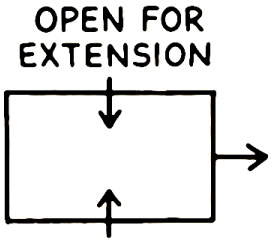
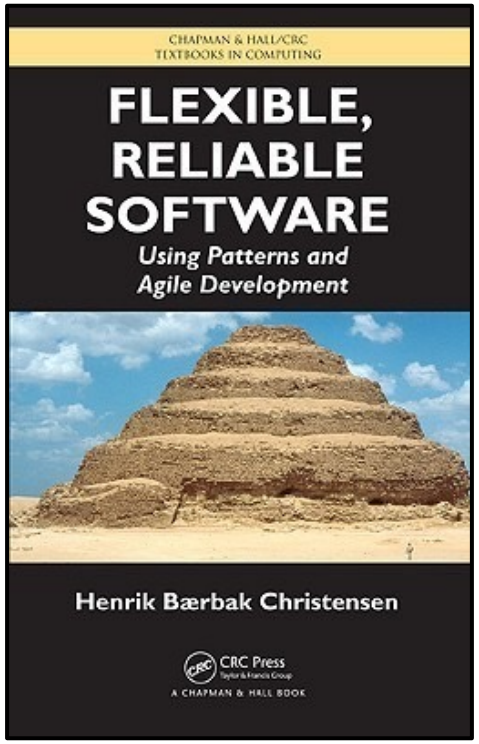
THE
OPEN-CLOSED PRINCIPLE



The Original Version



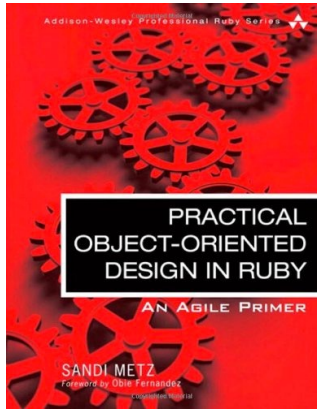
Bertrand Meyer

Henrik Christensen

deck by  @philip_schwarz FP Illuminated https://fpilluminated.org/

Sandi Metz

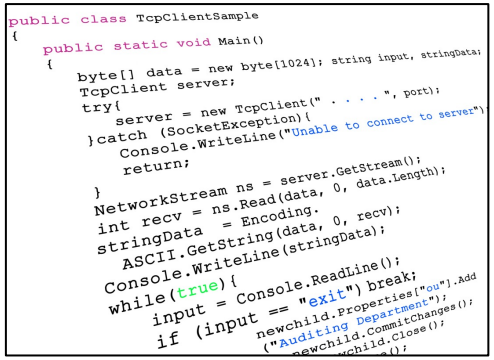
The Fundamental Target of Design



known needs of today

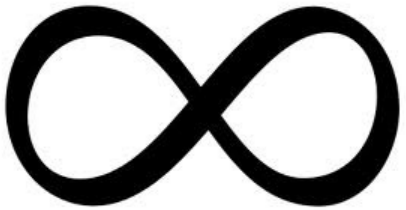
changes that will arrive in the future

Tension Between



Code needs to work today just once,

and it needs to be **easy to change forever**



today's needs

future changes

It is at this point of tension

Design Matters

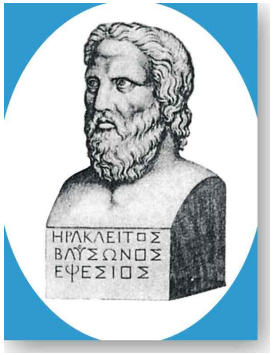
where **design matters**

design

its purpose, its entire reason for being, is to **reduce the cost of change**.

