $\mathfrak{P}$, that is the number of classes in $\mathfrak{P}$.

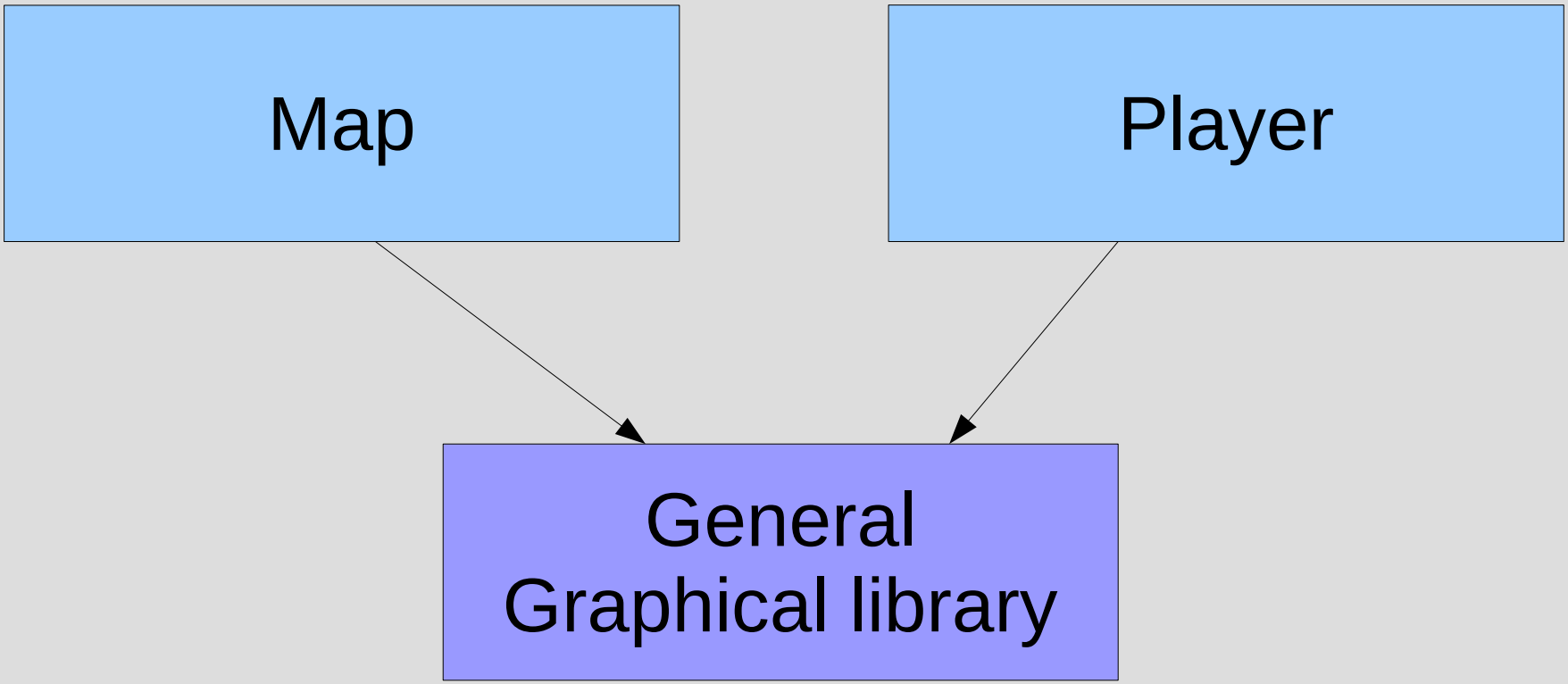
Instability VS Abstractness



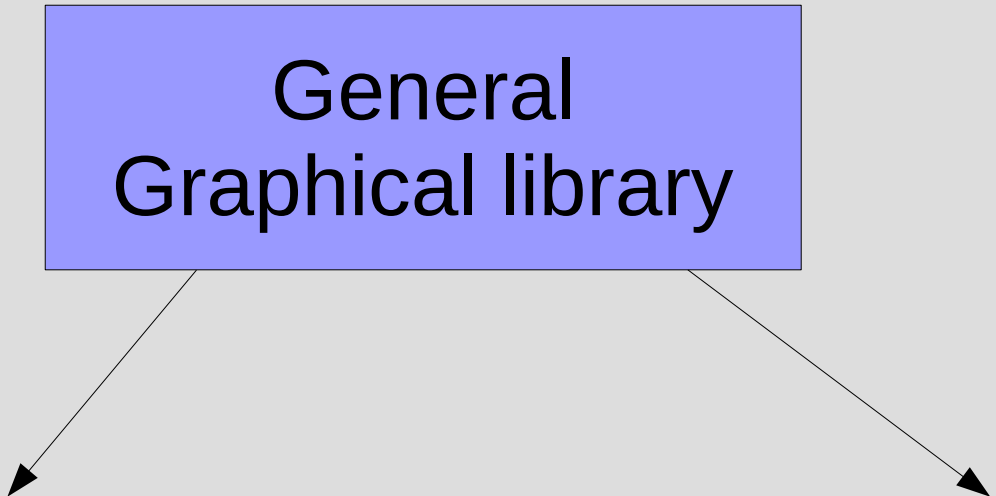
The zone of pain



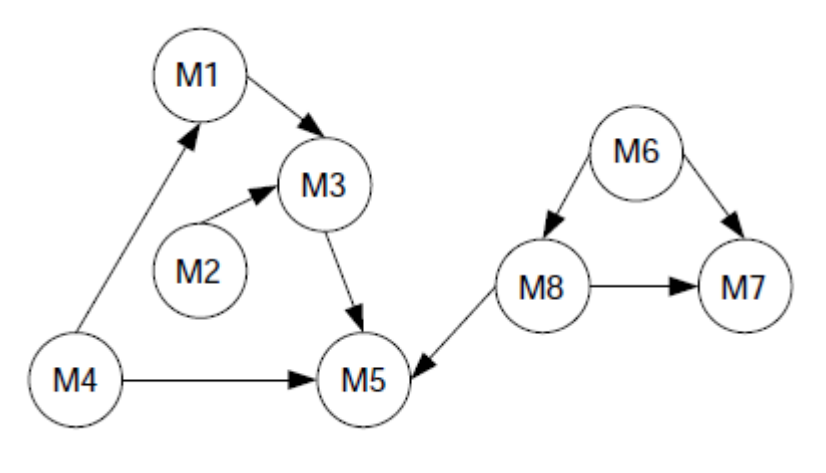
The main sequence



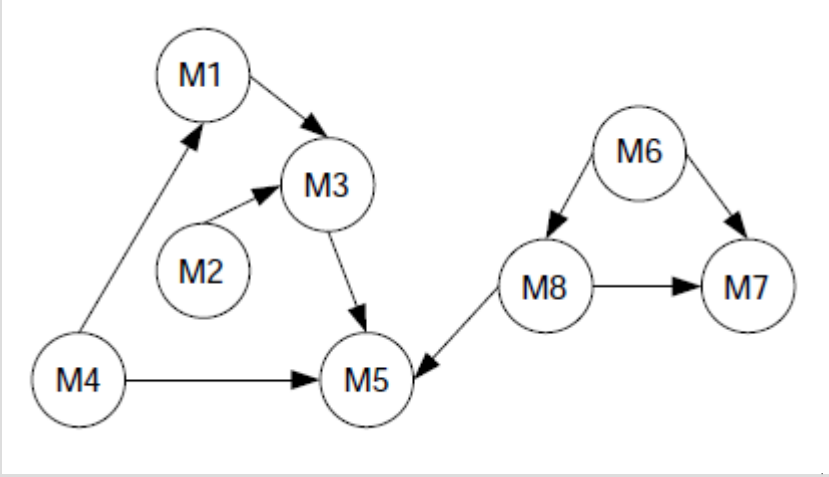
The zone of uselessness

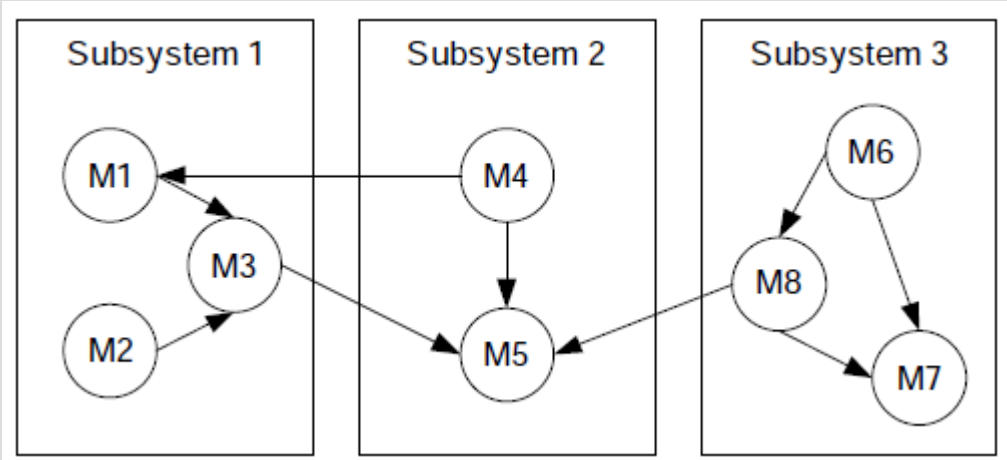


Dream 2: creating automatically the packages partition



Dream 2: creating automatically the packages partition





Dream 2: creating automatically the packages partition

A new field

- [Mitchell 2002]
- Bunch [Mitchell et al. 2006]

- Nothing about stability and abstractness
- Preliminary work...

Related problems

P:

- Minimal cut by flow algorithms

  = finding two packages with low coupling

NP:

- Graph partitioning (minimal cut plus a constraint over the size of the packages)

  = finding two `big' packages with low coupling

- The clique problem, NP-complete

