Object design

François Schwarzentruber
ENS Cachan – Antenne de Bretagne

Outline

- Symptoms of rotting systems
- Principles of object oriented class design
- Principles of Package Architecture
- Dreams

Outline

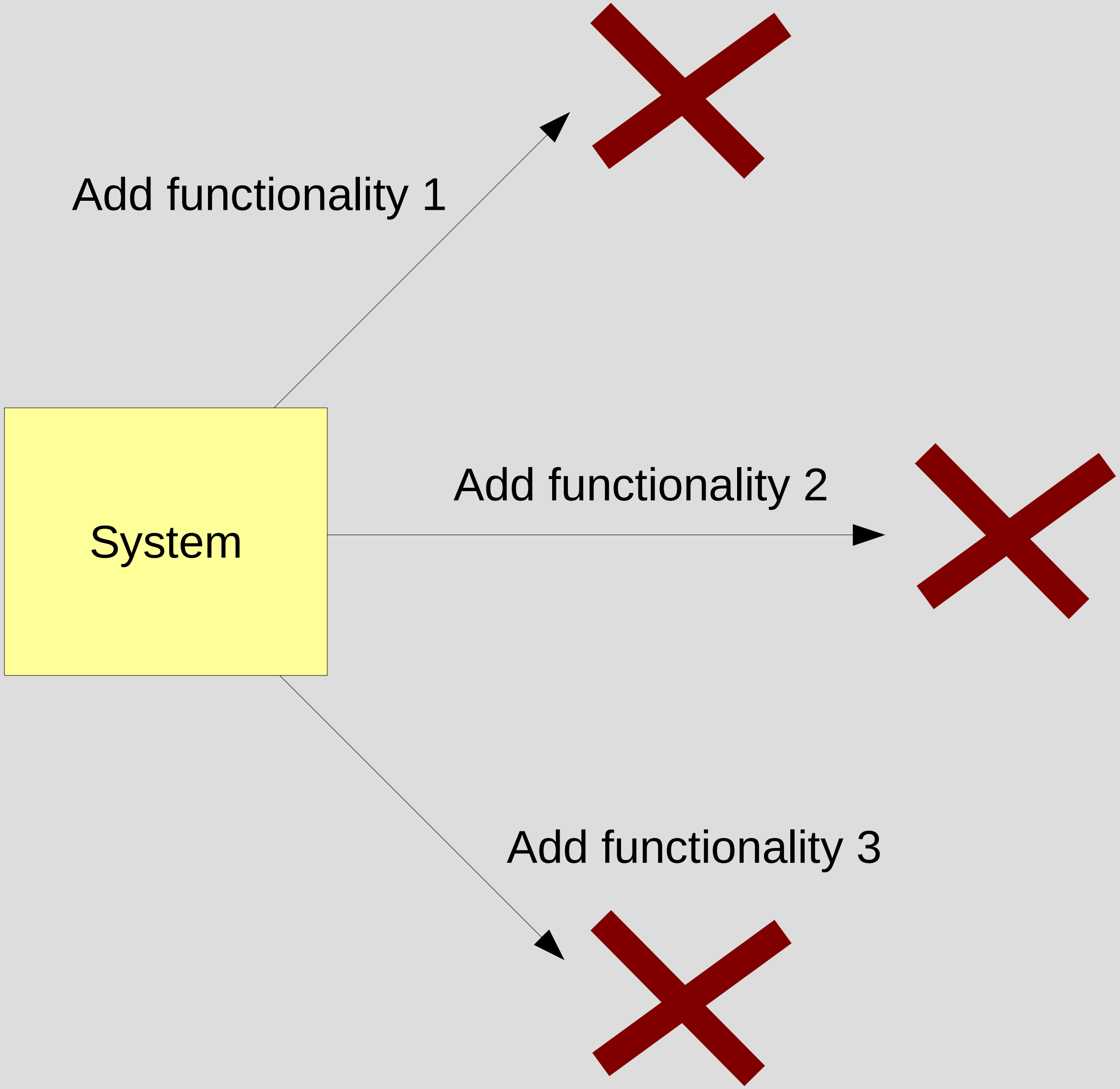
- **Symptoms of rotting systems**
- Principles of object oriented class design
- Principles of Package Architecture
- Dreams

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

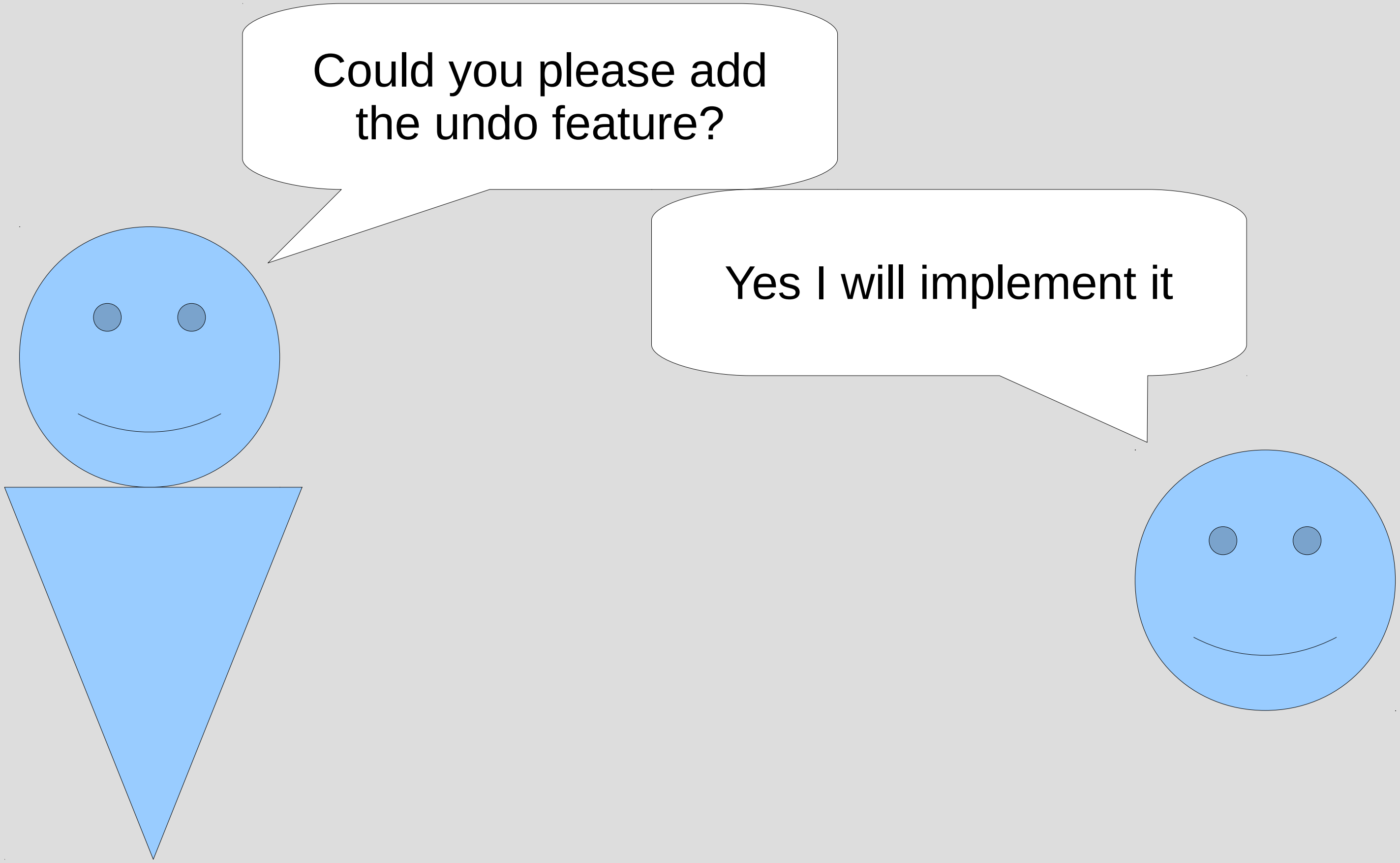
Four possible unsuitable behaviors of the developer team:

- Rigidity
- Fragility
- Immobility
- Viscosity

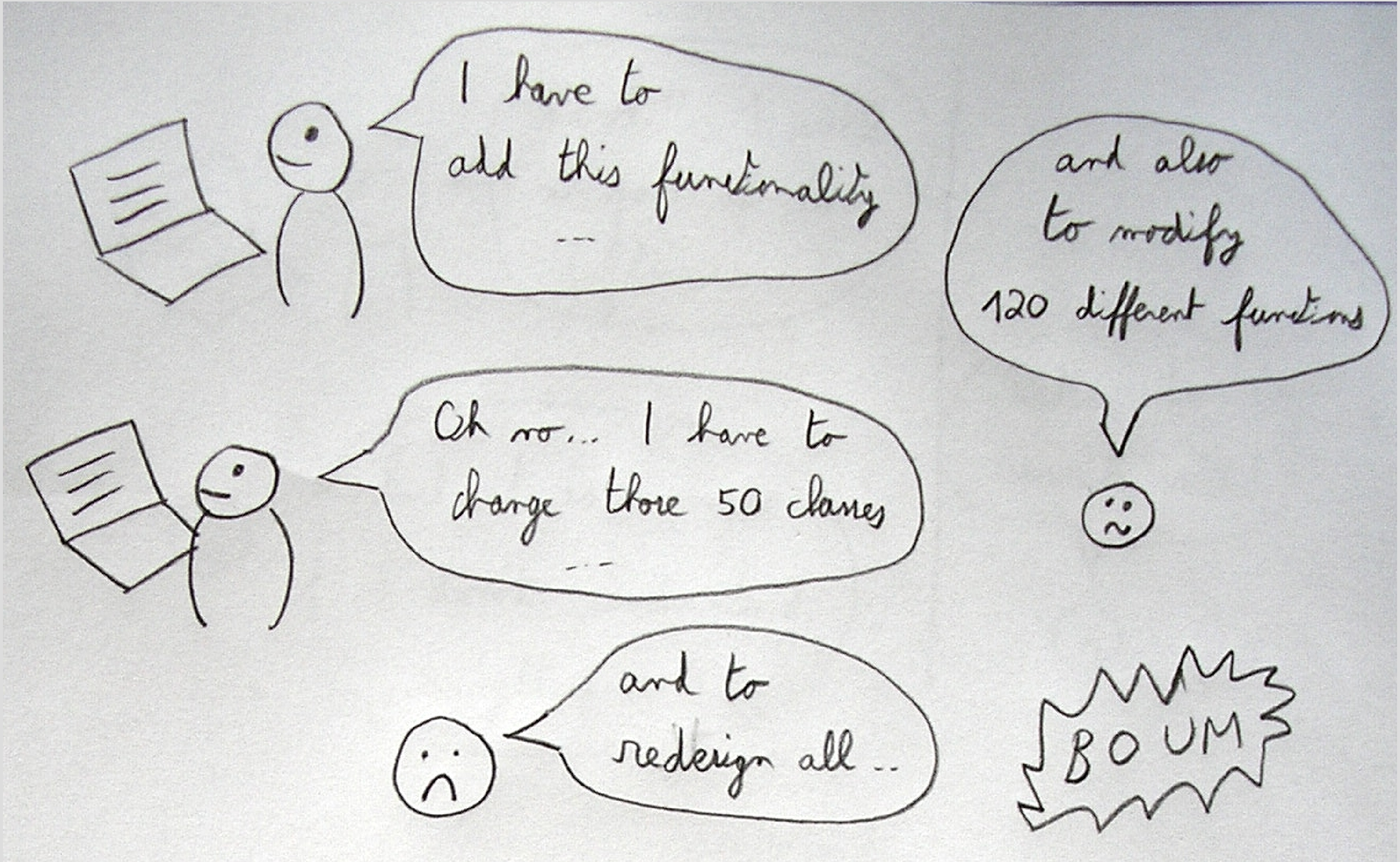
Rigidity


Add functionality 1
System
Add functionality 2
Add functionality 3

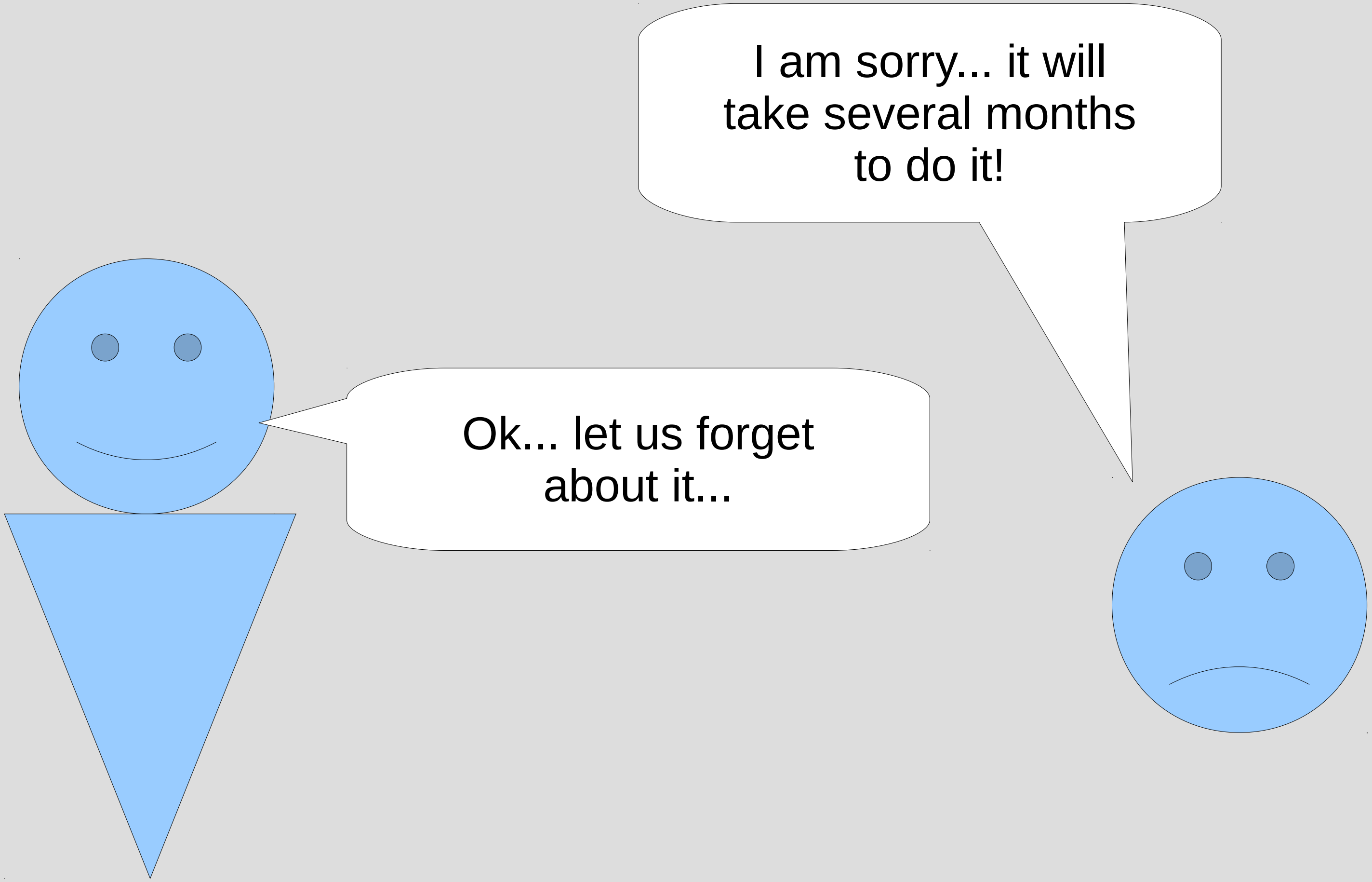
Rigidity


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

Rigidity

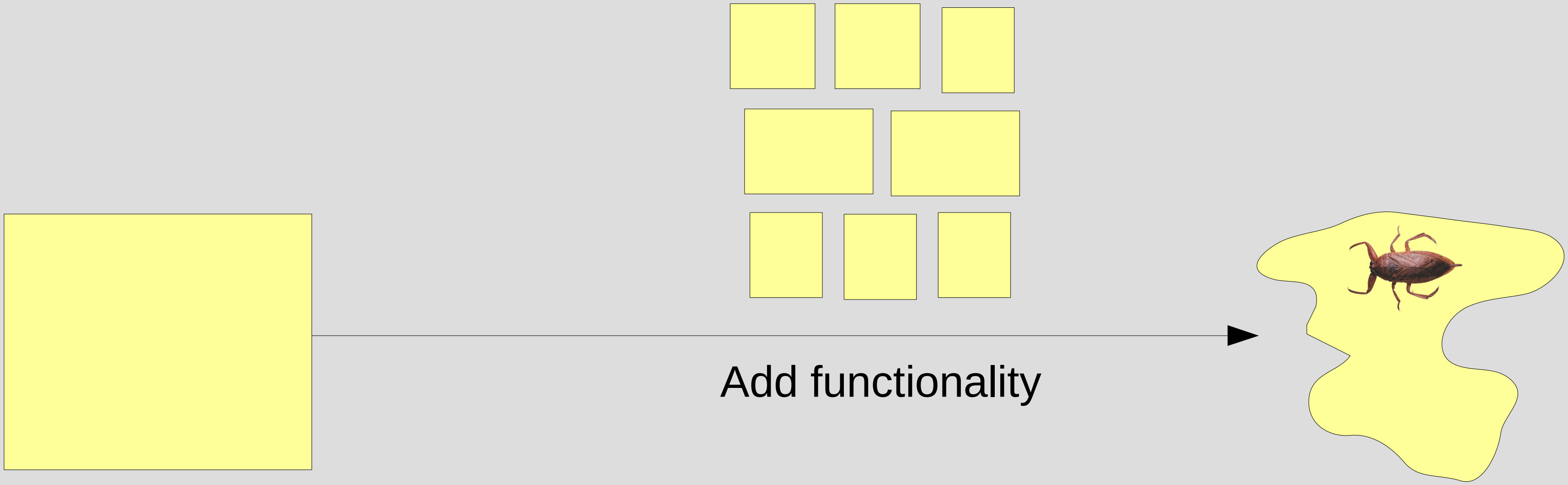

I have to add this functionality ...
and also to modify 120 different functions
Oh no... I have to change those 50 classes ...
and to redesign all ..
BOUM

Rigidity


I am sorry... it will take several months to do it!
Ok... let us forget about it...

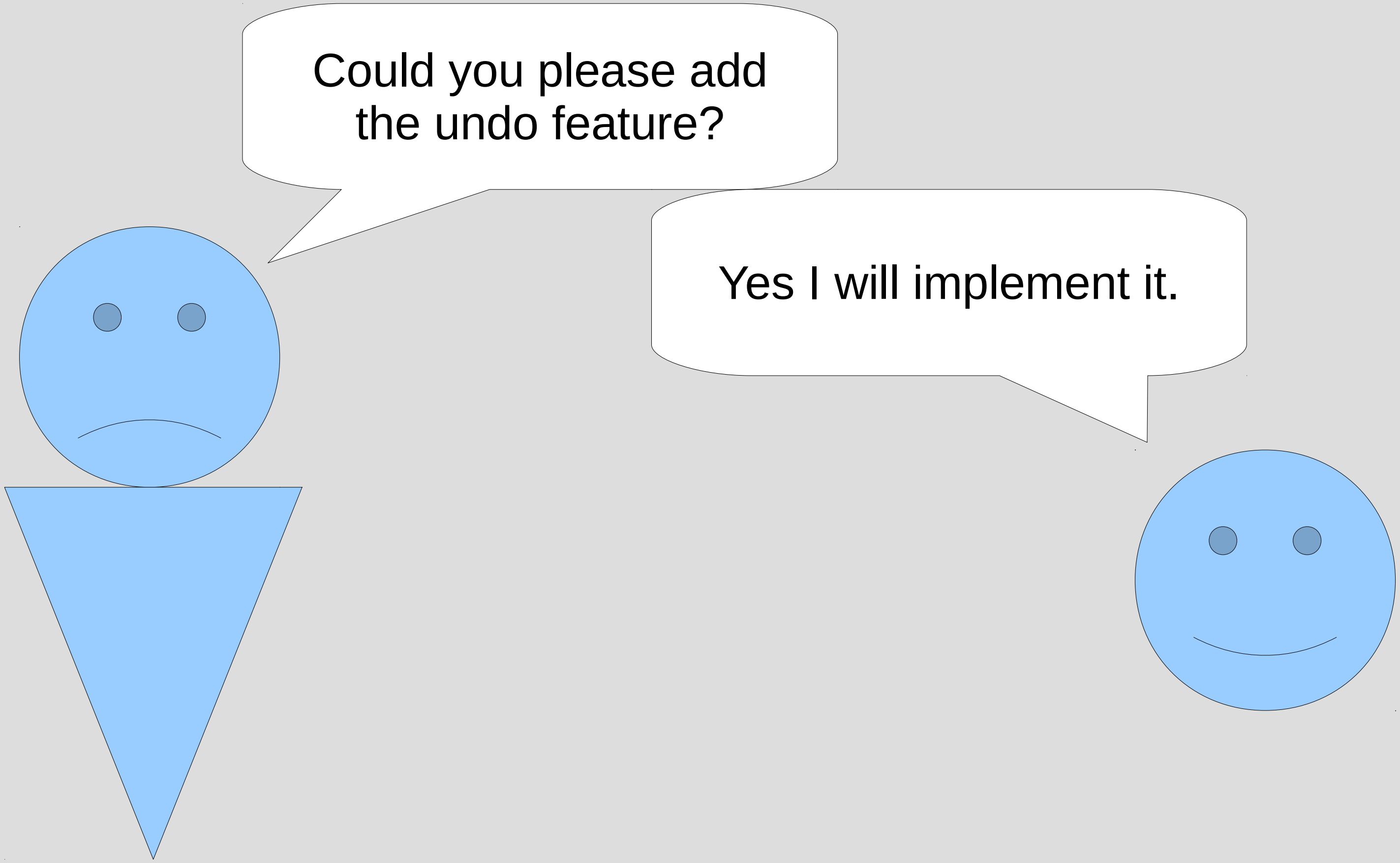
This functionality will never be implemented.