raison d'être

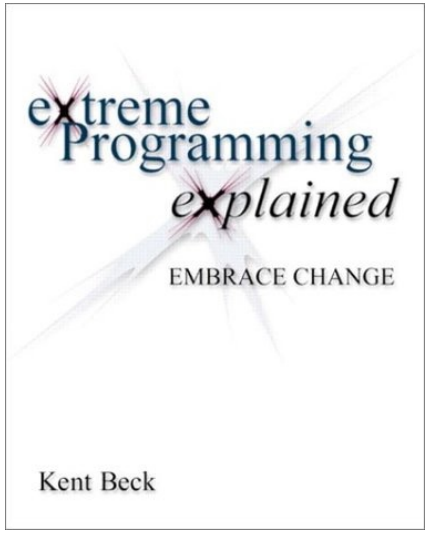
(n.) a reason for existing



Change is the only constant.

Heraclitus

Kent Beck



1999

3rd Apr 2014

@KentBeck **design is irrelevant for today. it only matters when we want to change the software...**

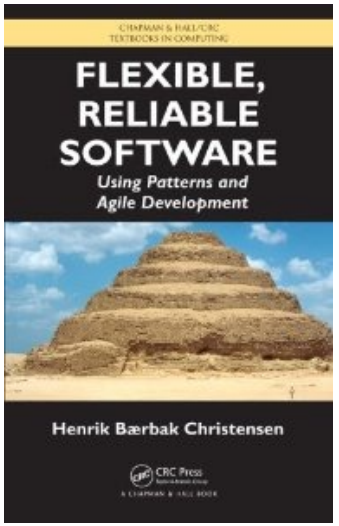
@KentBeck **change is the only reason to organize software at all...**

Change is the only constant.

Heraclitus

Henrik Christensen

adopted by agile community as a truth about software development



2010

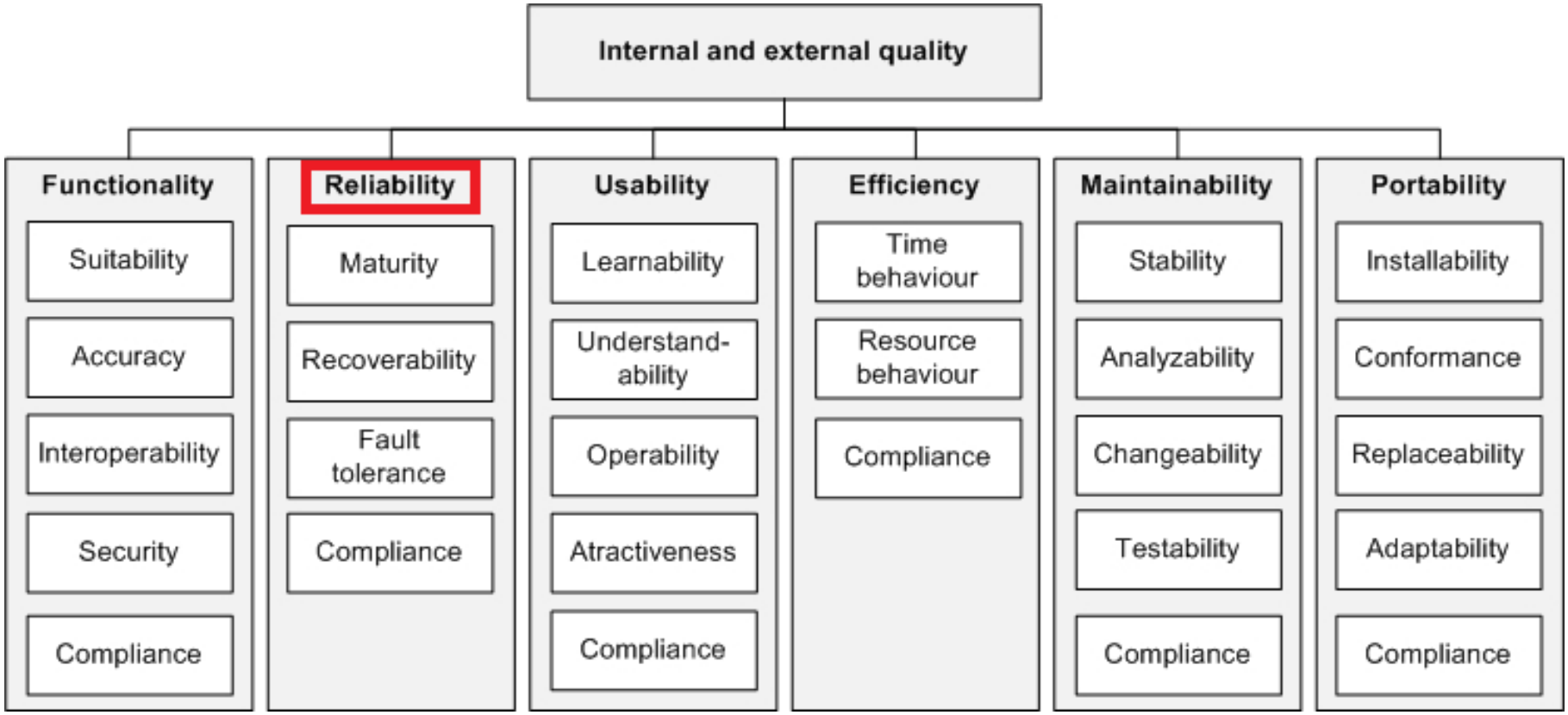
“software must be designed and developed to make it **easy to change**”