  = find a package with high cohesion

Mitchell's PhD

- Measuring cohesion

$$A_{\mathfrak{P}} = \frac{card(E \cap \mathfrak{P} \times \mathfrak{P})}{card(\mathfrak{P})^2}$$

- Measuring coupling

$$E_{\mathfrak{P},\mathfrak{P}'} = \begin{cases} 0 \text{ if } \mathfrak{P} = \mathfrak{P}' \\ \frac{card(E \cap \mathfrak{P} \times \mathfrak{P}') + card(E \cap \mathfrak{P}' \times \mathfrak{P})}{2 card(\mathfrak{P}) card(\mathfrak{P}')} \text{ else} \end{cases}$$

- Measuring the quality of a clustering

$$MQ = \begin{cases} A_{\mathfrak{P}} \text{ if } k = 1 \text{ and } \mathfrak{P} \text{ is the single package} \\ \frac{1}{k} \sum_{\mathfrak{P} \in \mathbb{P}} A_{\mathfrak{P}} - \frac{1}{\frac{k(k-1)}{2}} \sum_{\mathfrak{P},\mathfrak{P}' \in \mathbb{P}} E_{\mathfrak{P},\mathfrak{P}'} \text{if } k > 1 \end{cases}$$

Heuristics

- Hill-climbing algorithms
- Genetic algorithms

PS : People claim the problem is NP-complete (I want a proof)