Fragility

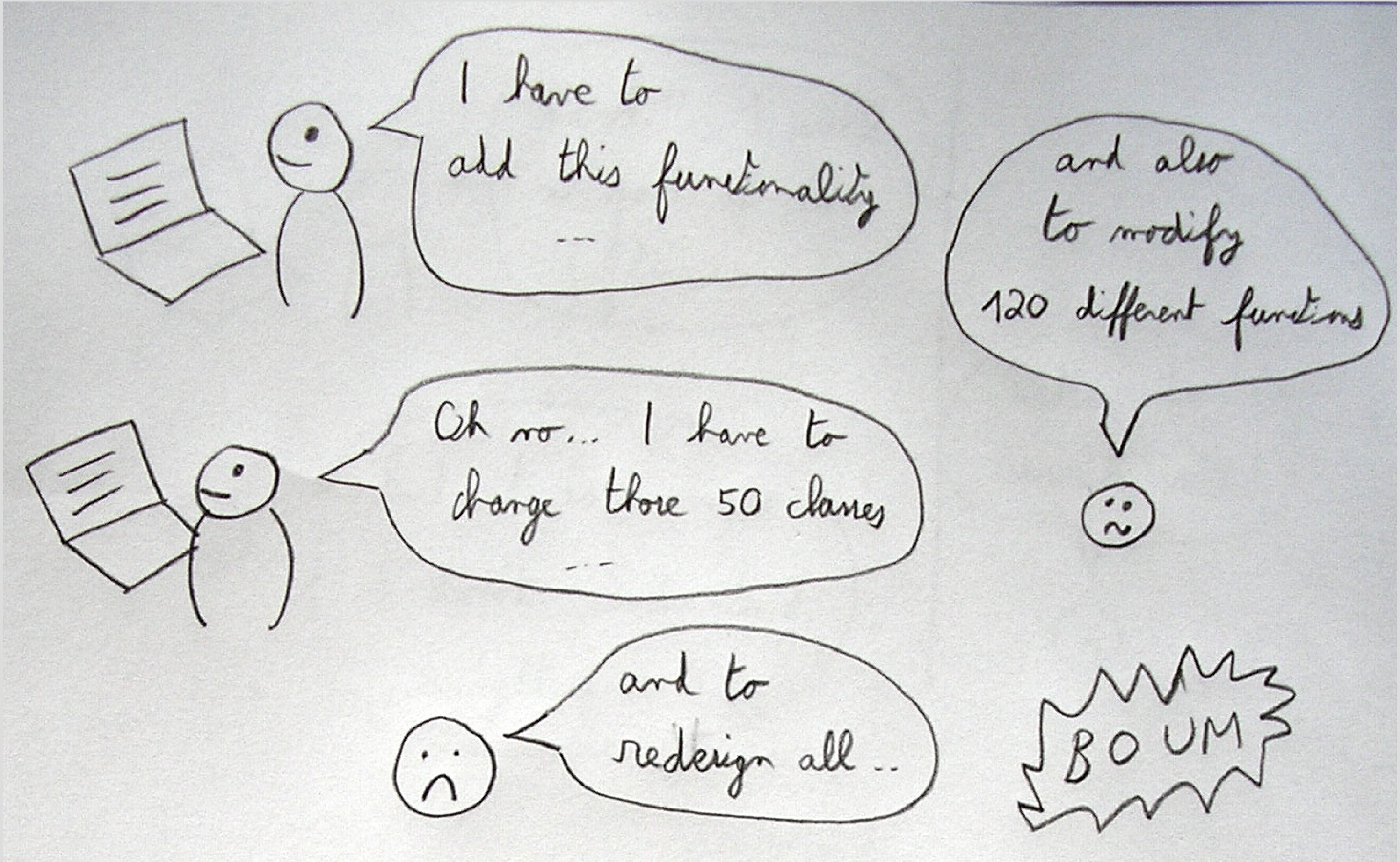


Add functionality

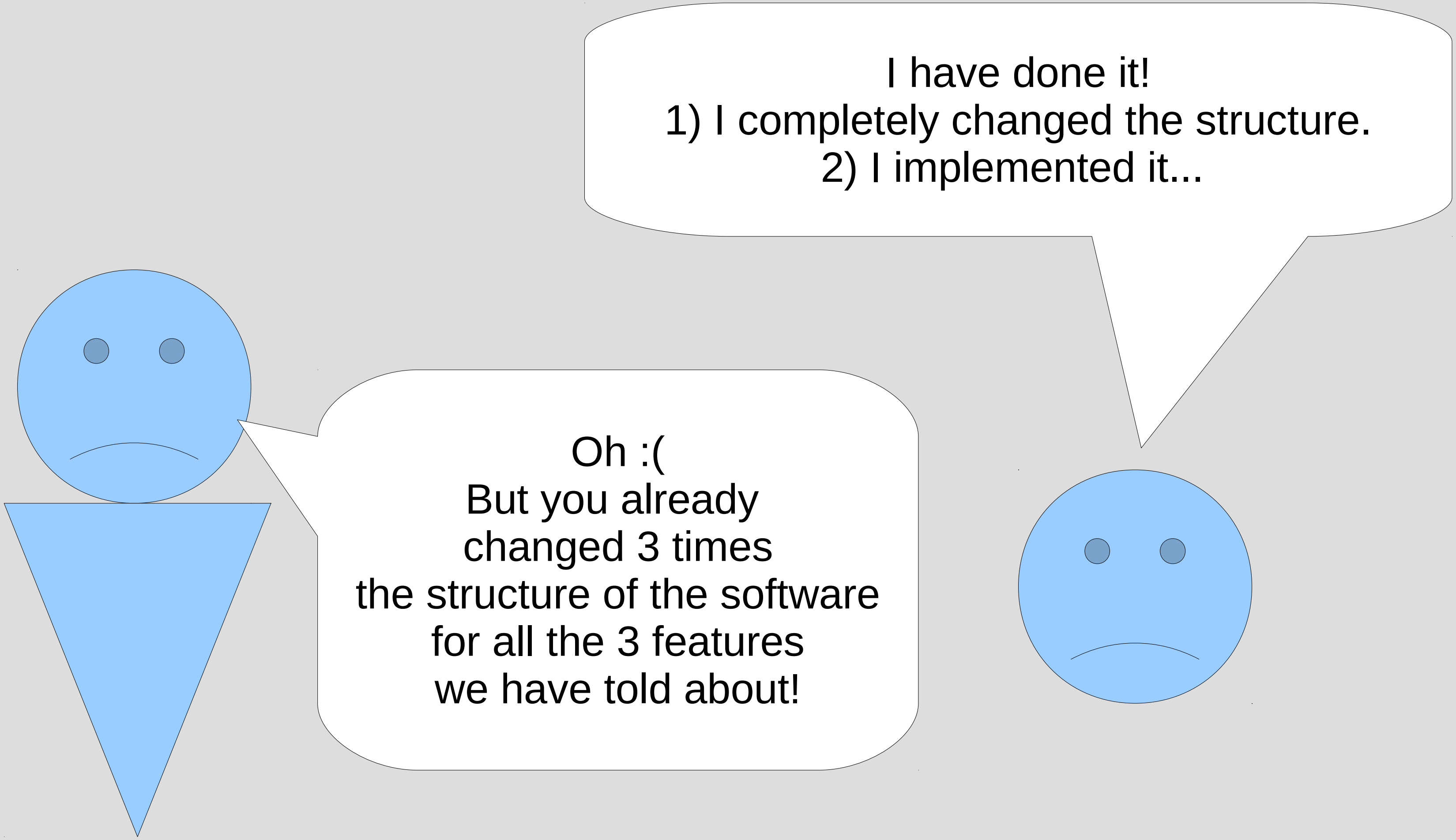
Fragility


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

Fragility

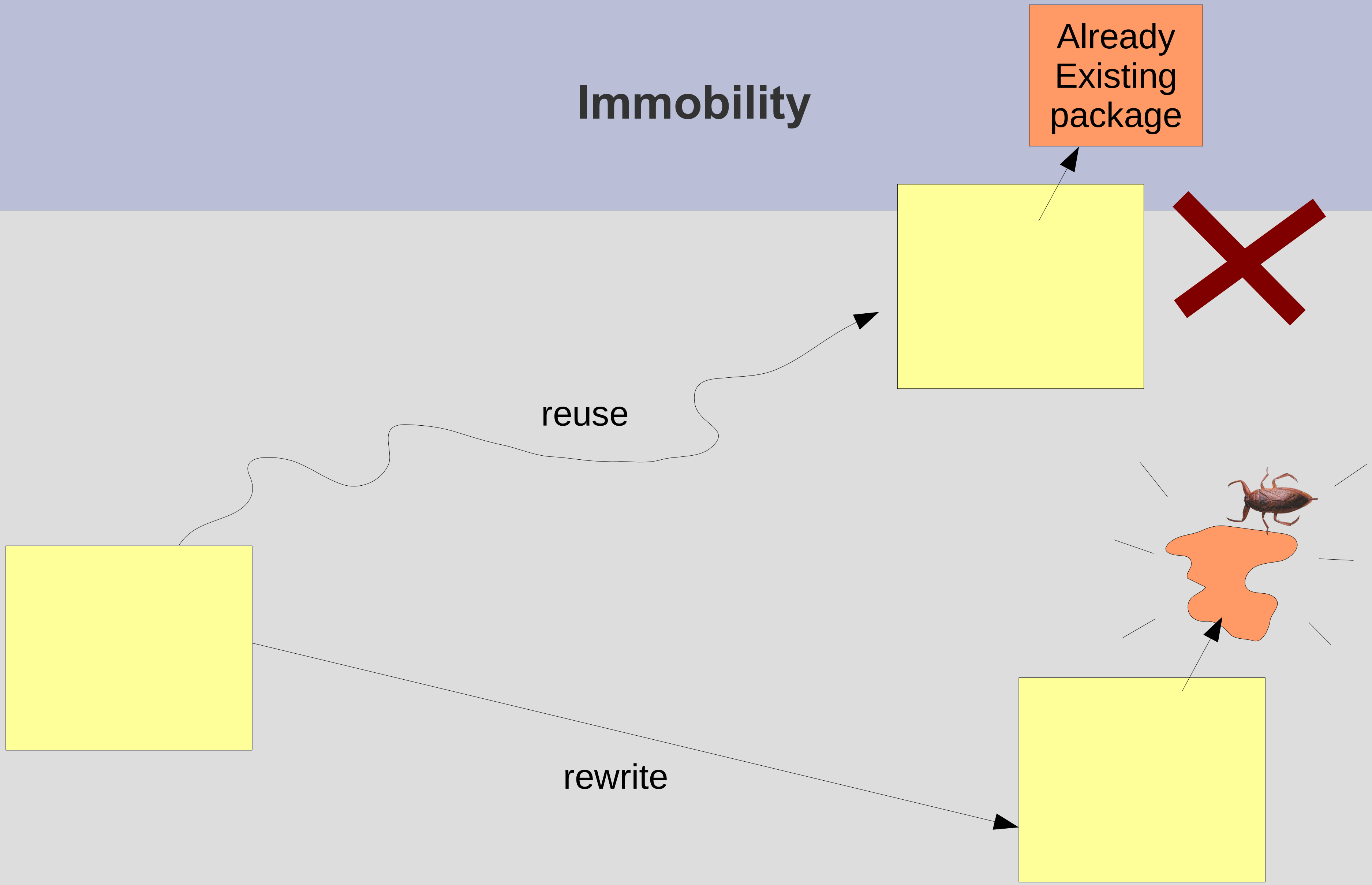

I have to add this functionality ...
and also to modify 120 different functions
Oh no... I have to change those 50 classes ...
and to redesign all ..
BOUM

Fragility


I have done it! 1) I completely changed the structure. 2) I implemented it...
Oh :(But you already changed 3 times the structure of the software for all the 3 features we have told about!

The developer team is not to be trusted.

Immobility

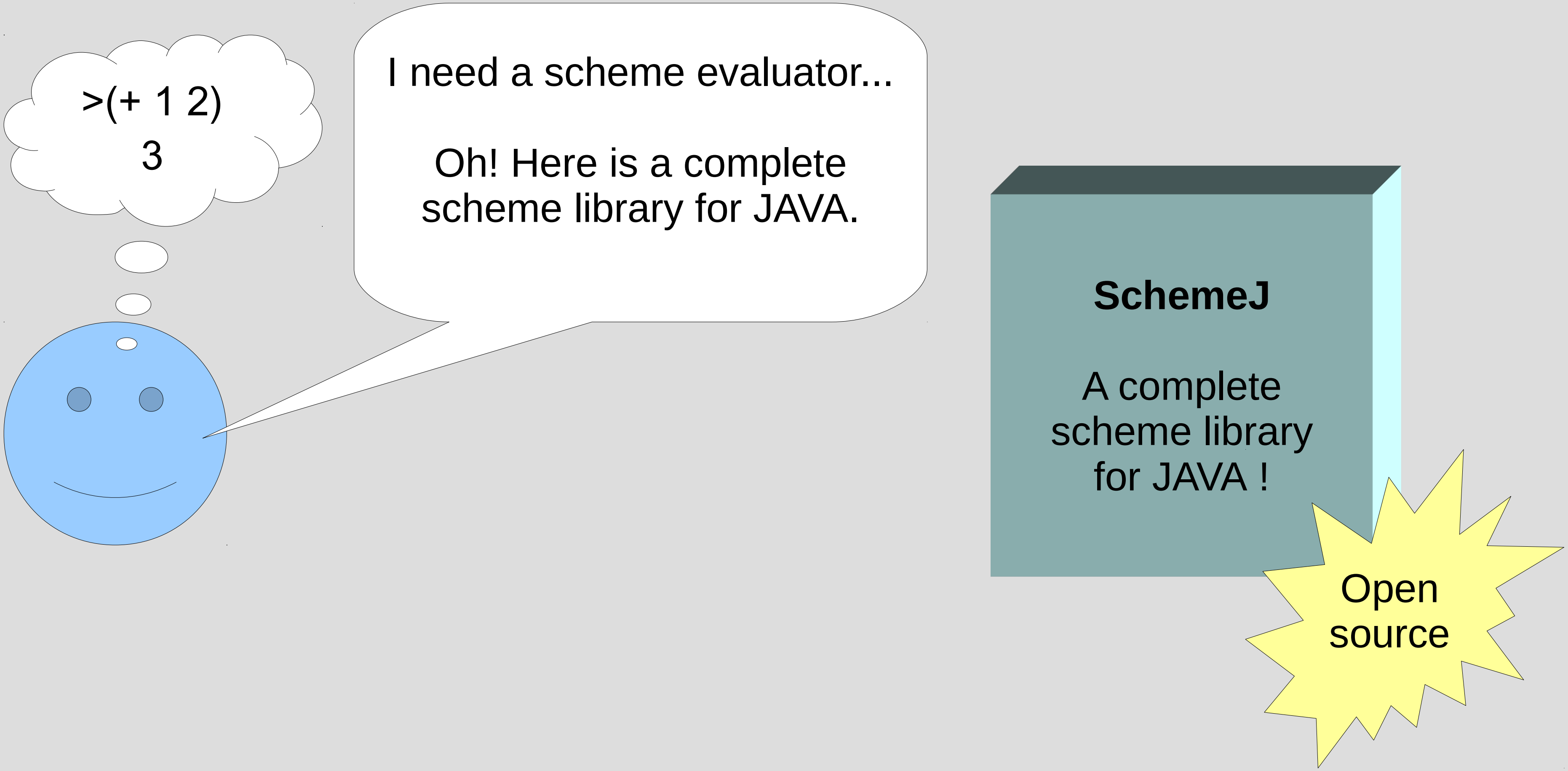


Already Existing package

reuse

rewrite

Immobility



```
>(+ 1 2)
3
```

I need a scheme evaluator...

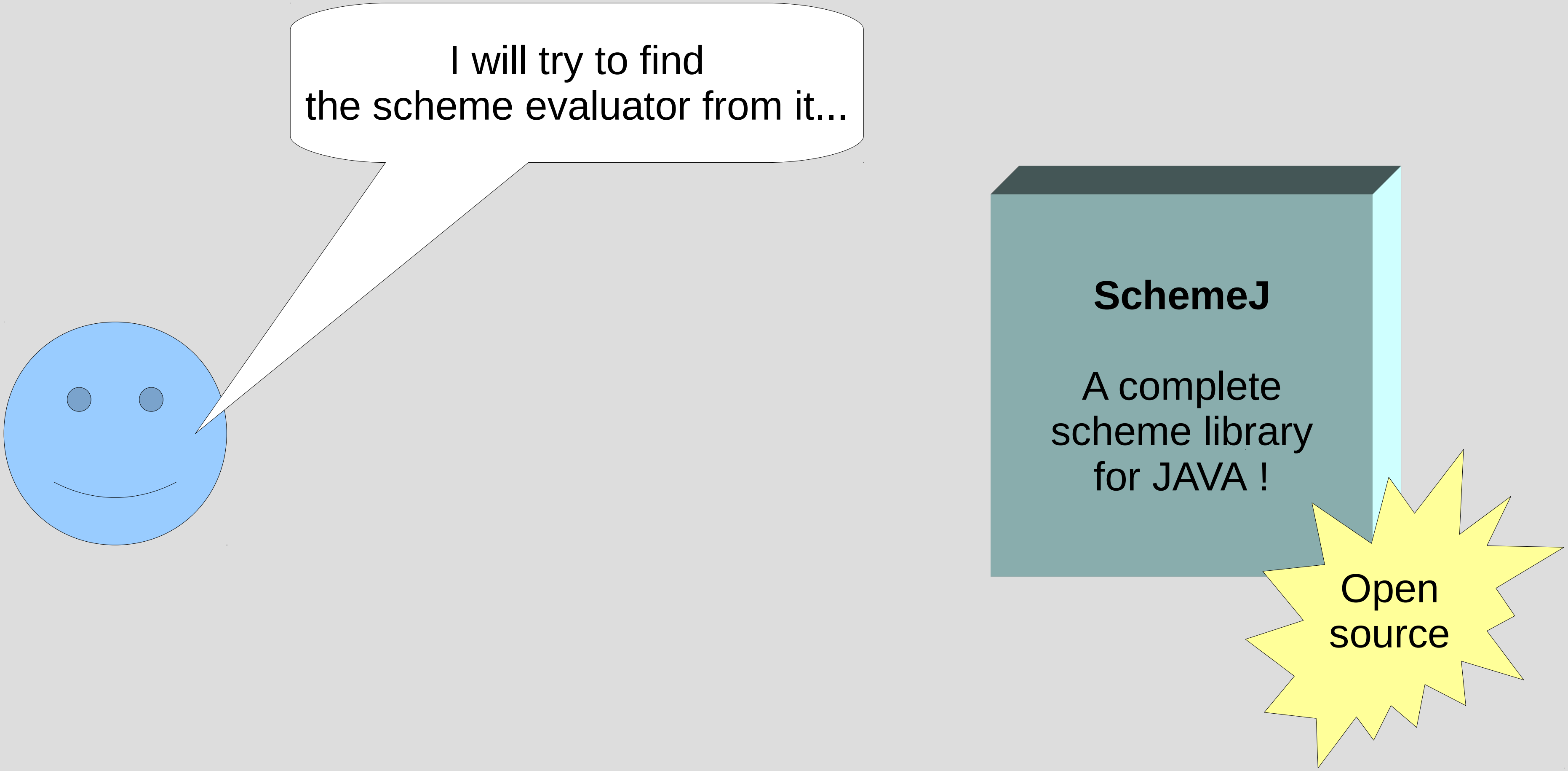
Oh! Here is a complete scheme library for JAVA.

SchemeJ

A complete scheme library for JAVA !

Open source

Immobility



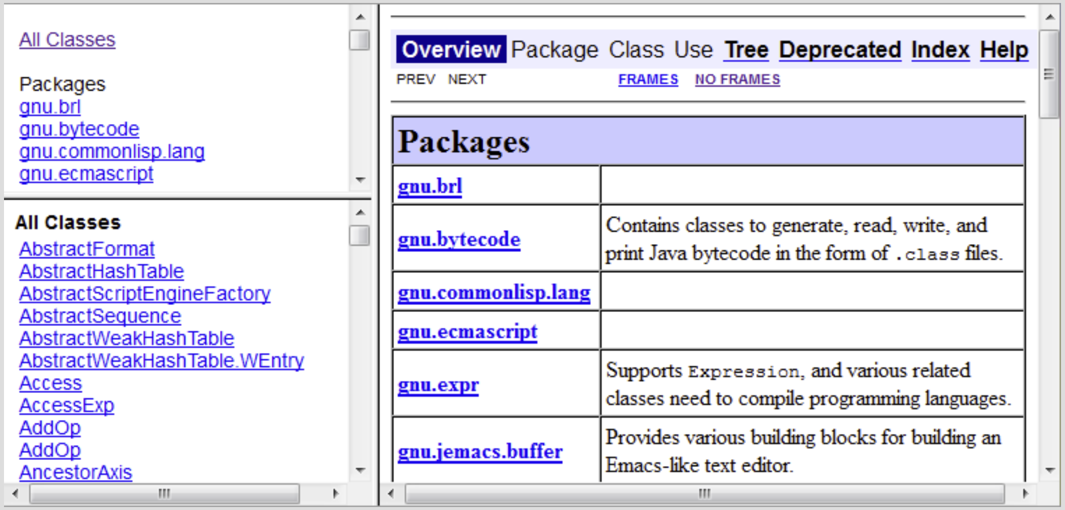
I will try to find
the scheme evaluator from it...

SchemeJ

A complete
scheme library
for JAVA !

Open
source

Immobility



All Classes

Packages
gnu.brl
gnu.bytecode
gnu.commonlisp.lang
gnu.ecmascript

All Classes
AbstractFormat
AbstractHashTable
AbstractScriptEngineFactory
AbstractSequence
AbstractWeakHashTable
AbstractWeakHashTable.WEntry
Access
AccessExp
AddOp
AddOp
AncestorAxis

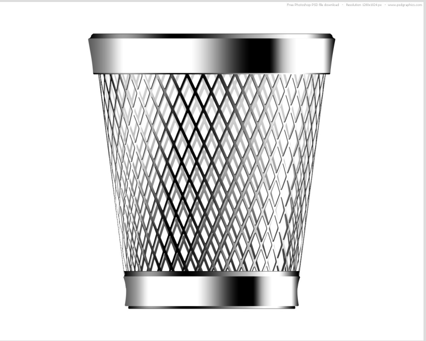
Overview Package Class Use **Tree** **Deprecated** **Index** **Help**
PREV NEXT FRAMES NO FRAMES

Packages	
gnu.brl	
gnu.bytecode	Contains classes to generate, read, write, and print Java bytecode in the form of `.class` files.
gnu.commonlisp.lang	
gnu.ecmascript	
gnu.expr	Supports `Expression`, and various related classes need to compile programming languages.
gnu.jemacs.buffer	Provides various building blocks for building an Emacs-like text editor.