Let’s examine software qualities that enable **ease of change**

Look for definitions in ISO 9126

Projects sometimes **fail** due to not having any clear **definitions of "success”**

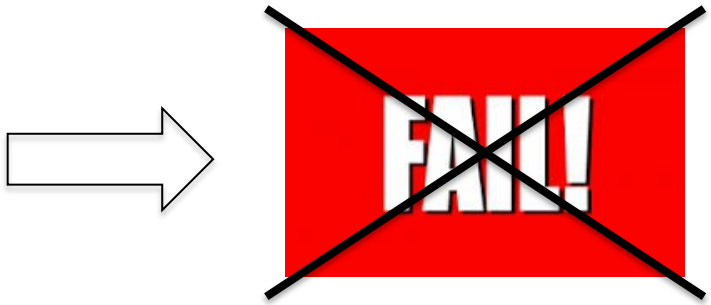
This standard tries to develop **a common understanding of project objectives and goals,** e.g. software qualities



Software is **Reliabile** if it can **maintain a specific level of performance** under specific usage conditions

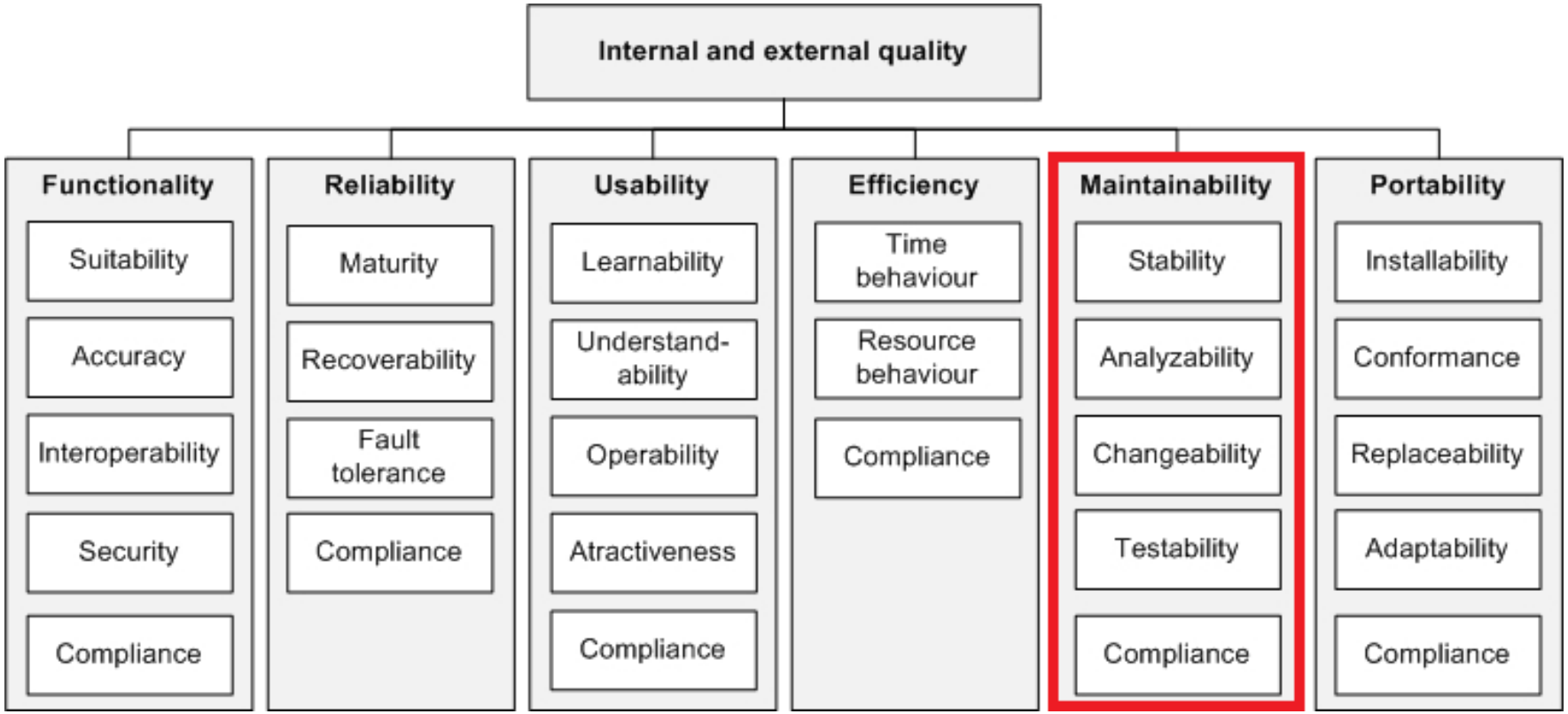
in our
setting…

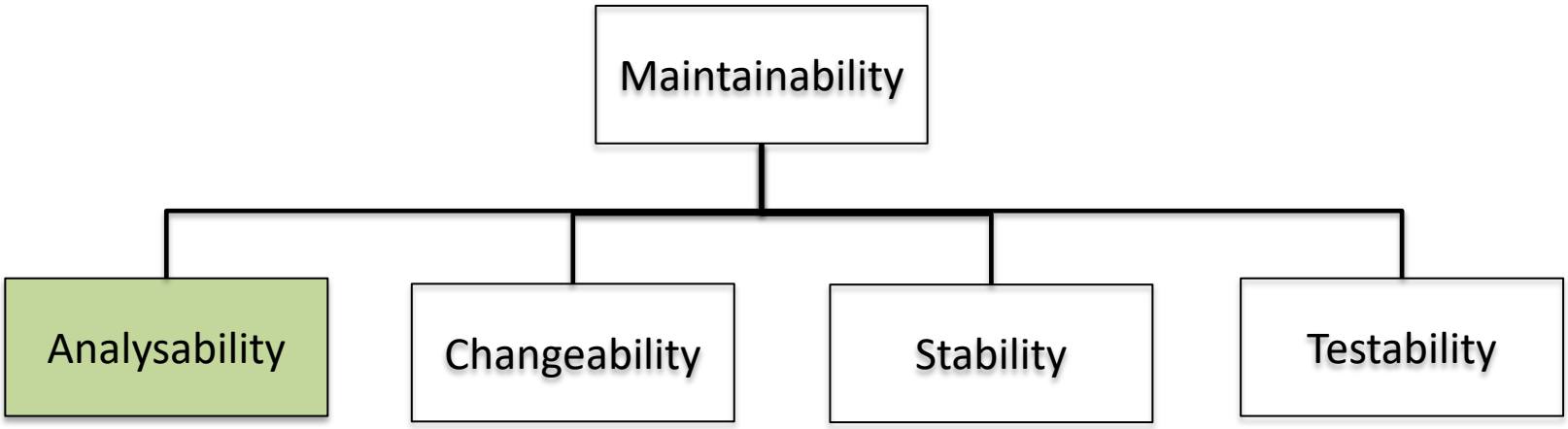
Software is **Reliabile** if it can **perform the required functions without failing**