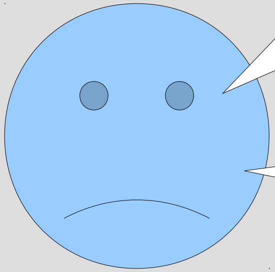
SchemeJ

A complete scheme library for JAVA !

Open source

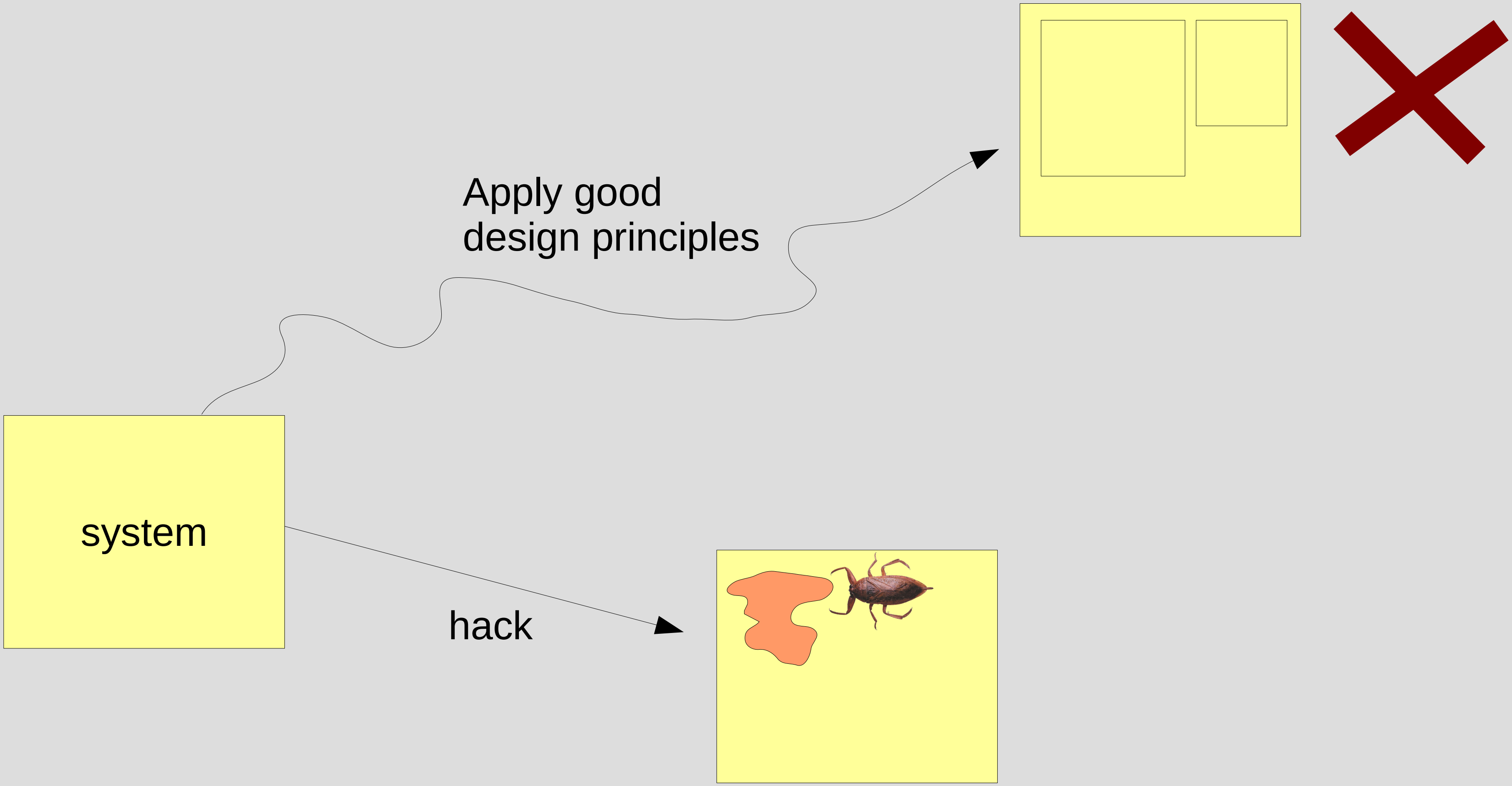


I do not understand anything.



I will write my own interpreter...

Viscosity

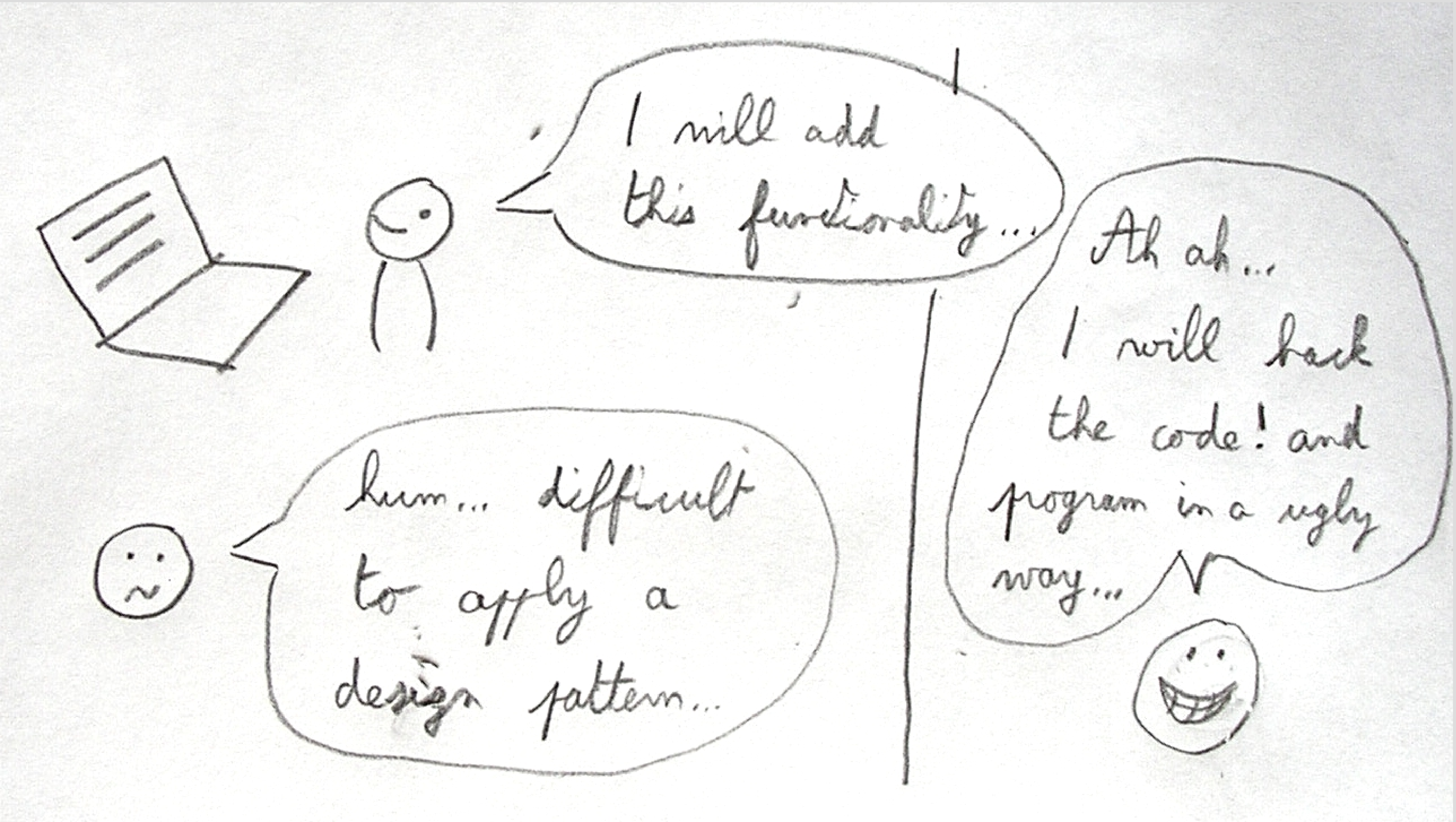


Apply good design principles

system

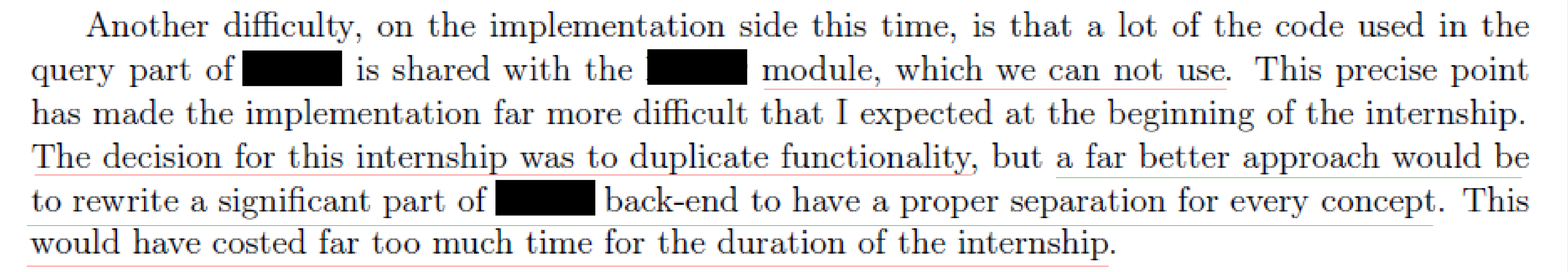
hack

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 - example

Extract from a MIT2 internship report



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.

Outline

- Symptoms of rotting systems
- **Principles of object oriented class design**
- Principles of Package Architecture
- Dreams

SOLID

- 5 principles of object oriented class design
- Introduced by Robert Cecil 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

Change the gravity ?

Change the way of counting the score ?

✓

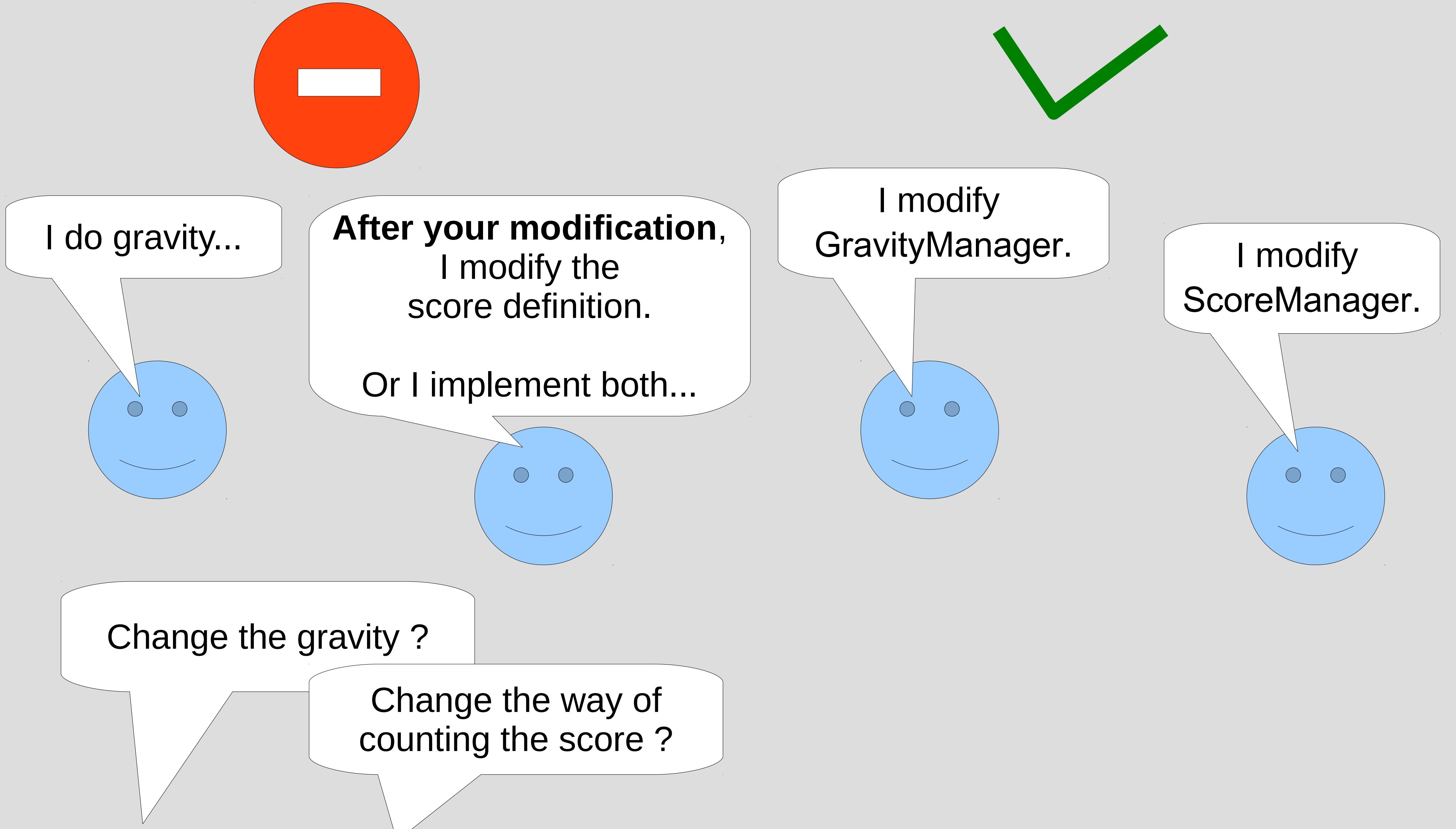
- Class of a game that uses two objects:

  - 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.



I do gravity...

After your modification, I modify the score definition.

Or I implement both...

I modify GravityManager.

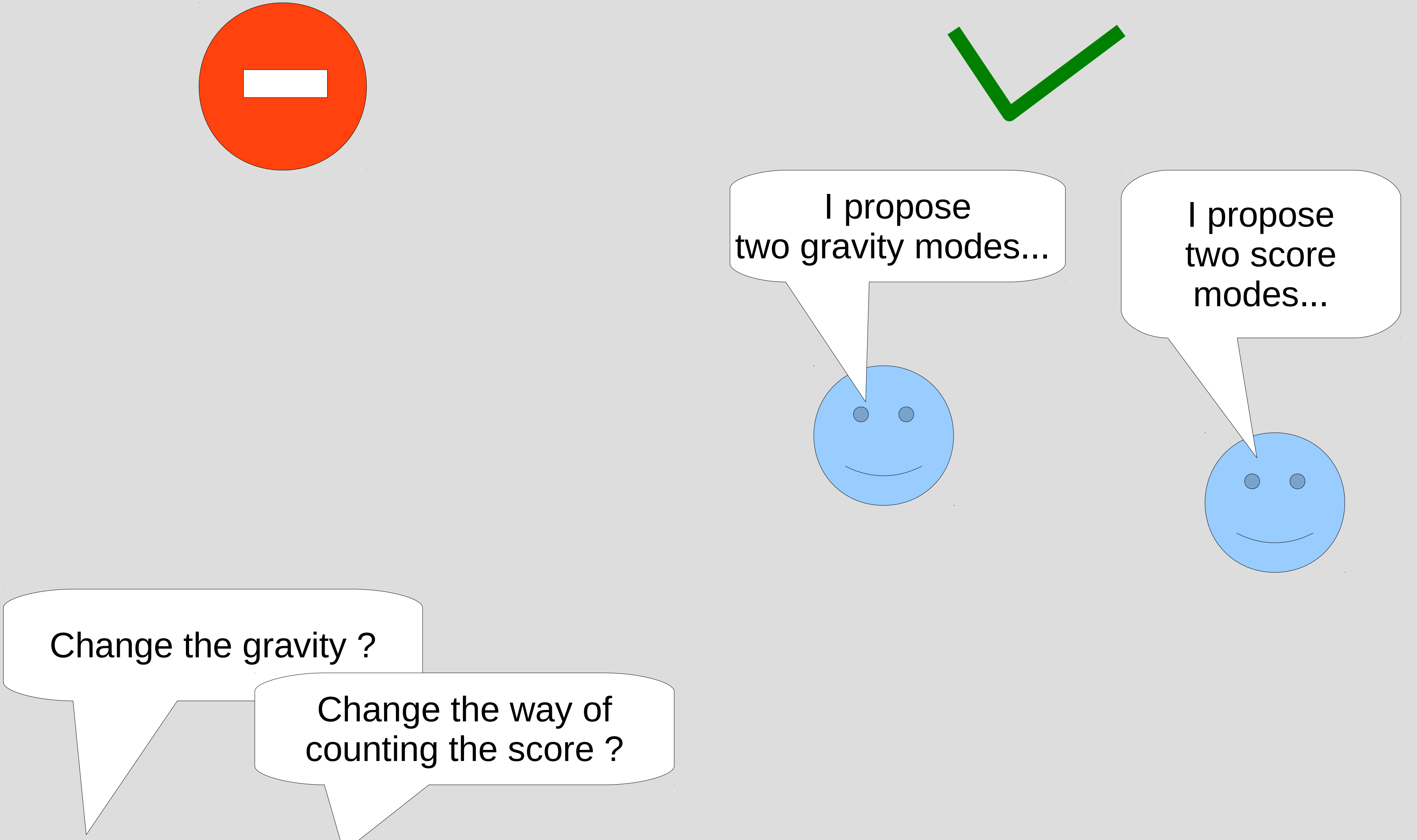
I modify ScoreManager.

Change the gravity ?

Change the way of counting the score ?

S : Single responsibility principle

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



I propose two gravity modes...

I propose two score modes...

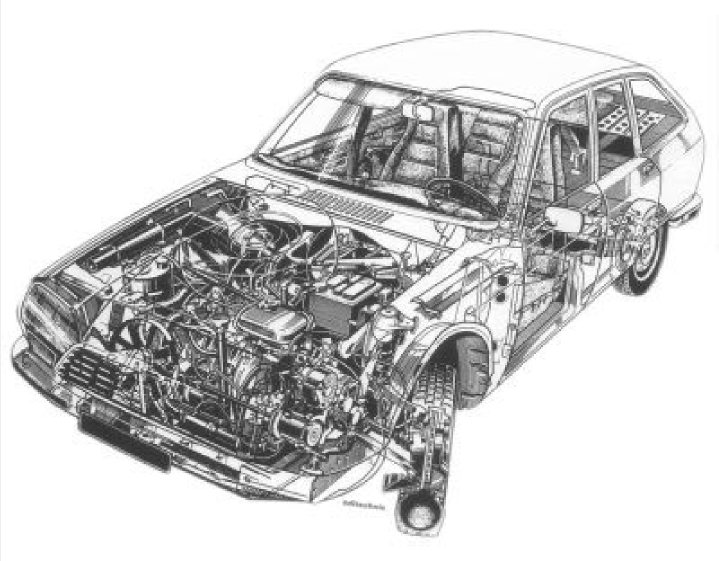
Change the gravity ?

Change the way of counting the score ?

O : Open Closed Principle

⛔

- Change the code source of module to add functionnality



✓

- Being able to extend modules without changing the code source

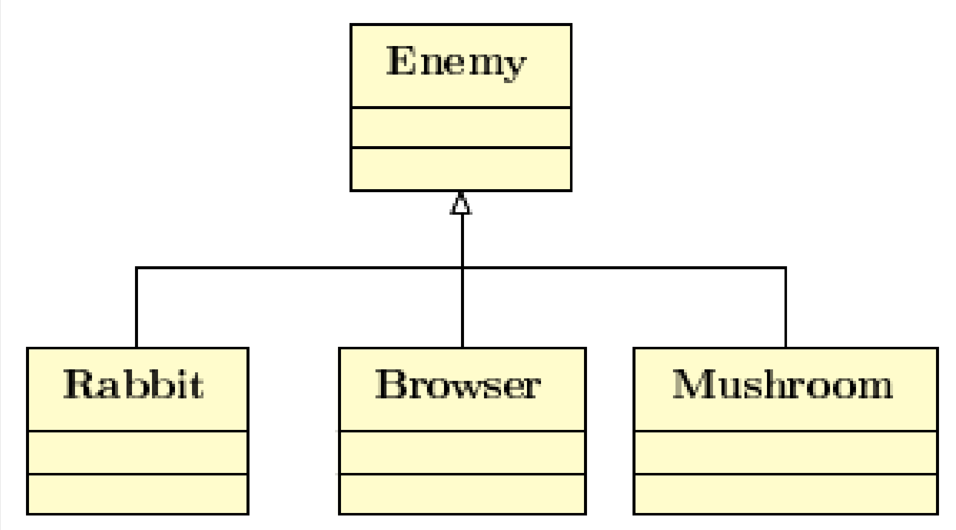
→ abstraction

O : Open Closed Principle

⛔

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

✓

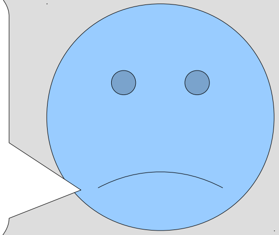


O : Open Closed Principle

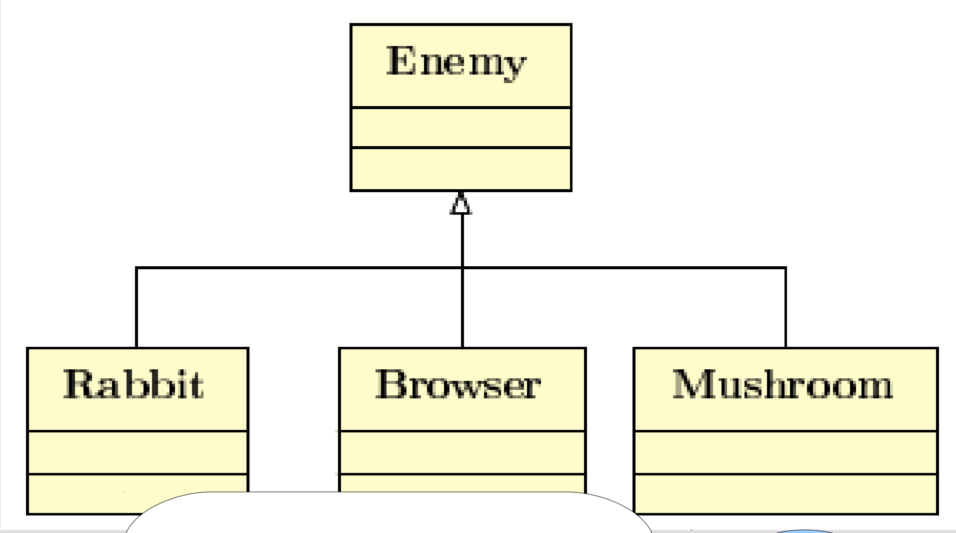
⛔

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

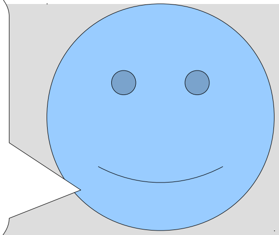
For each change of a type of enemy, I recommpile all the class Enemy...



✓



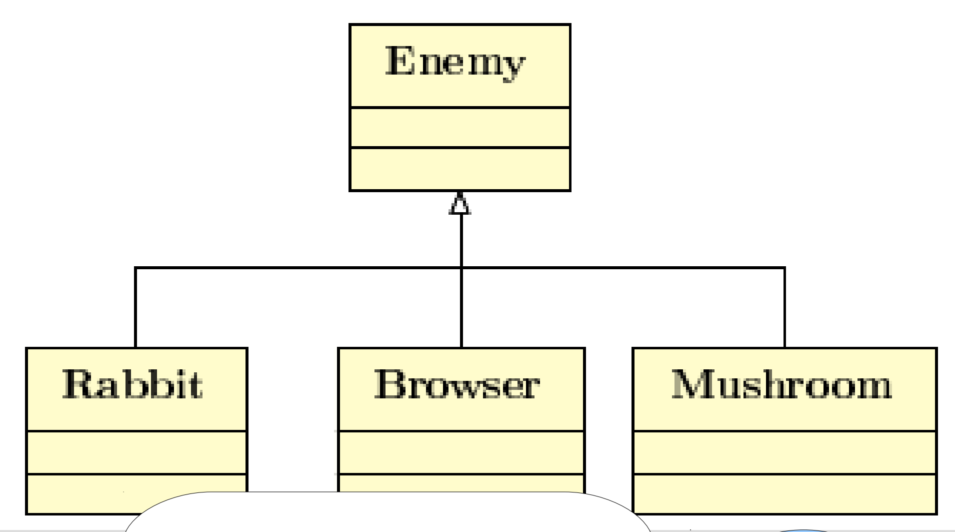
Oh I just recompile one class...



O : Open Closed Principle

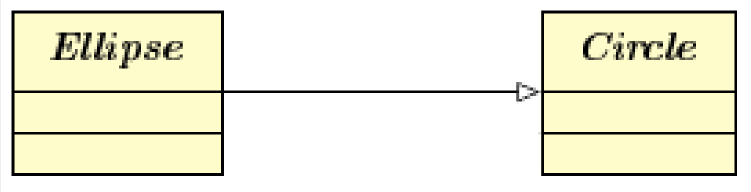
```
Class Enemy
{
        void move()
        {
                if(type == RABBIT)
                        ...
                else if(type == BROWSER)
                        ...
                else if(type == MUSHROOM)
                        ...
        }

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

        }
        .
        .
        .
}
```

If I add a new
type of enemy,
I check all the if/else
statements...

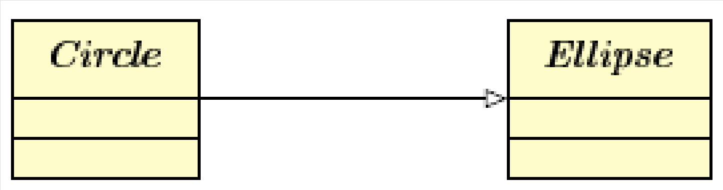


Oh I just add
a new class...

L : Liskov Substitution Principle



or

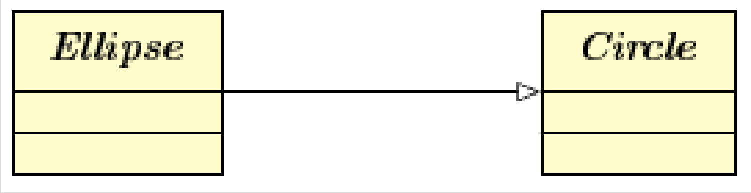


?

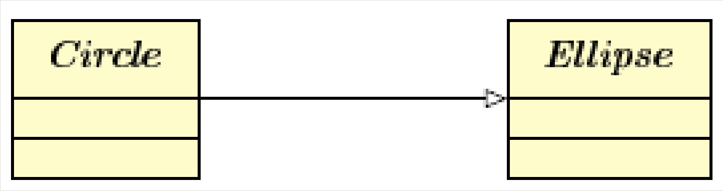
Barbara Liskov
Turing award 2008

L : Liskov Substitution Principle

⛔

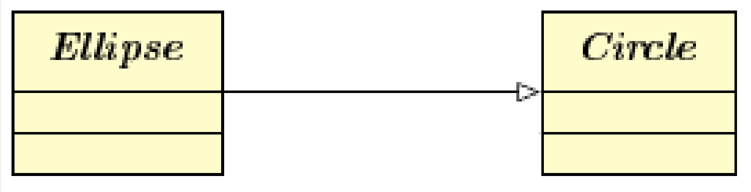


✓



L : Liskov Substitution Principle

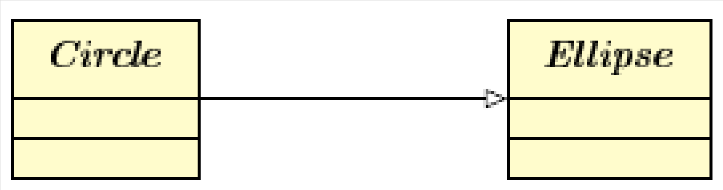
⛔



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

☹

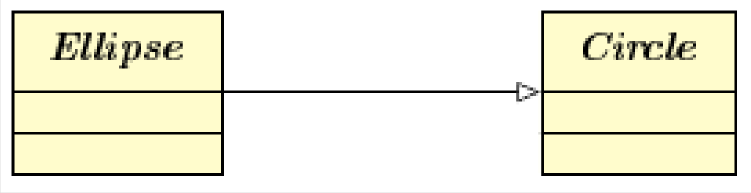
✓



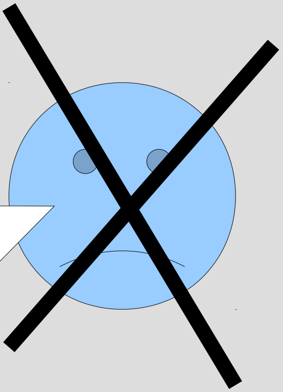
It seems reasonable...

☺

L : Liskov Substitution Principle



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

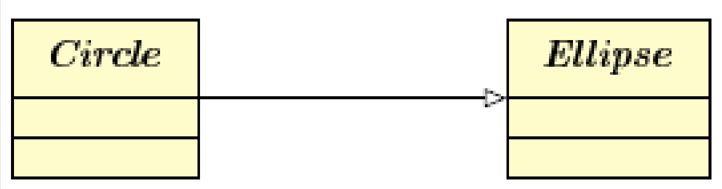


```
Class Circle
{
    public float getR();
    public float getArea();

}

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
{
    public float getR1();
    public float getR2();
    public float getArea();

}

Class Circle extends Ellipse
{
    ...
}


Ellipse e;
e = new Circle(...);

/* Here we expect that
the area of e is
c.getR1() * c.getR2() * PI
*/
```

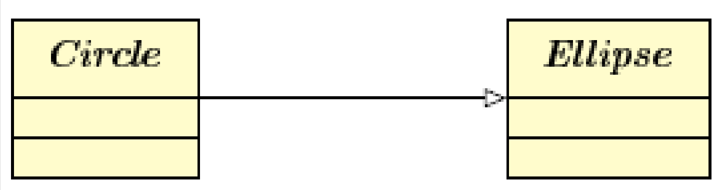
✓



L : Liskov Substitution Principle and design by contract

```
Class Ellipse
{
    invariant: inv

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

Class Circle extends Ellipse
{
    invariant: stronger than inv

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

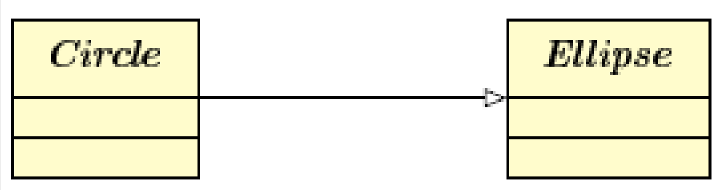
✓



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;
    }
    :
}

:
```

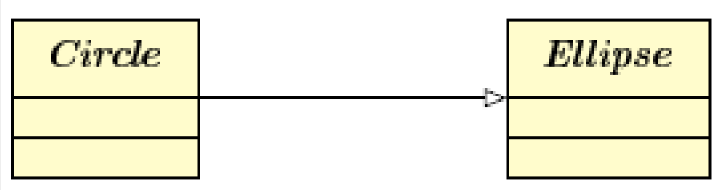
✓



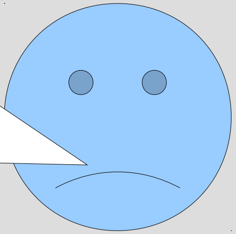
☺

A little problem

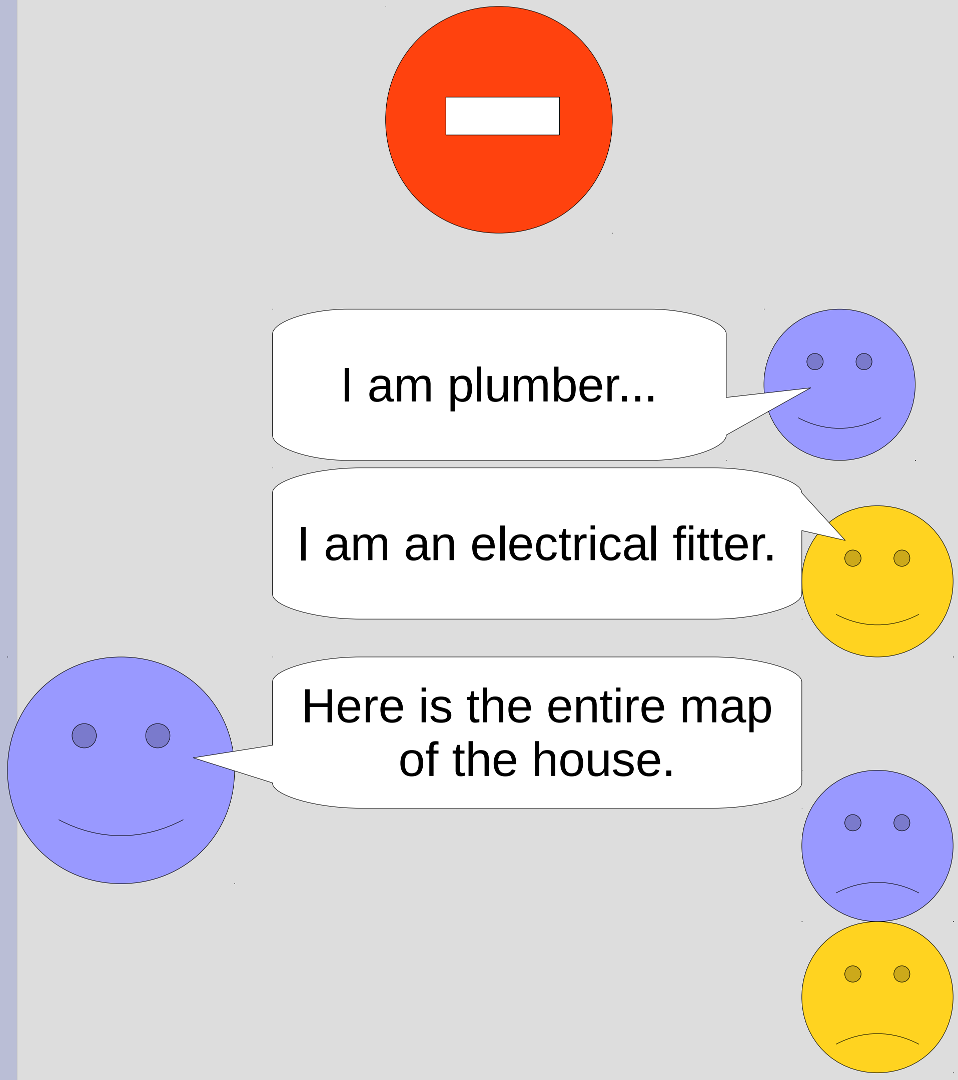
```
Class Ellipse
{
    postcondition:
       this.p1 == p1 & this.p2 == p2
    void setFocus(Point p1, p2)
    {
        this.p1 = p1;
        this.p2 = p2;
    }
    :
}

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);
```



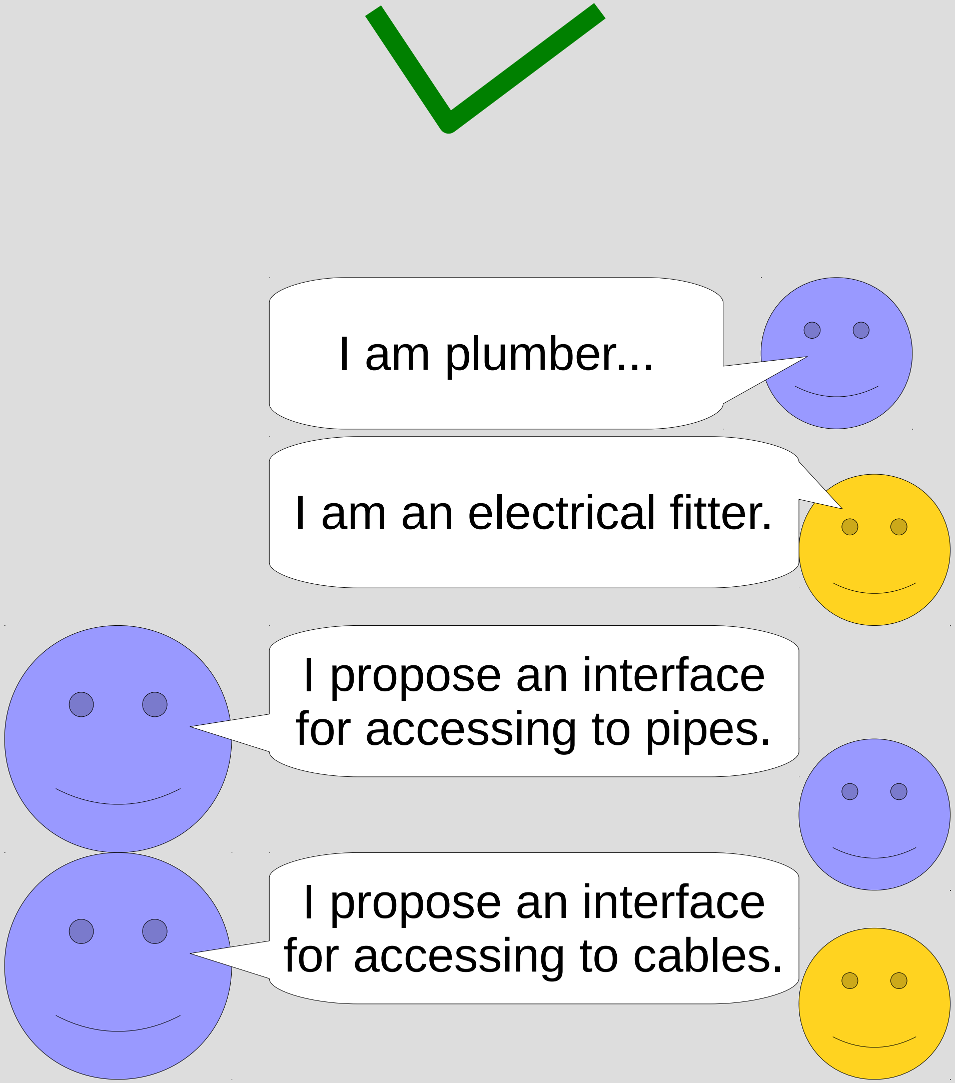
I : Interface Segregation Principle



I am plumber...

I am an electrical fitter.

Here is the entire map of the house.



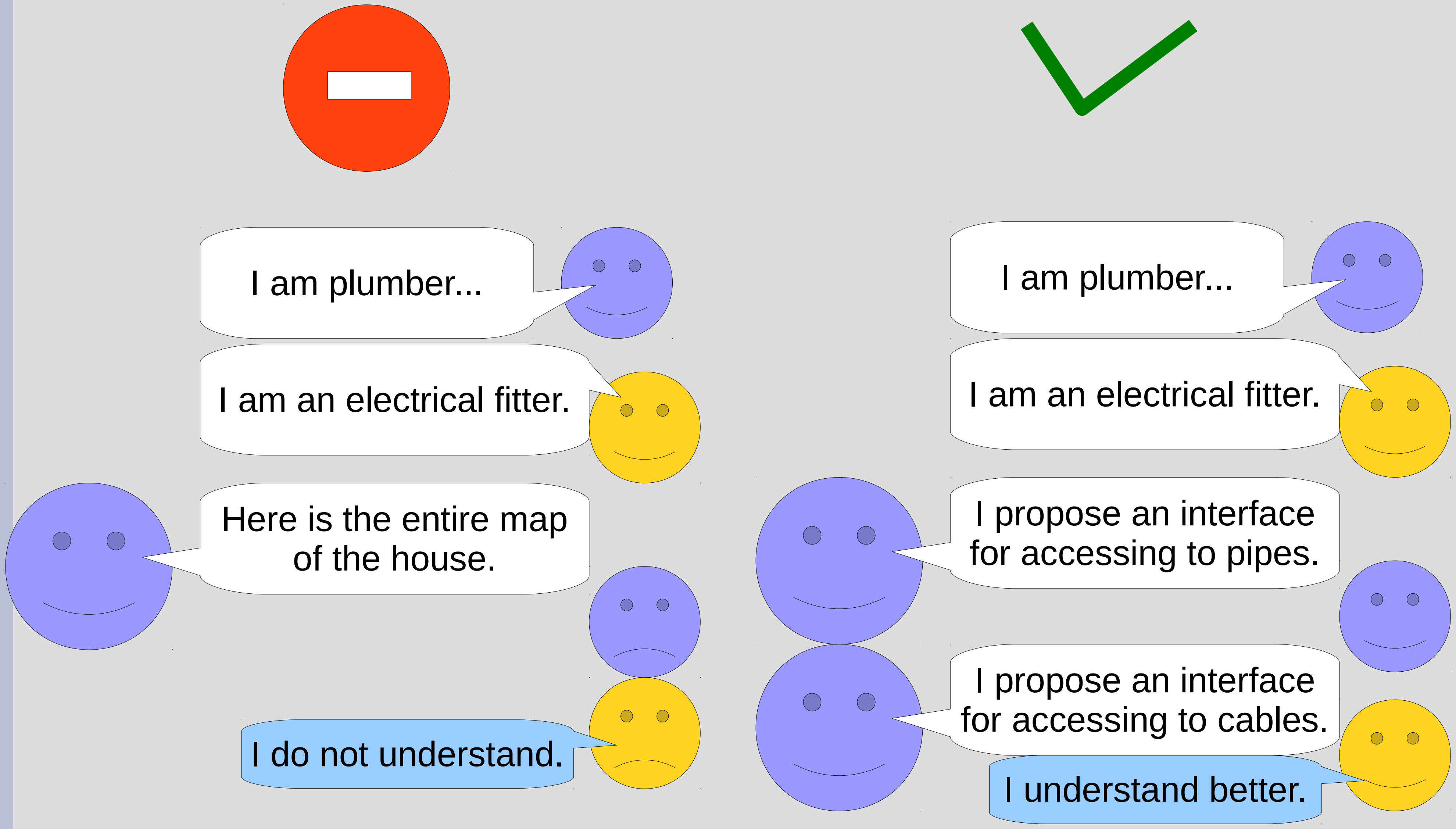
I am plumber...

I am an electrical fitter.

I propose an interface for accessing to pipes.

I propose an interface for accessing to cables.

I : Interface Segregation Principle



I am plumber...

I am an electrical fitter.

Here is the entire map of the house.

I do not understand.

I am plumber...

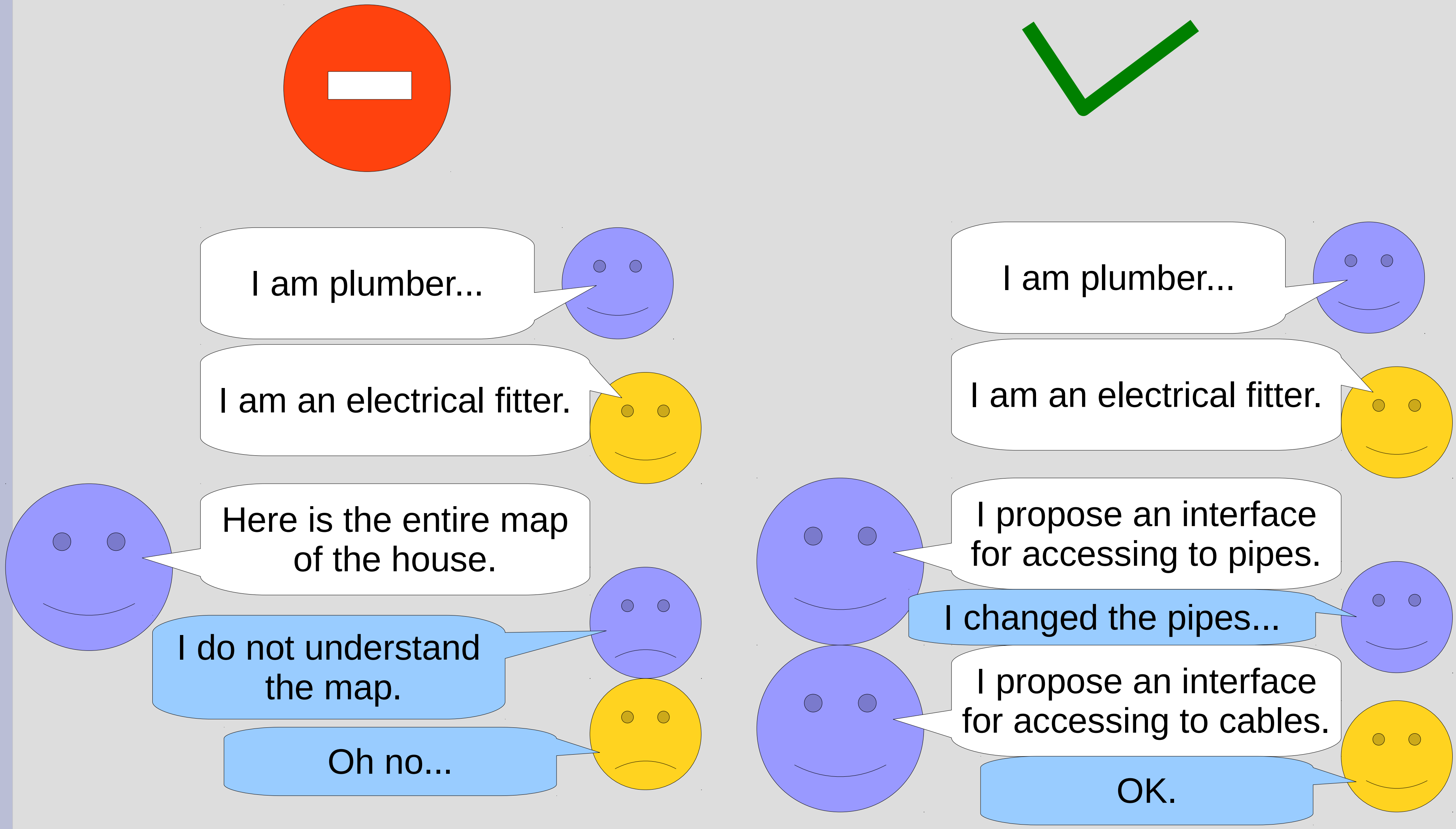
I am an electrical fitter.

I propose an interface for accessing to pipes.

I propose an interface for accessing to cables.

I understand better.

I : Interface Segregation Principle



I am plumber...

I am an electrical fitter.

Here is the entire map of the house.

I do not understand the map.

Oh no...

I am plumber...

I am an electrical fitter.

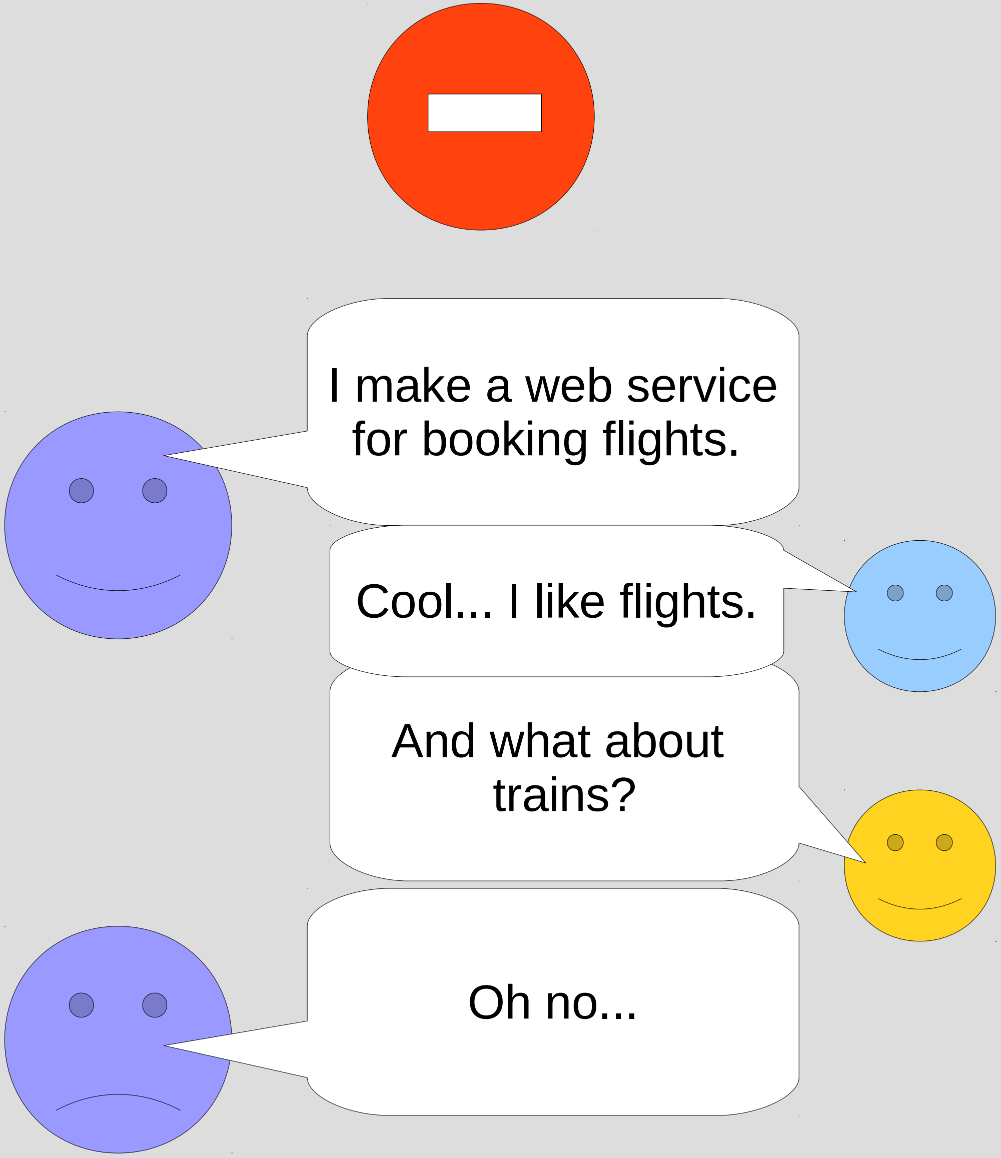
I propose an interface for accessing to pipes.

I changed the pipes...

I propose an interface for accessing to cables.

OK.

D: Dependency Inversion Principle

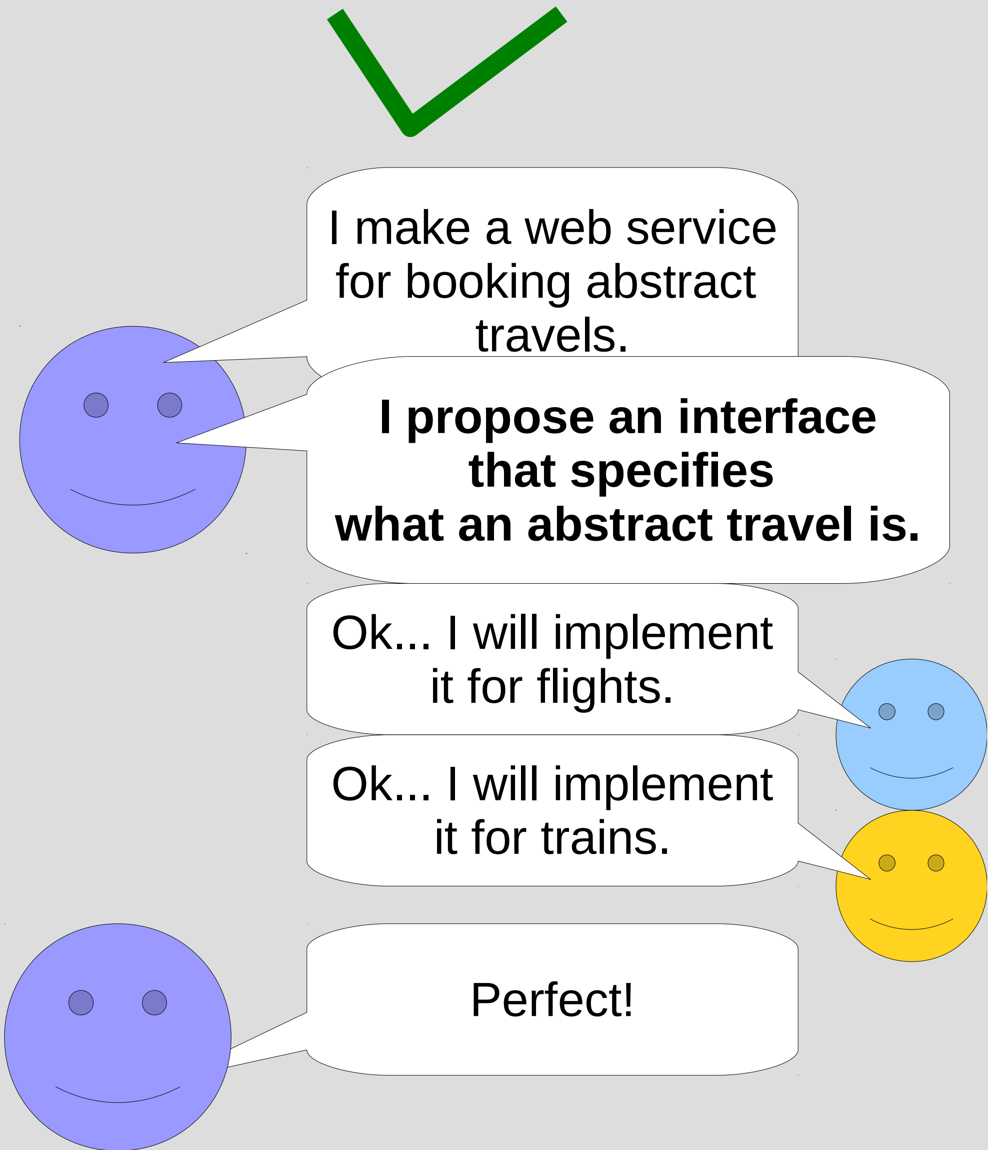


I make a web service for booking flights.

Cool... I like flights.

And what about trains?

Oh no...



I make a web service for booking abstract travels.

I propose an interface that specifies what an abstract travel is.

Ok... I will implement it for flights.

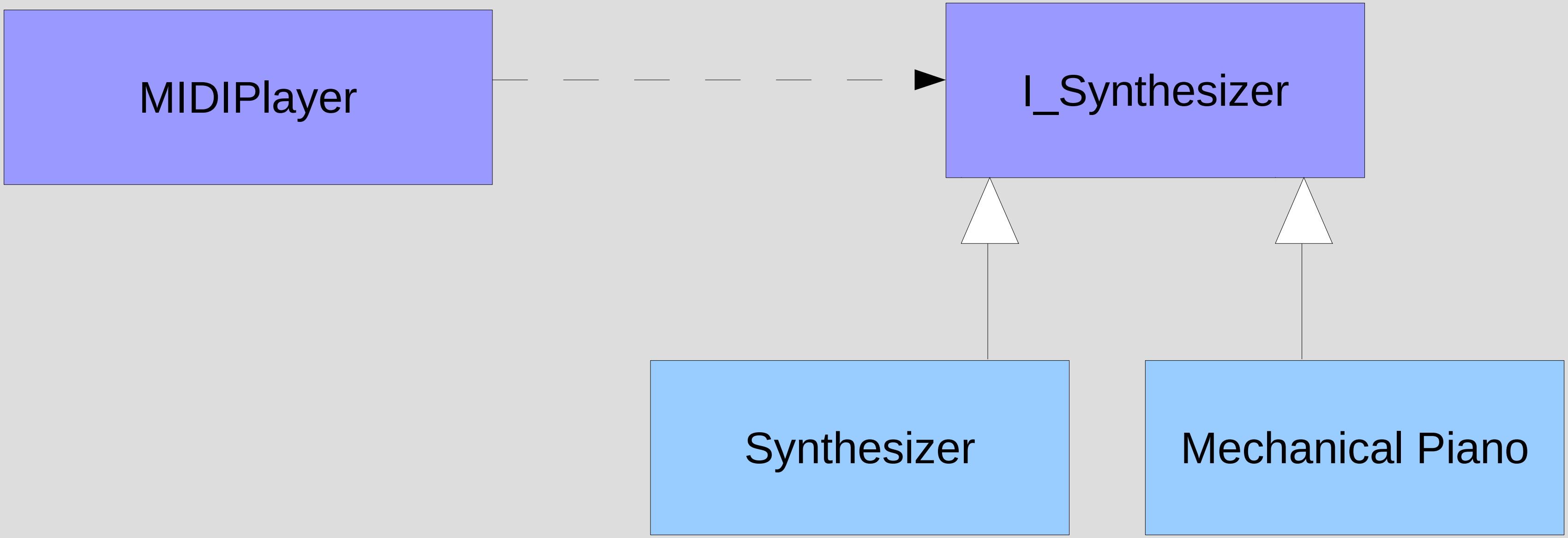
Ok... I will implement it for trains.

Perfect!

D: Dependency Inversion Principle

Example:

- JAVA MidiSound



Outline

- Symptoms of rotting systems
- Principles of object oriented class design
- **Principles of Package Architecture**
- Dreams

Outline

- Symptoms of rotting systems
- Principles of object oriented class design
- **Principles of Package Architecture**
  - **Inside a package**
  - Between packages
- Dreams

Remark

We refactor the packages during the development:

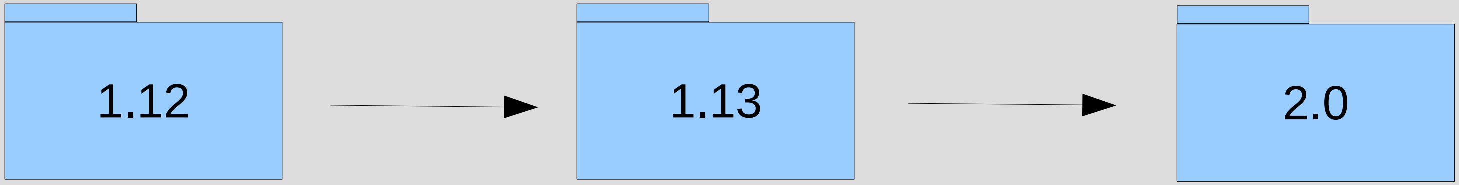
- At the beginning stage, we favor the developer
- At the end, we favor the clients.

The Release Reuse Equivalency Principle

A package

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

Good
for the client!



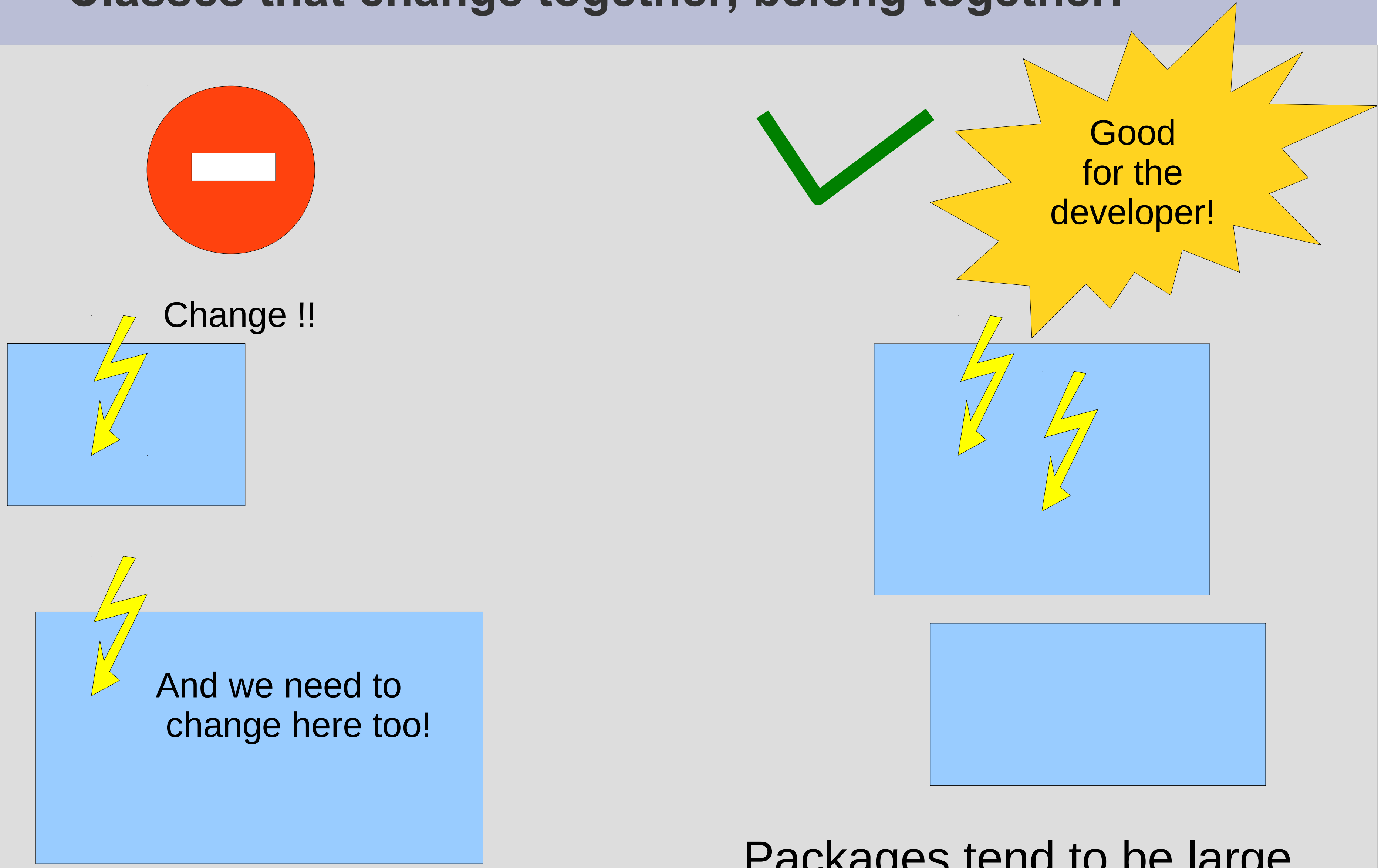
1.12

1.13

2.0

The Common Closure Principle

Classes that change together, belong together.



Change !!

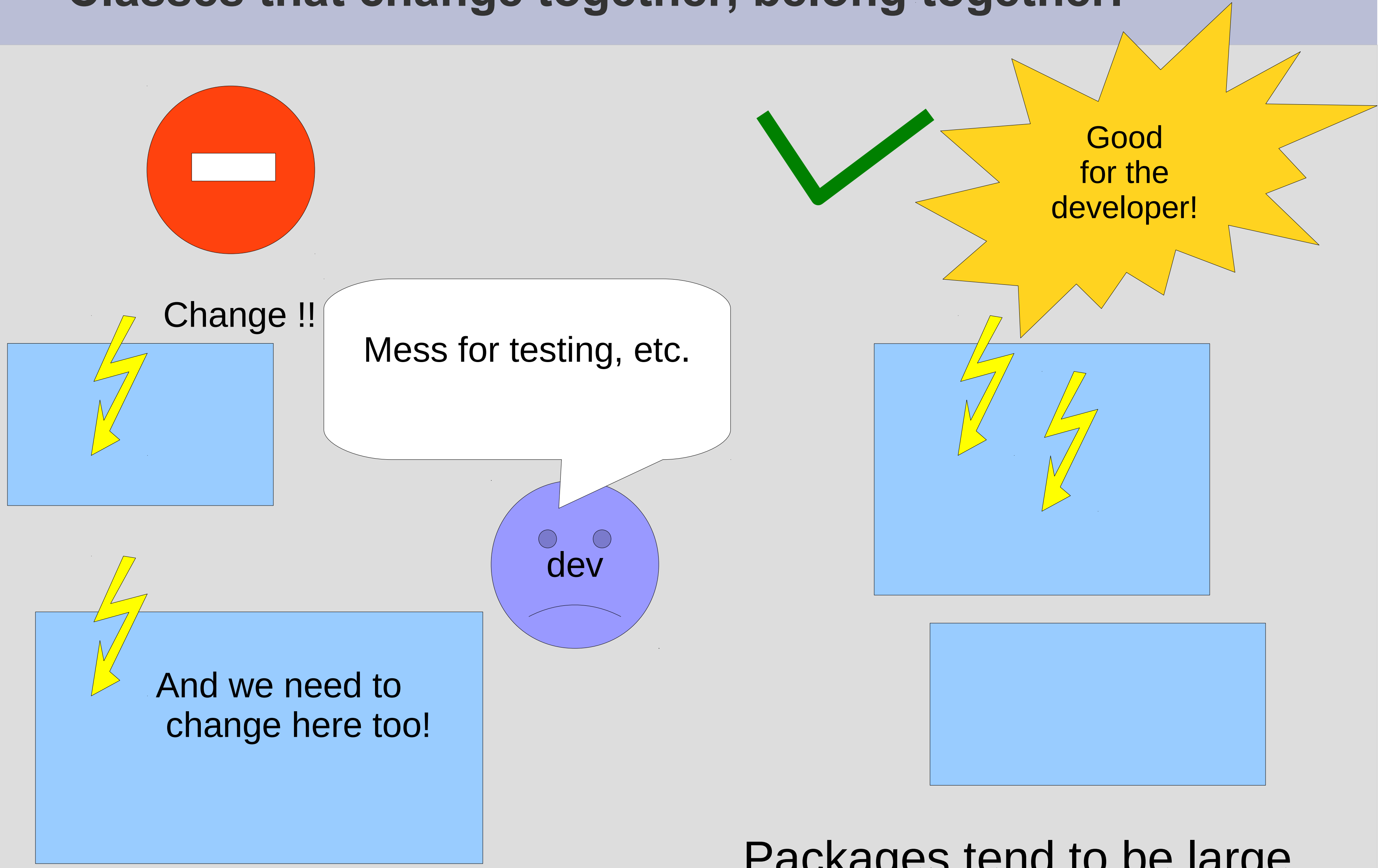
And we need to change here too!

Good for the developer!

Packages tend to be large.

The Common Closure Principle

Classes that change together, belong together.



Change !!

Mess for testing, etc.

dev

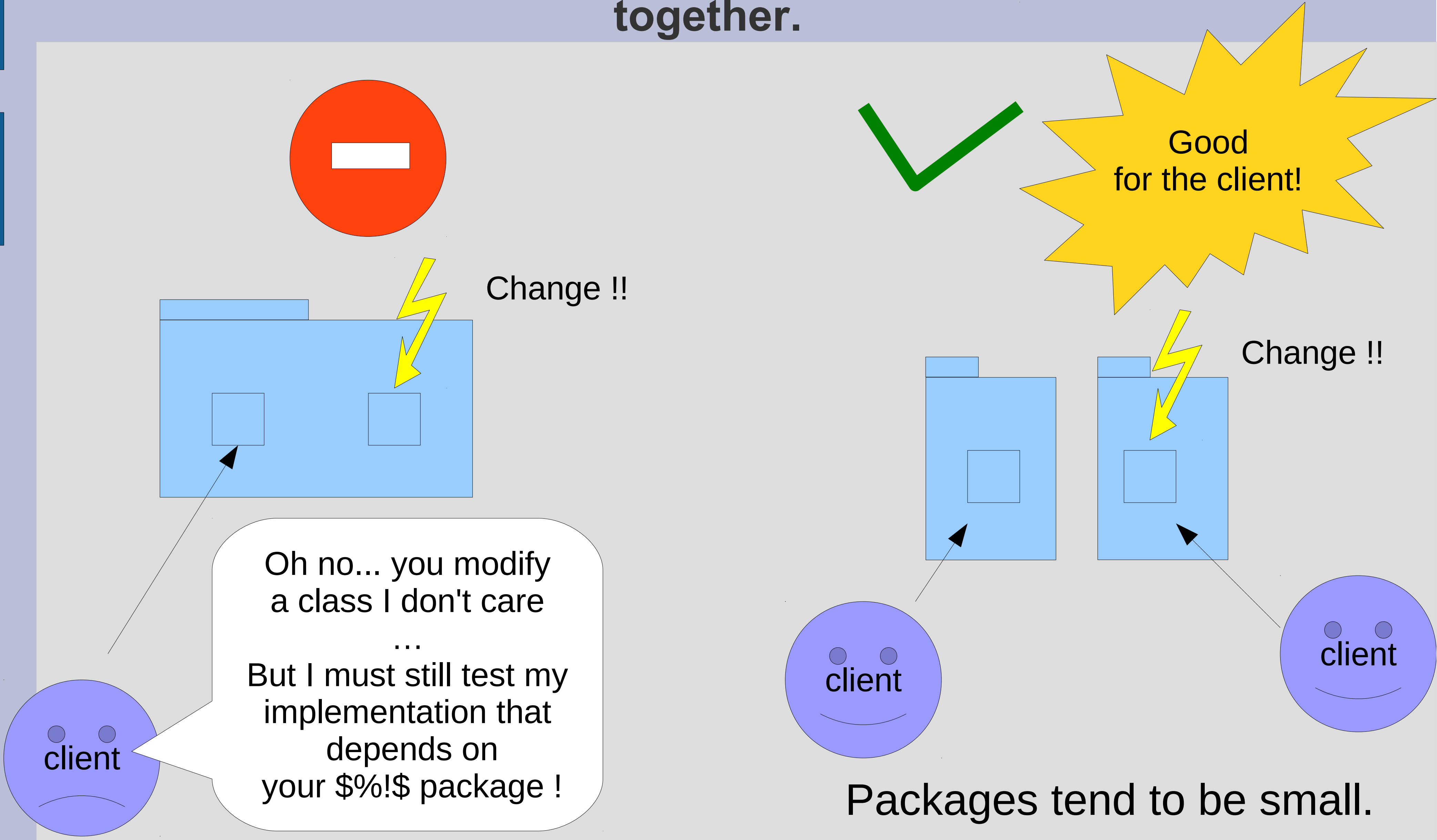
And we need to change here too!

Good for the developer!

Packages tend to be large.

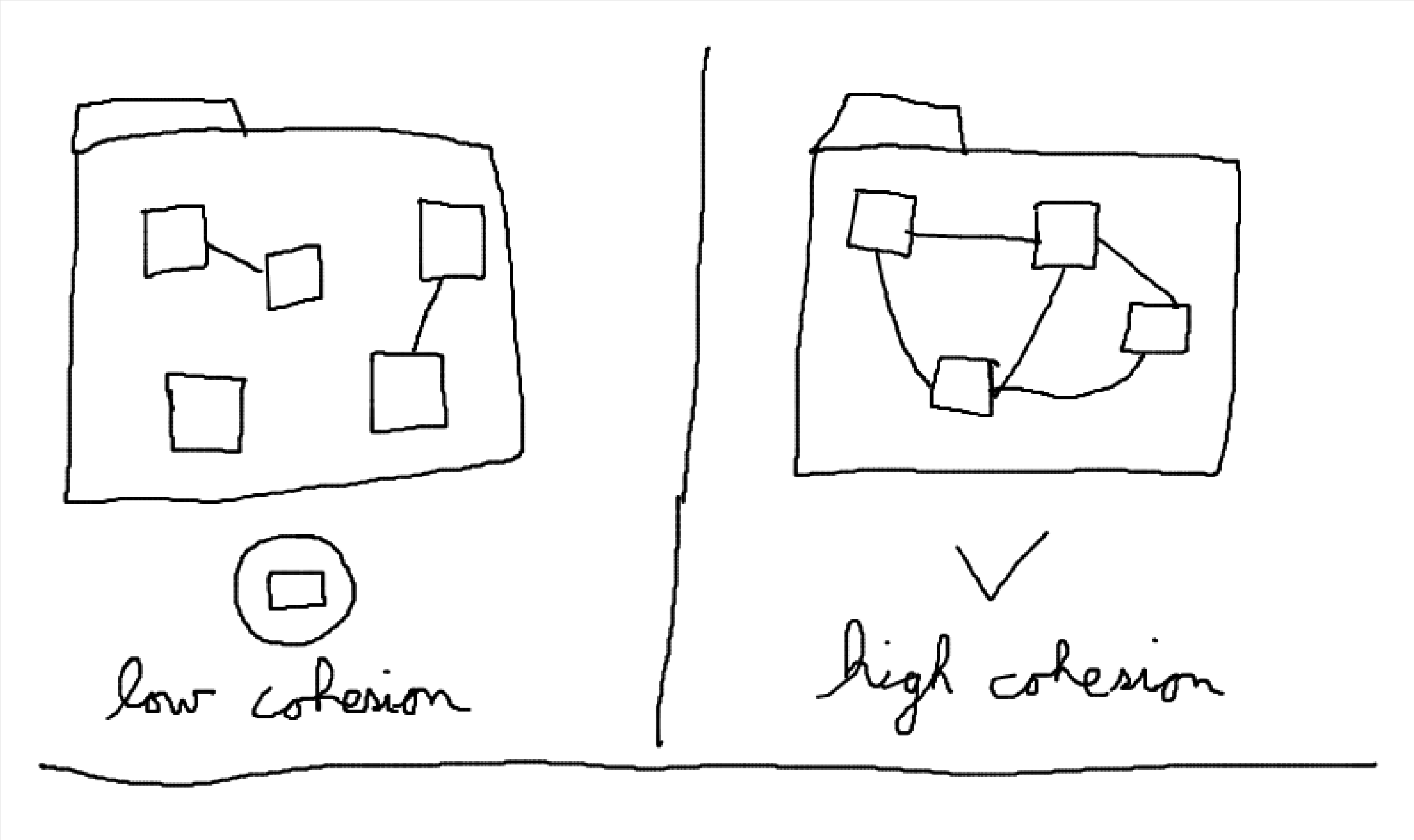
The Common Reuse Principle

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


Change !!
client
Oh no... you modify a class I don't care
...
But I must still test my implementation that depends on your $%!$ package !
Good for the client!
Change !!
client
client

Packages tend to be small.

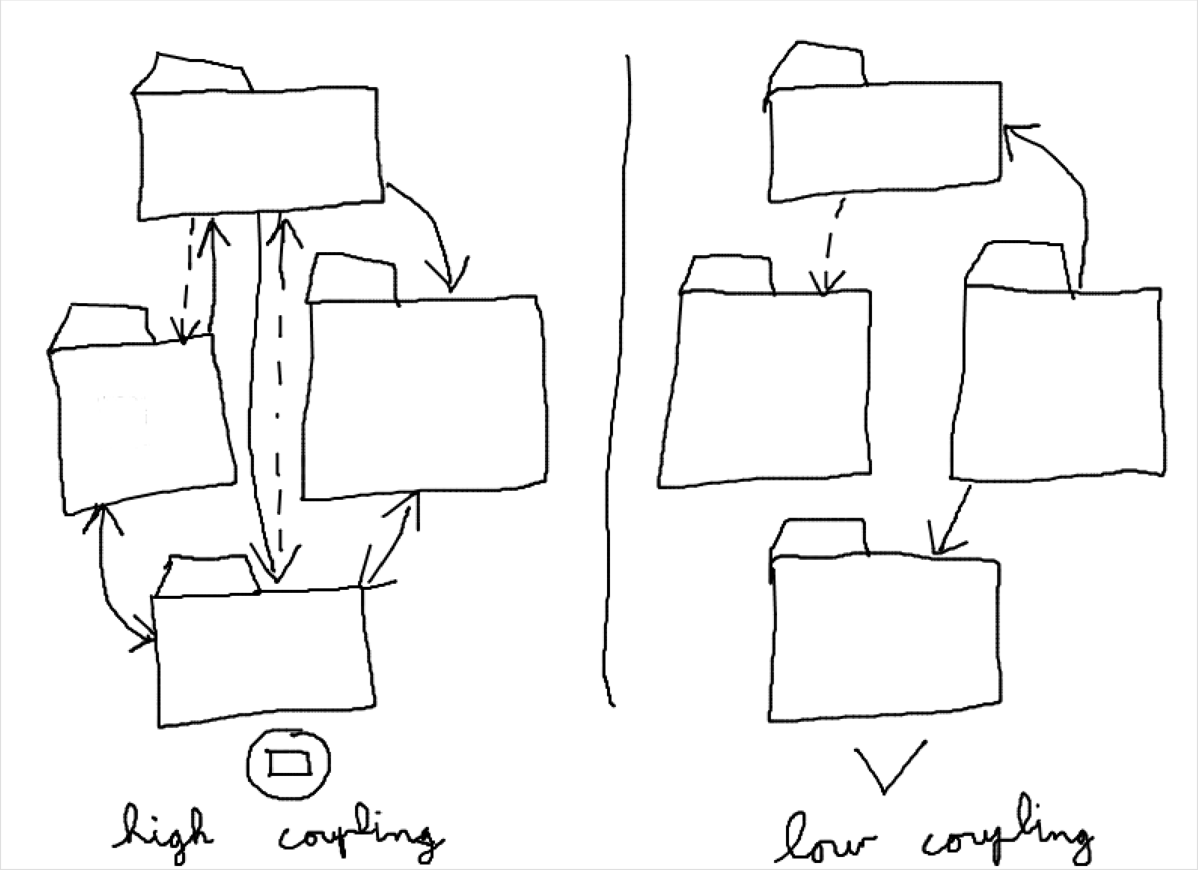
Cohesion



Outline

- Symptoms of rotting systems
- Principles of object oriented class design
- **Principles of Package Architecture**
  - Inside a package
  - **Between packages**
- Dreams

Coupling

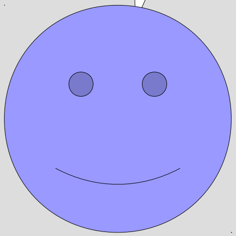


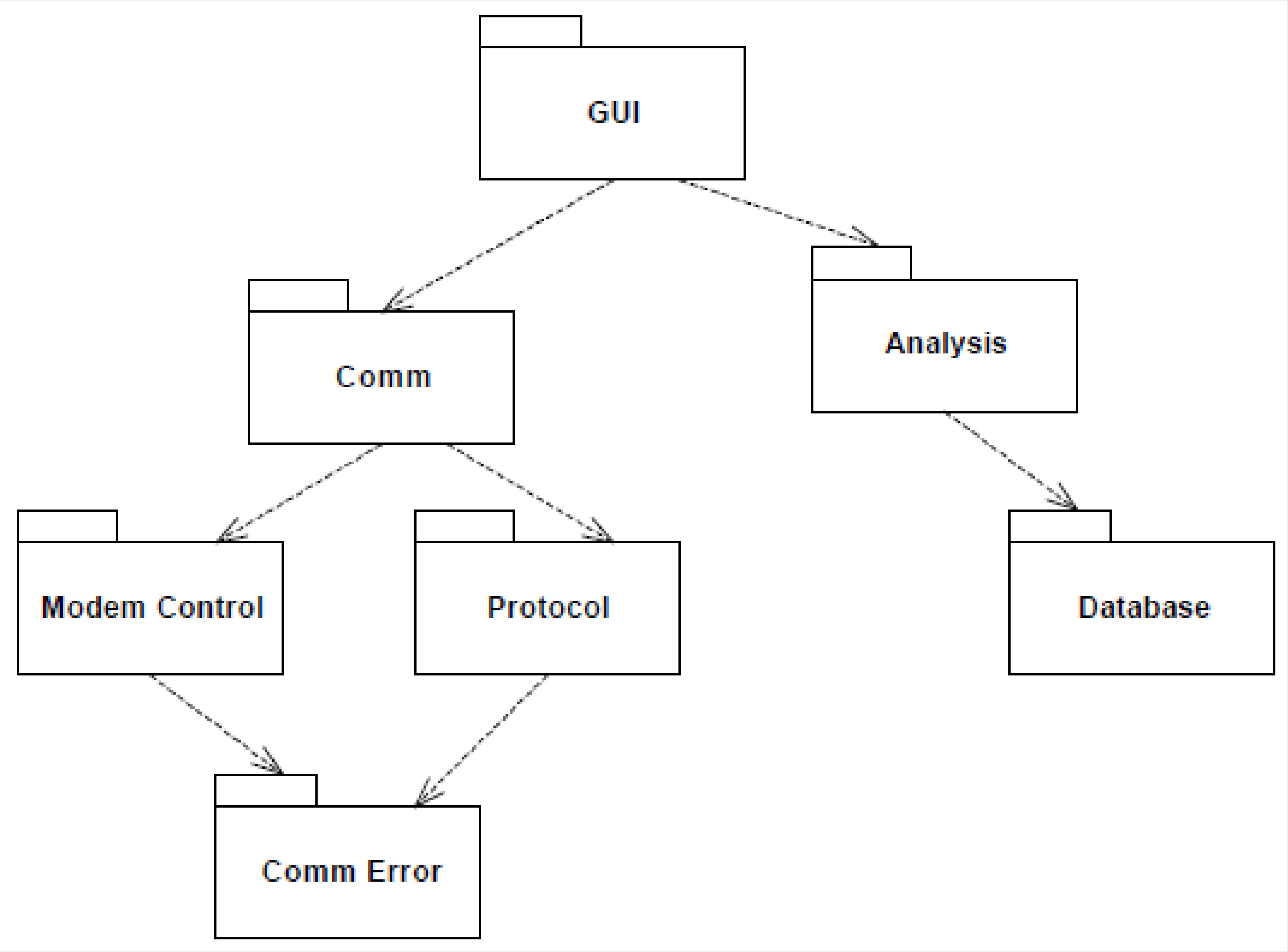
The Acyclic Dependencies Principle

The dependencies between packages must not form cycles.

✓

I work on Protocol... and I need to test my package with Comm Error.

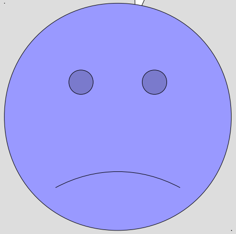


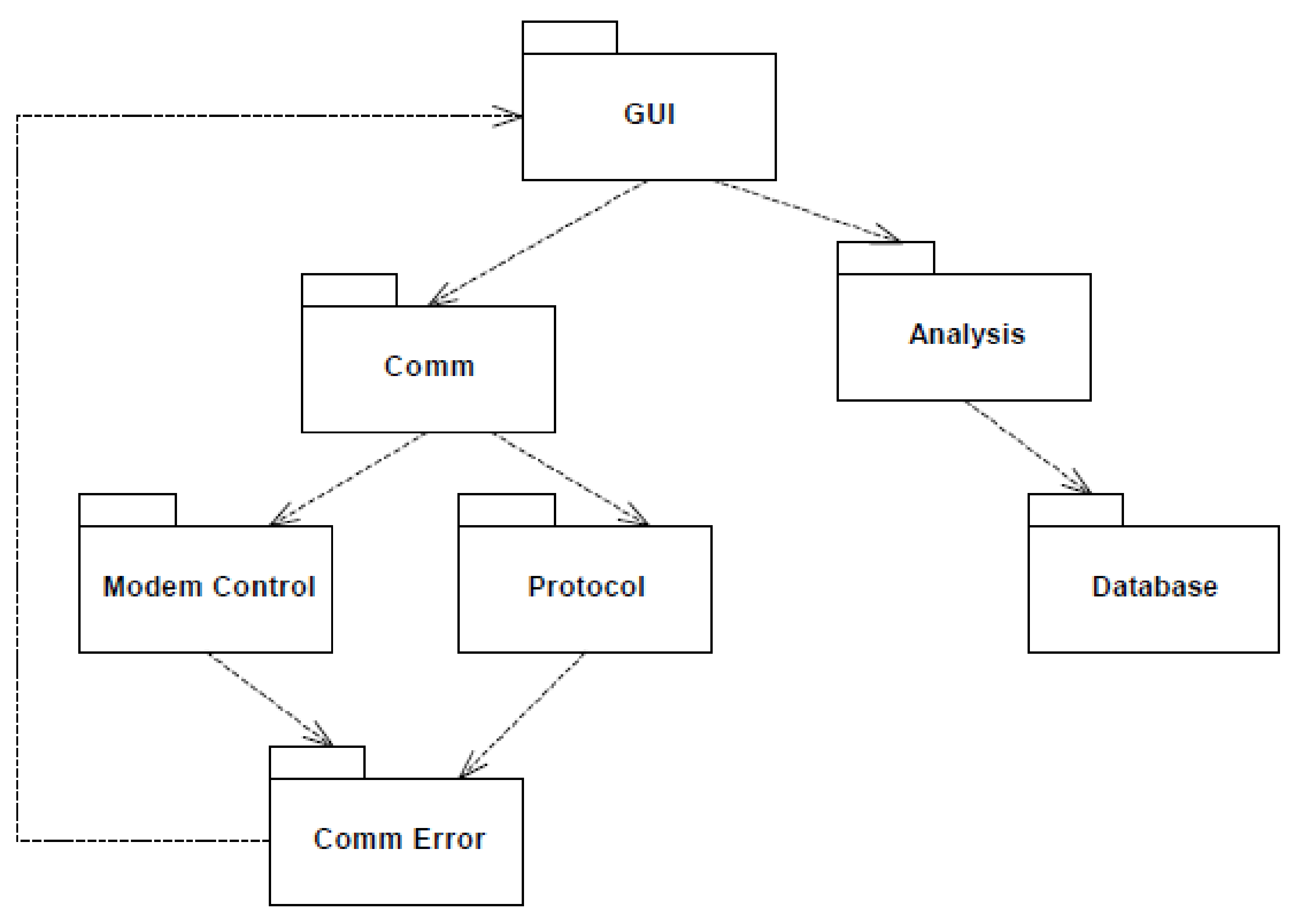


The Acyclic Dependencies Principle

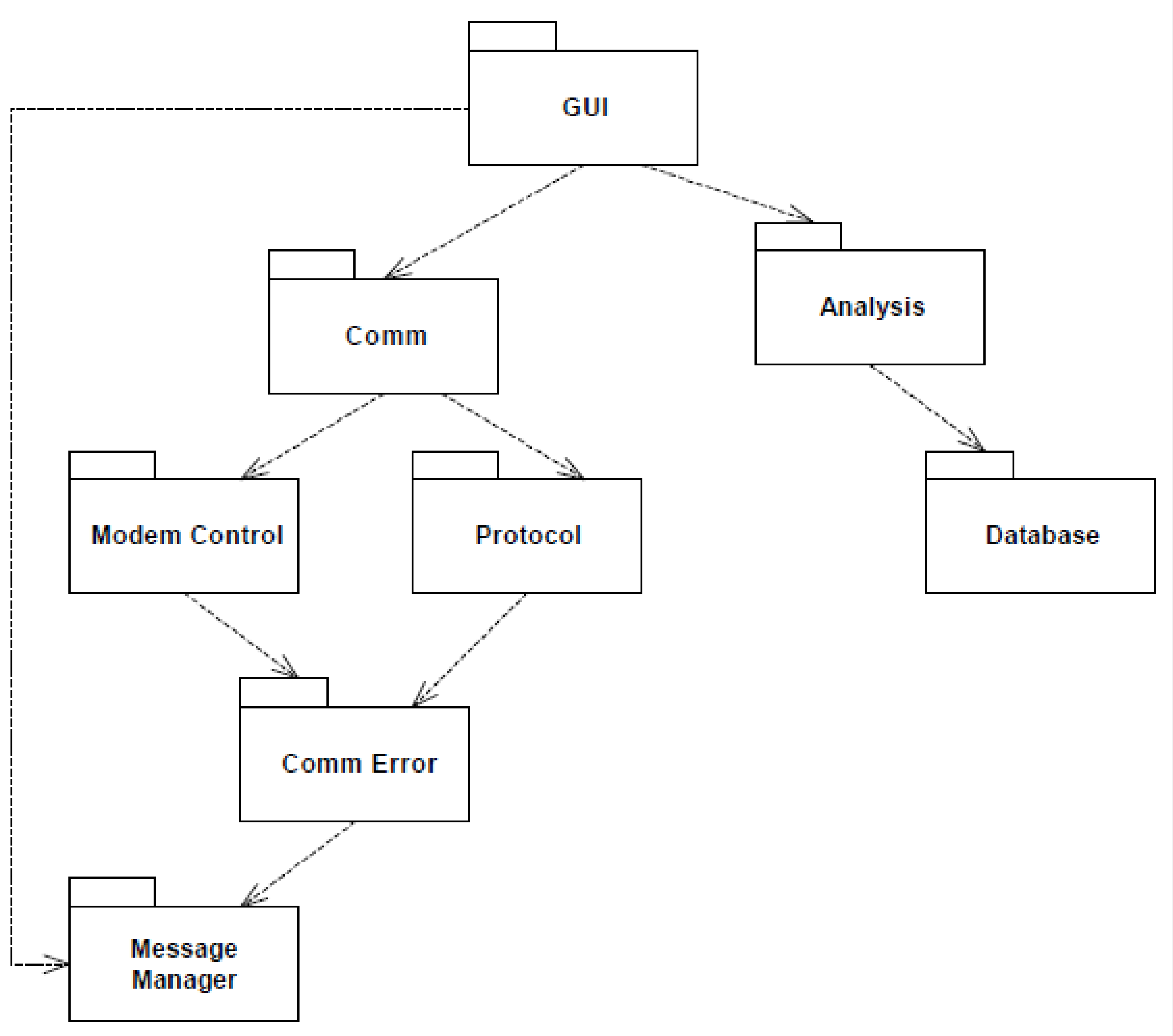
The dependencies between packages must not form cycles.

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

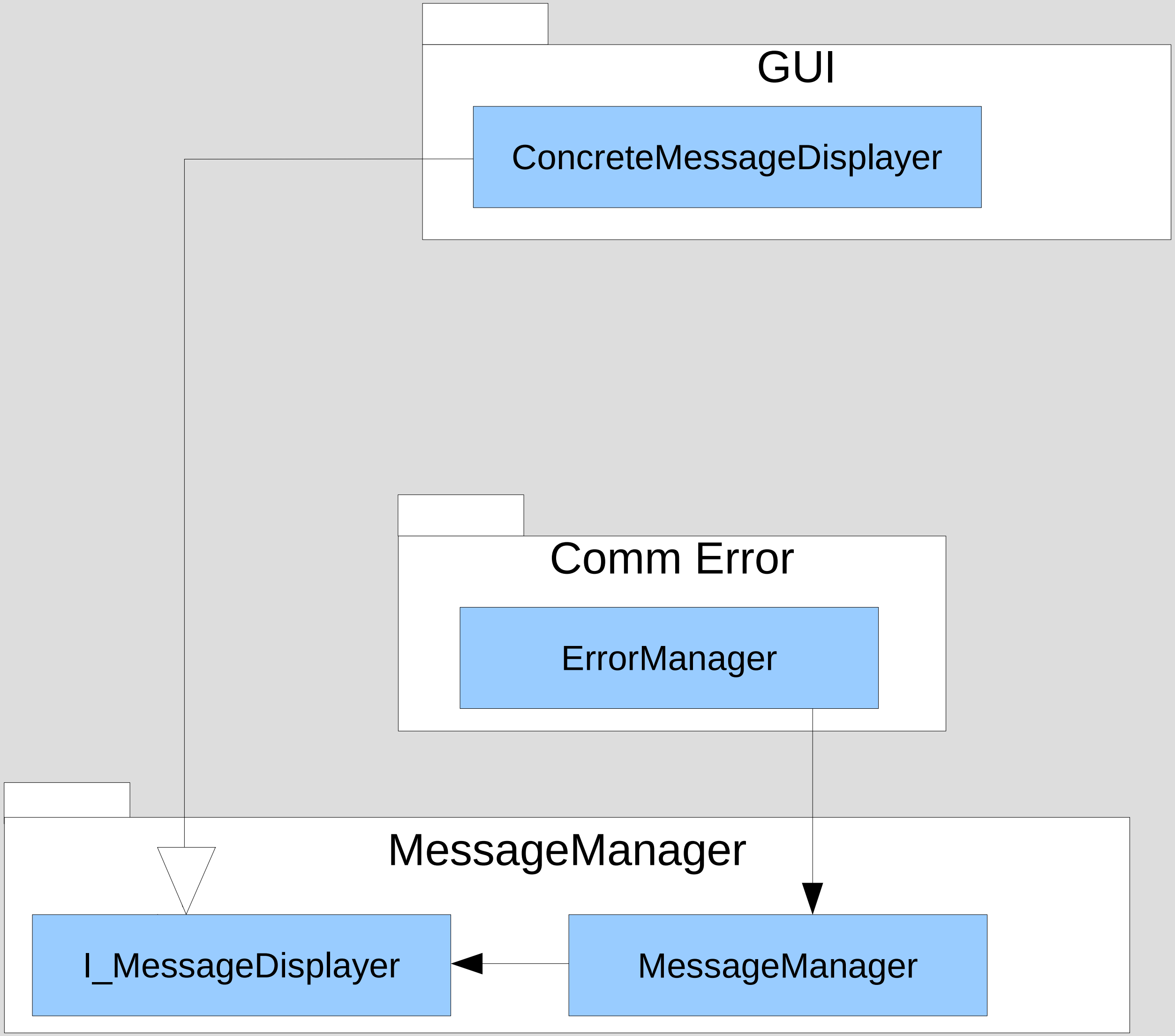




Solution: Dependency Inversion Principle



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...

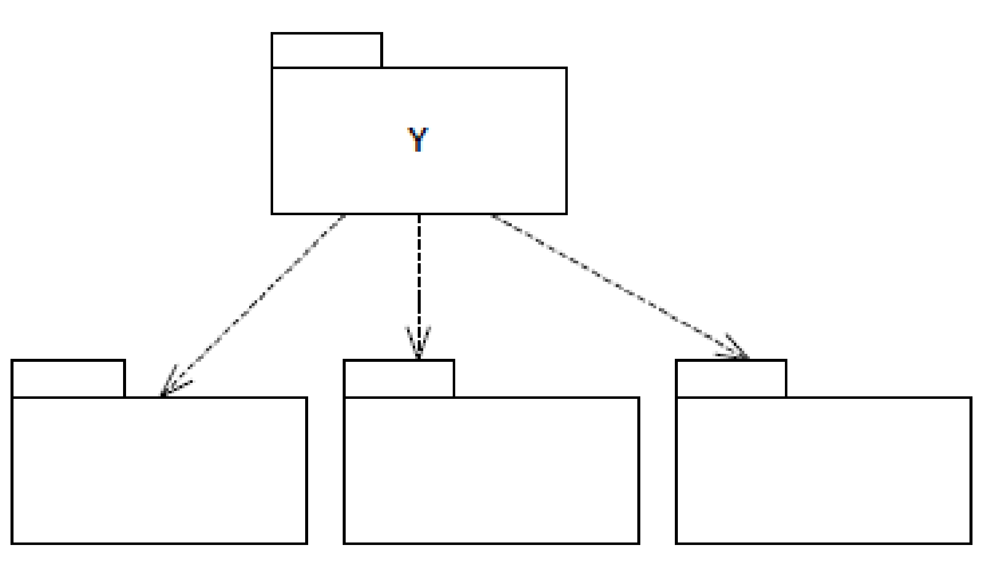
✓

My work depends on package X !

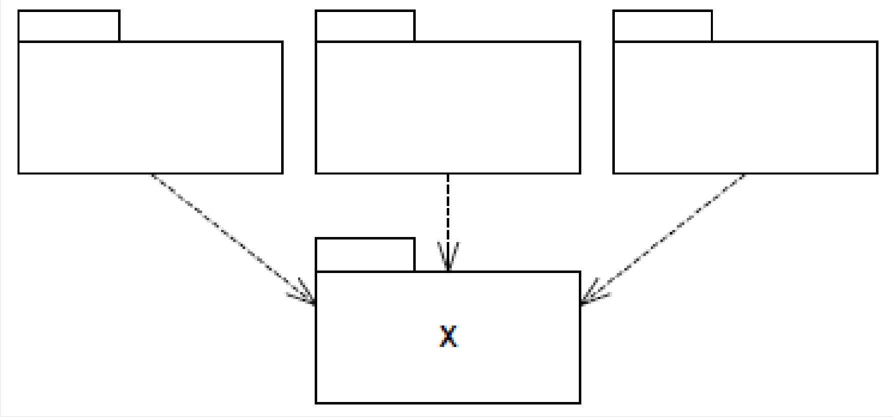
Good point. X will not Change anymore.

X is stable !

Stable / instable



Y instable

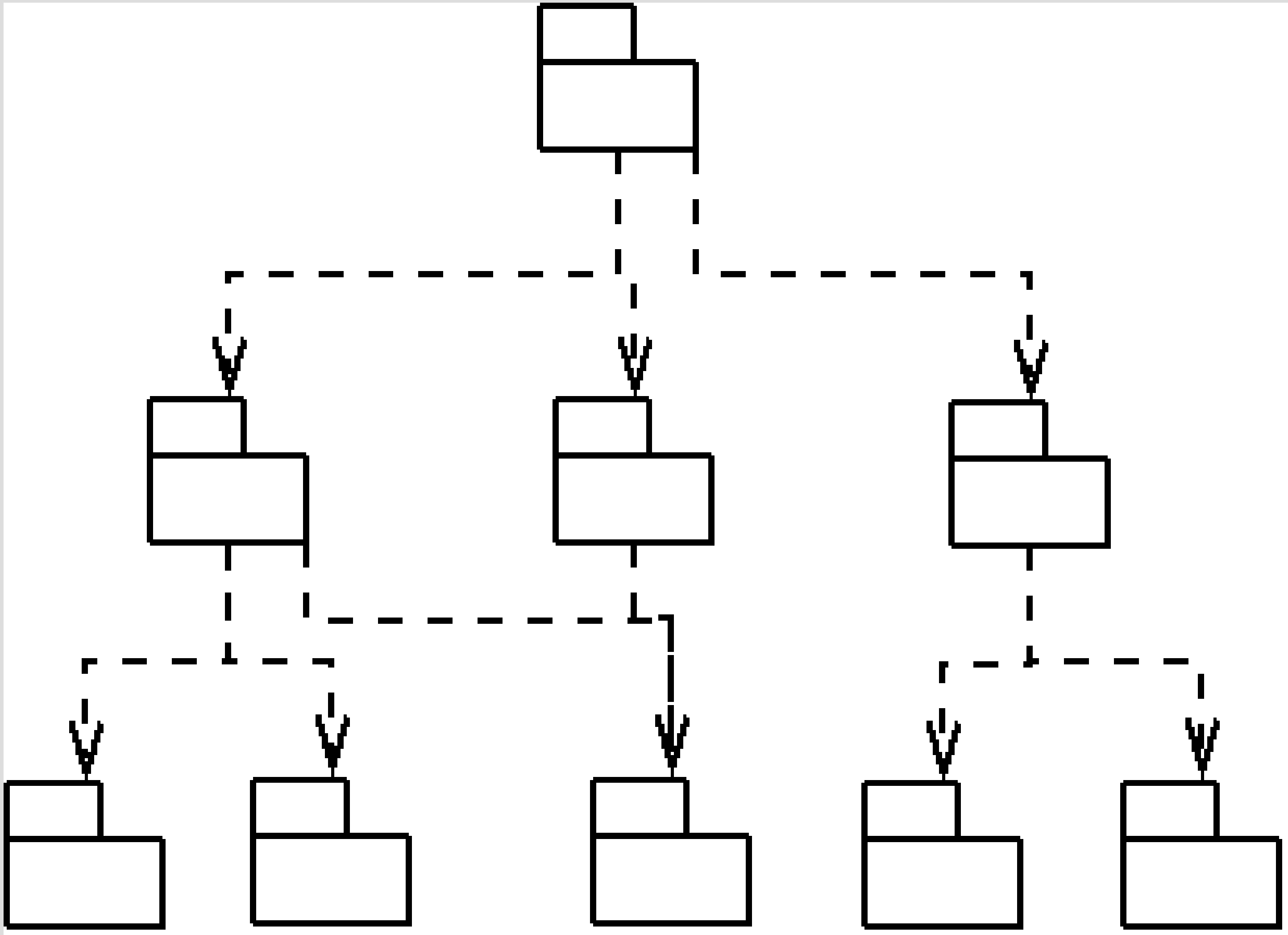


X stable

The stable abstractions principle

Stable packages should be abstract packages.

Flexible / instable

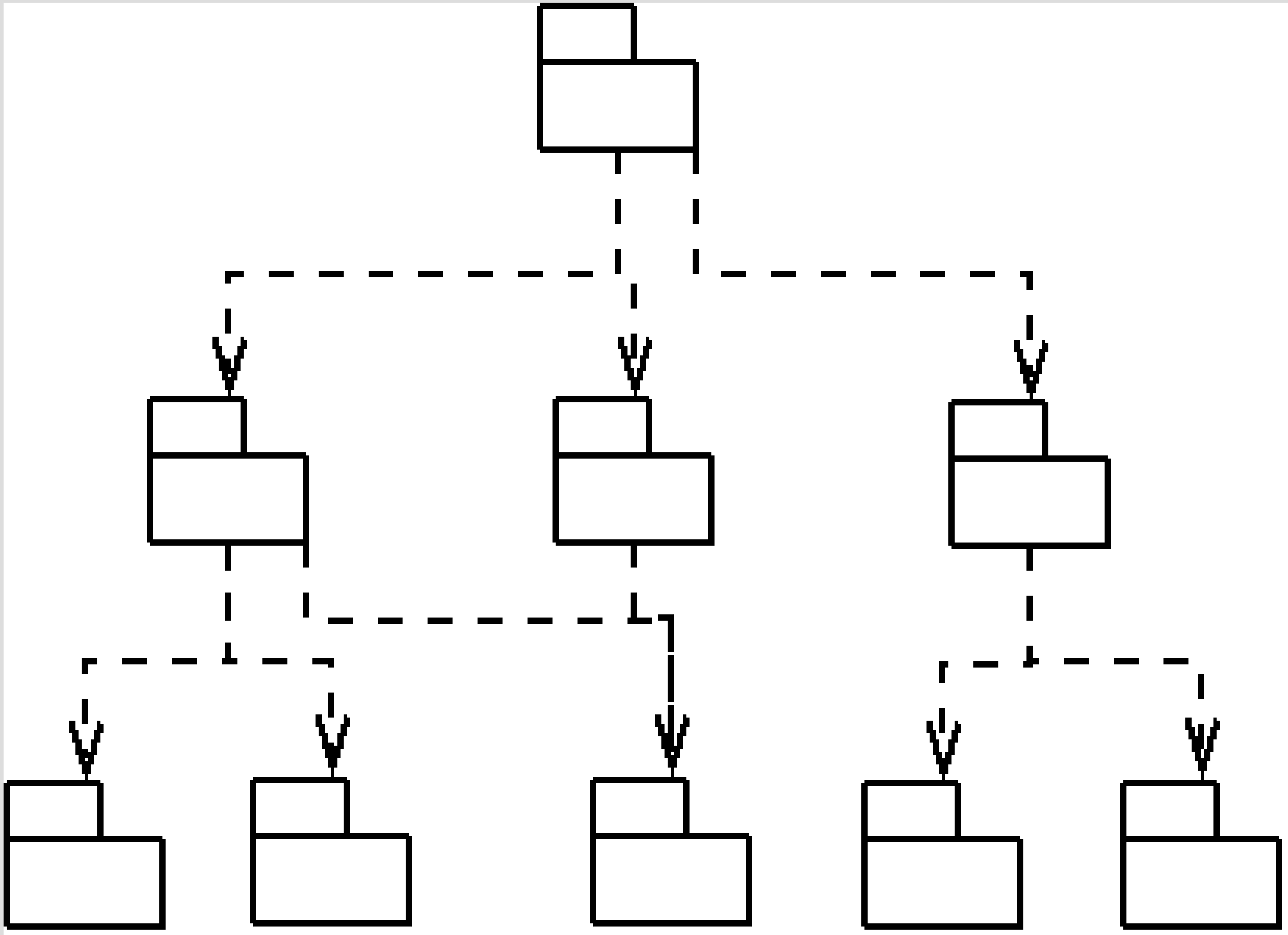


Stable

The stable abstractions principle

Stable packages should be abstract packages.

Flexible / instable

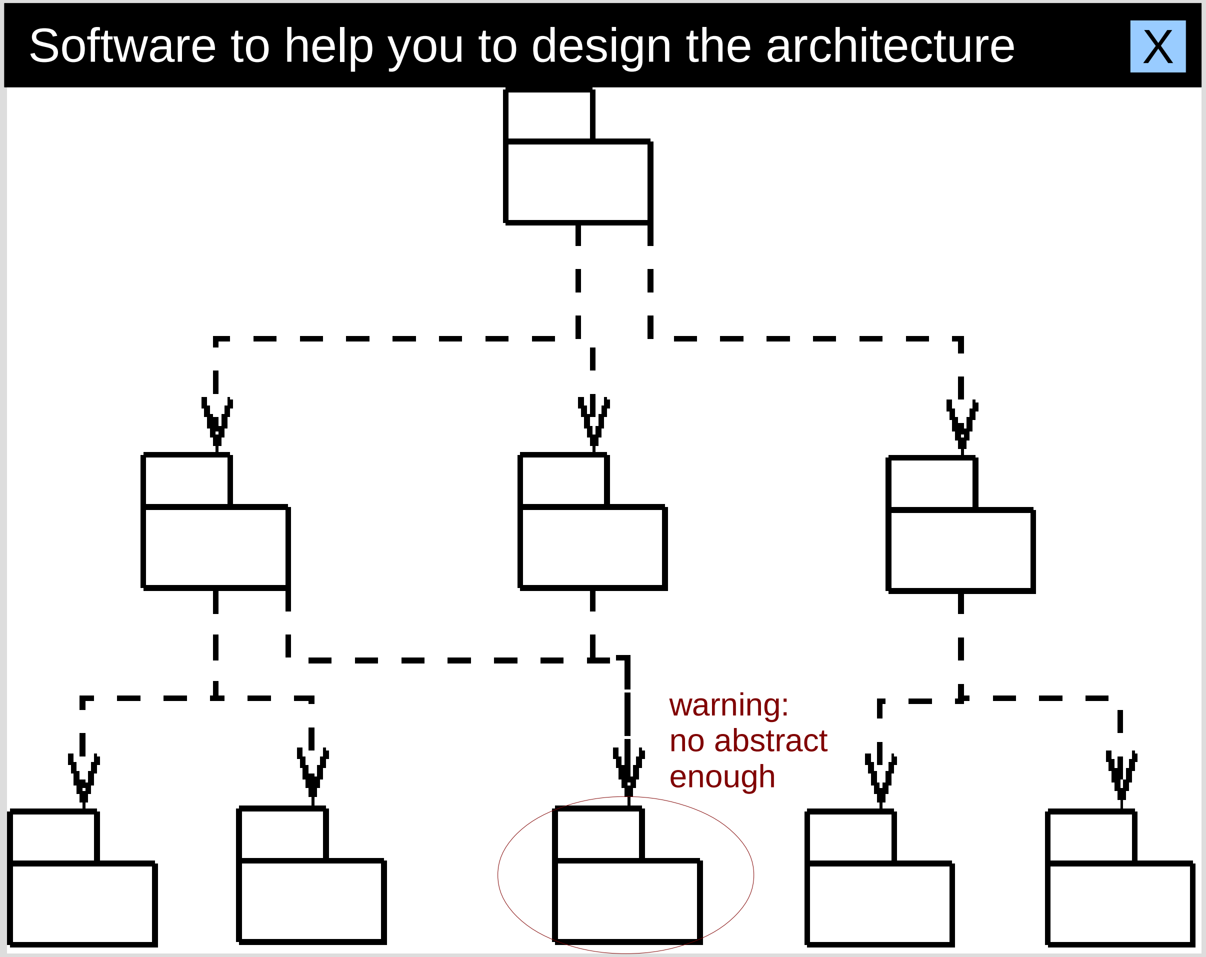


Stable
→ but we want them
flexible
→ should be abstract
in order to be
extended!

Outline

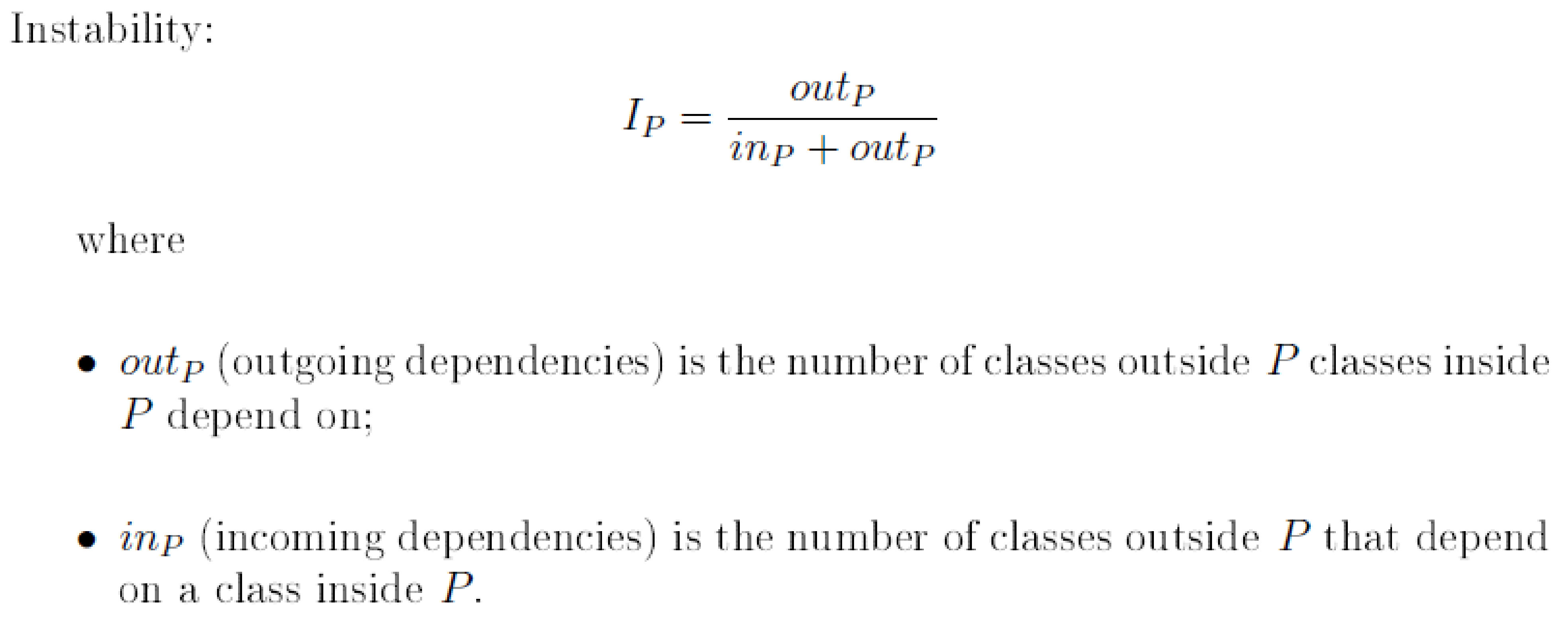
- Symptoms of rotting systems
- Principles of object oriented class design
- Principles of Package Architecture
- **Dreams**

Dream 1: Automated assistance



warning:
no abstract
enough

Measuring instability

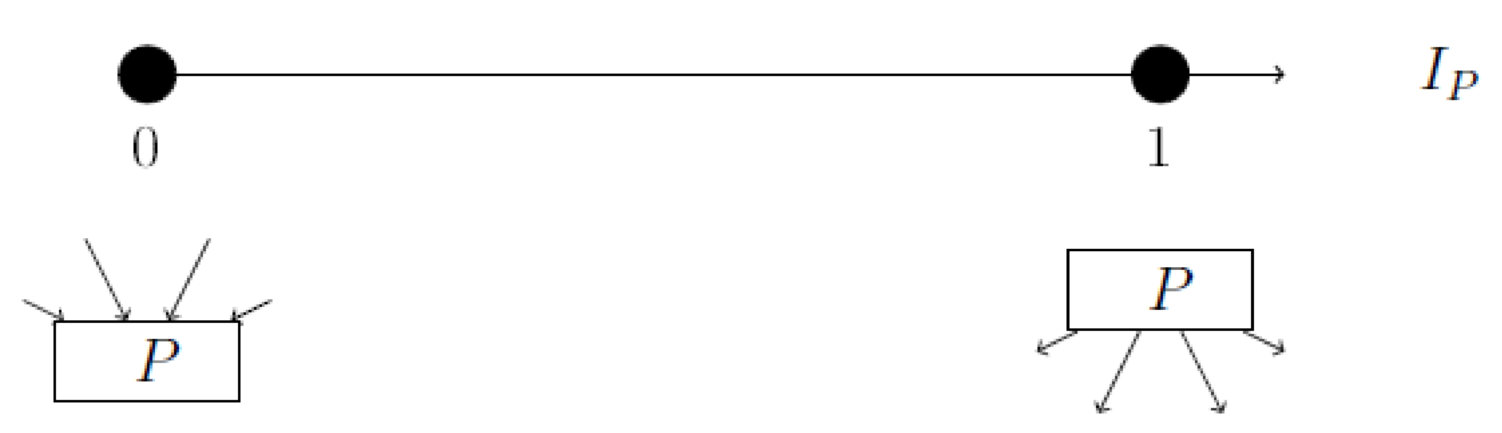


Instability:

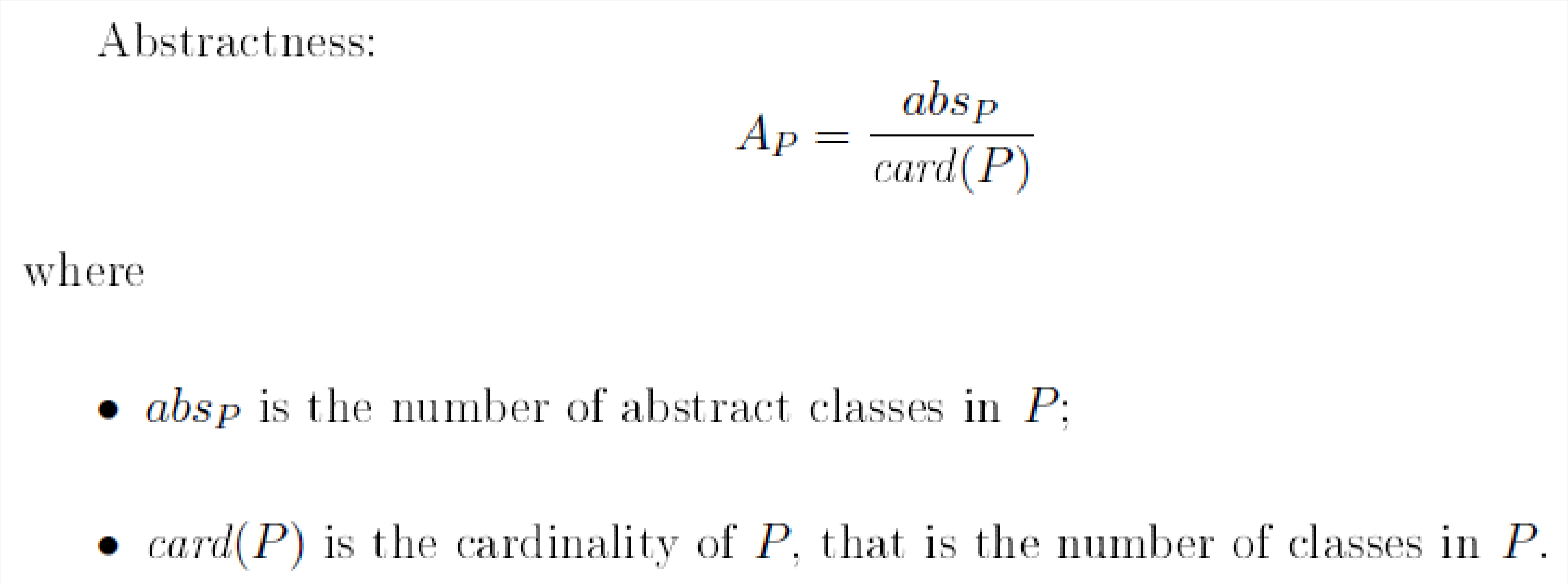
$$I_P = \frac{out_P}{in_P + out_P}$$

where

- out_P (outgoing dependencies) is the number of classes outside P classes inside P depend on;
- in_P (incoming dependencies) is the number of classes outside P that depend on a class inside P.



Measuring abstractness



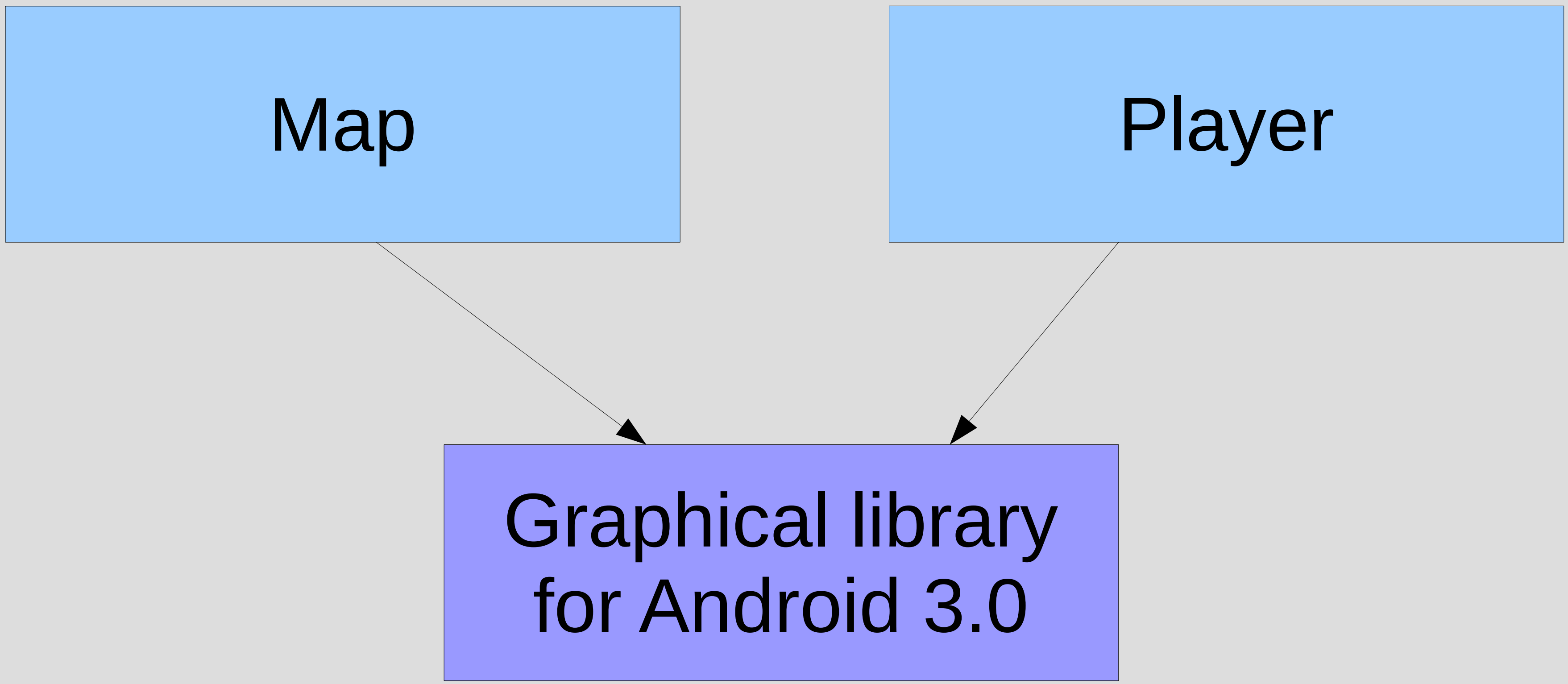
Abstractness:

$$A_P = \frac{abs_P}{card(P)}$$

where

- abs_P is the number of abstract classes in P;
- $card(P)$ is the cardinality of P, that is the number of classes in P.

The zone of pain: stable and too concrete

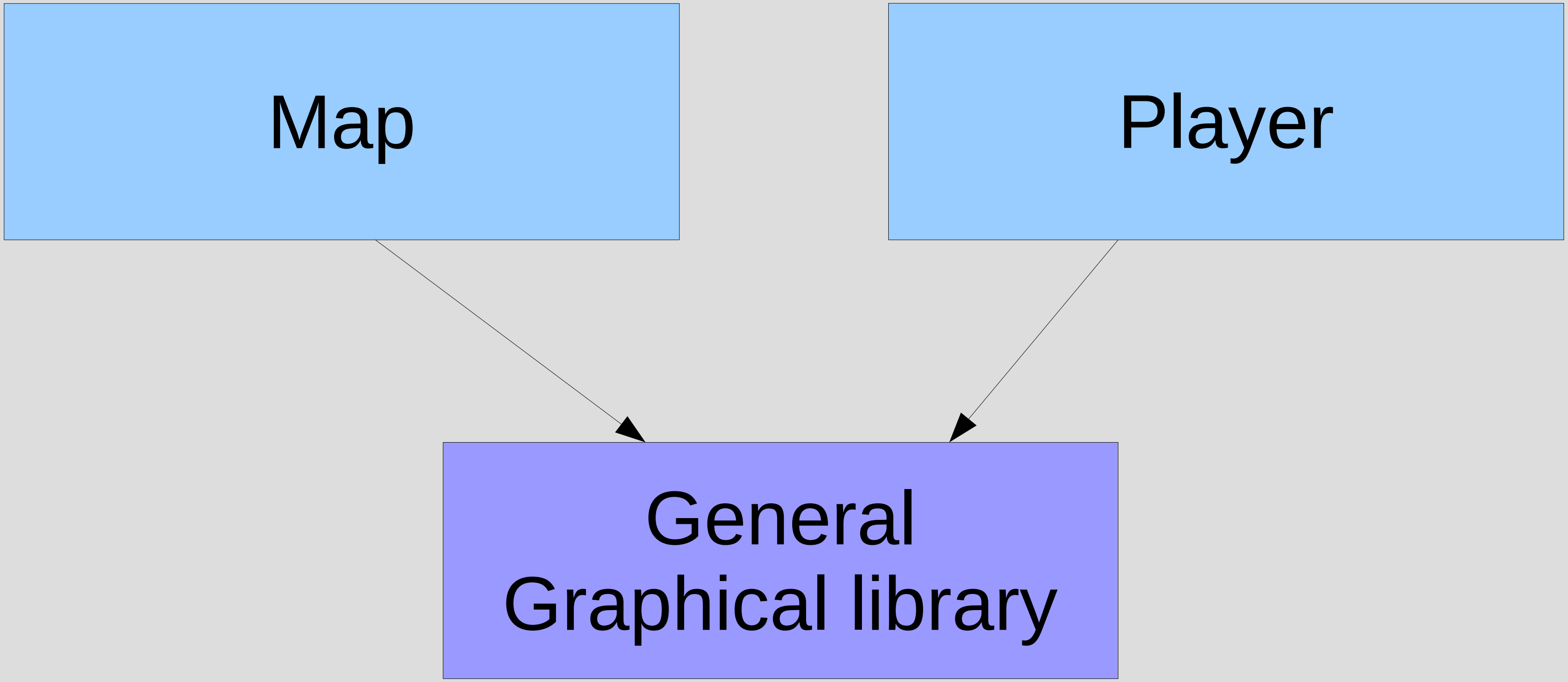


Map

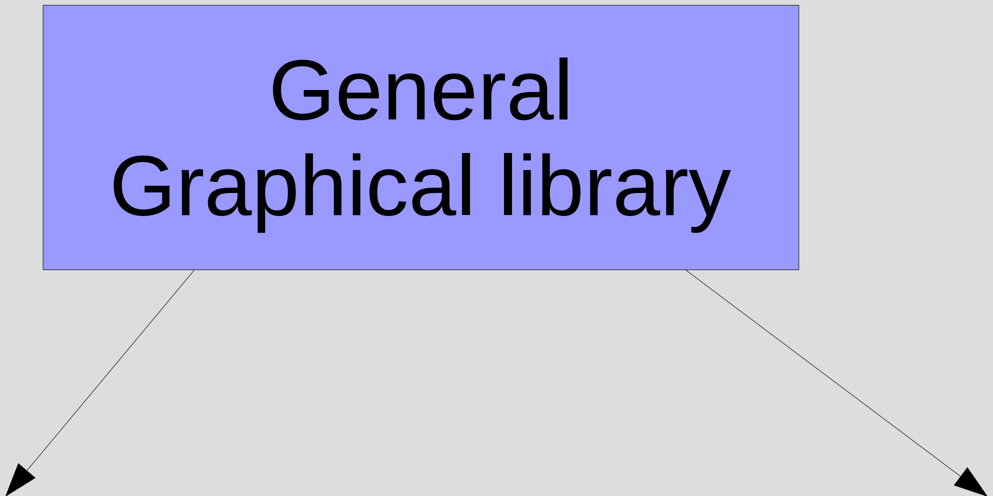
Player

Graphical library for Android 3.0

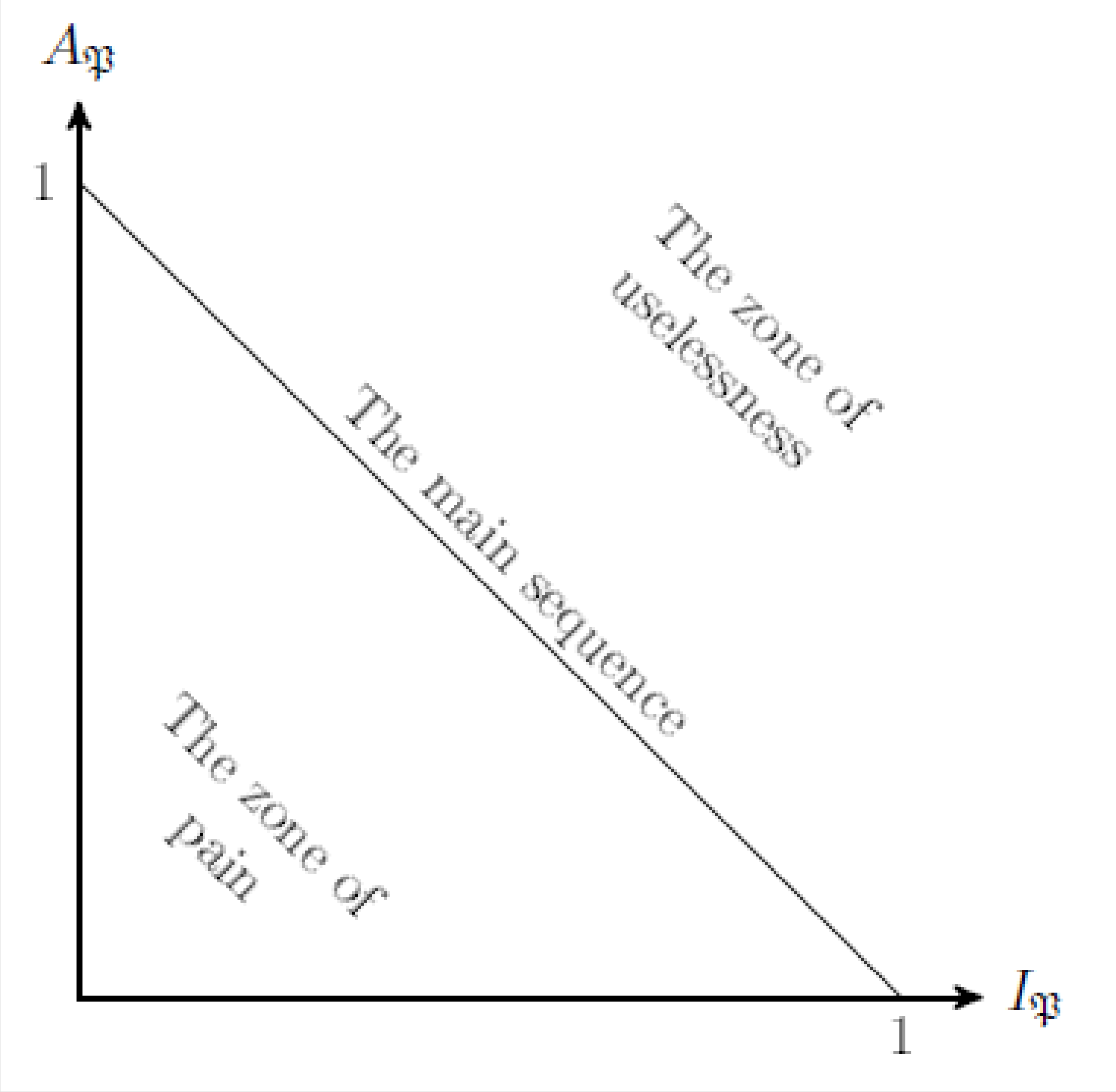
The main sequence: stability = abstractness



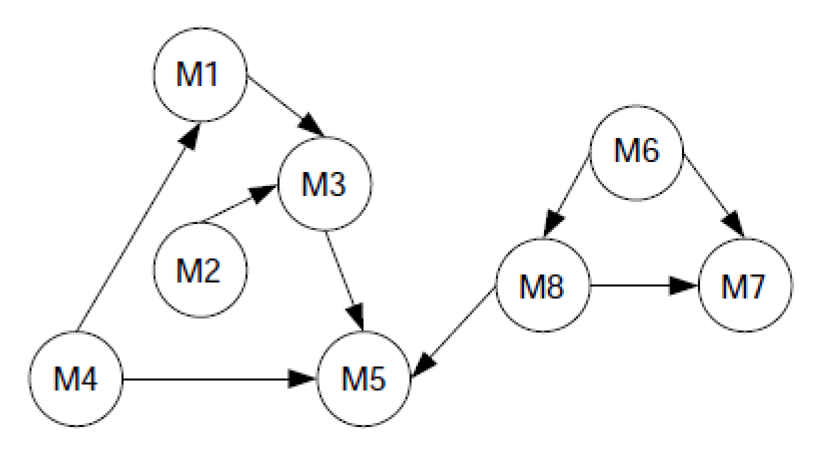
The zone of uselessness: abstract but not used!



Instability VS Abstractness

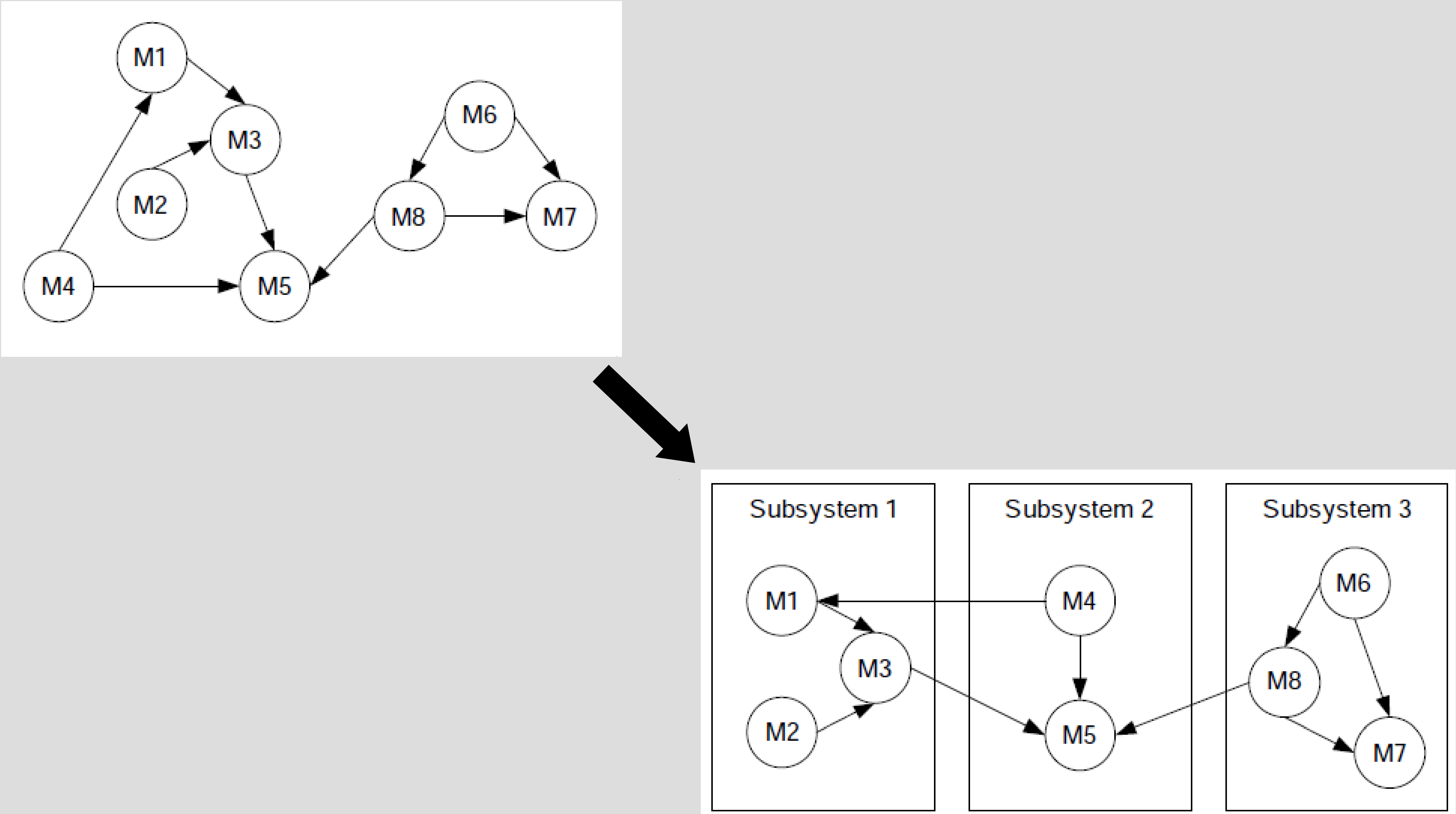


Dream 2: creating automatically the packages partition



Graph of dependencies G = (V, E)

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

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

- Measuring coupling

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

- Measuring the quality of a clustering

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

Heuristics

- Hill-climbing algorithms
- Genetic algorithms

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