In particular, we are interested in
preserving reliability in the face of change

“maintainability is composed of several, finer grained, qualities”

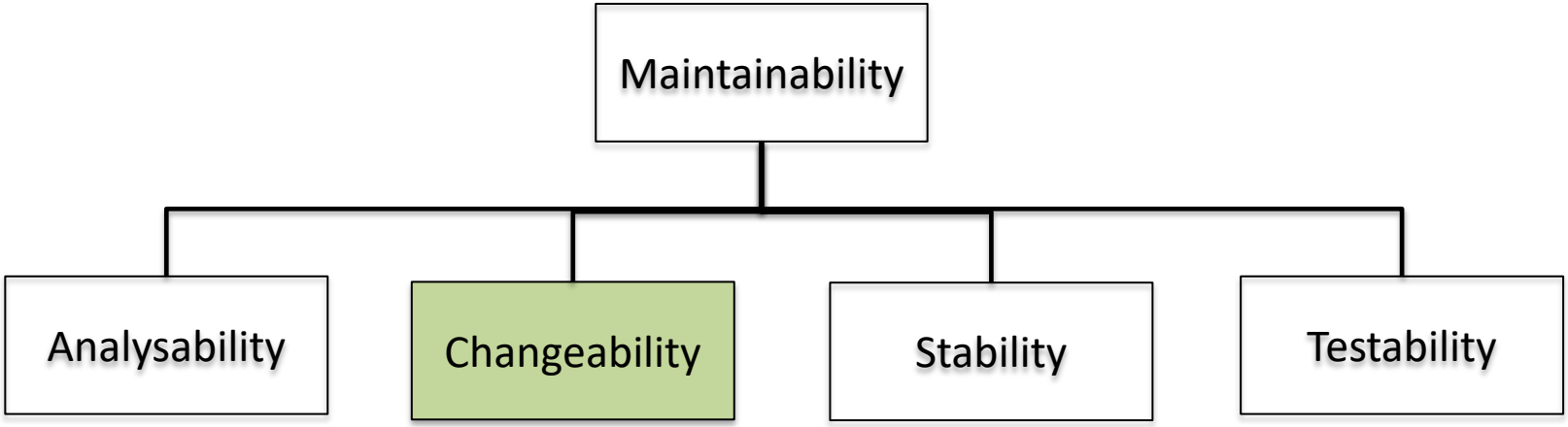




Software is **Analysable** if you can

- diagnose it for
  - deficiencies
  - causes of failure
- identify the parts to be modified

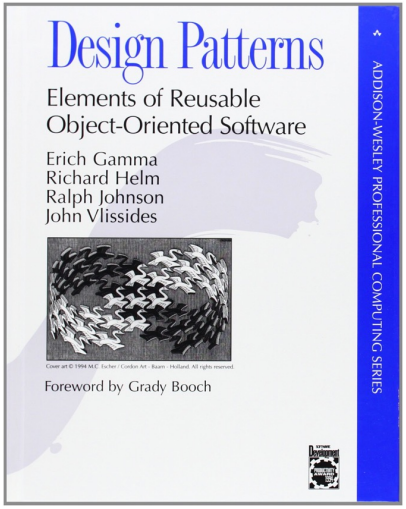
"**Analysability** is basically the ability to ***understand*** software"

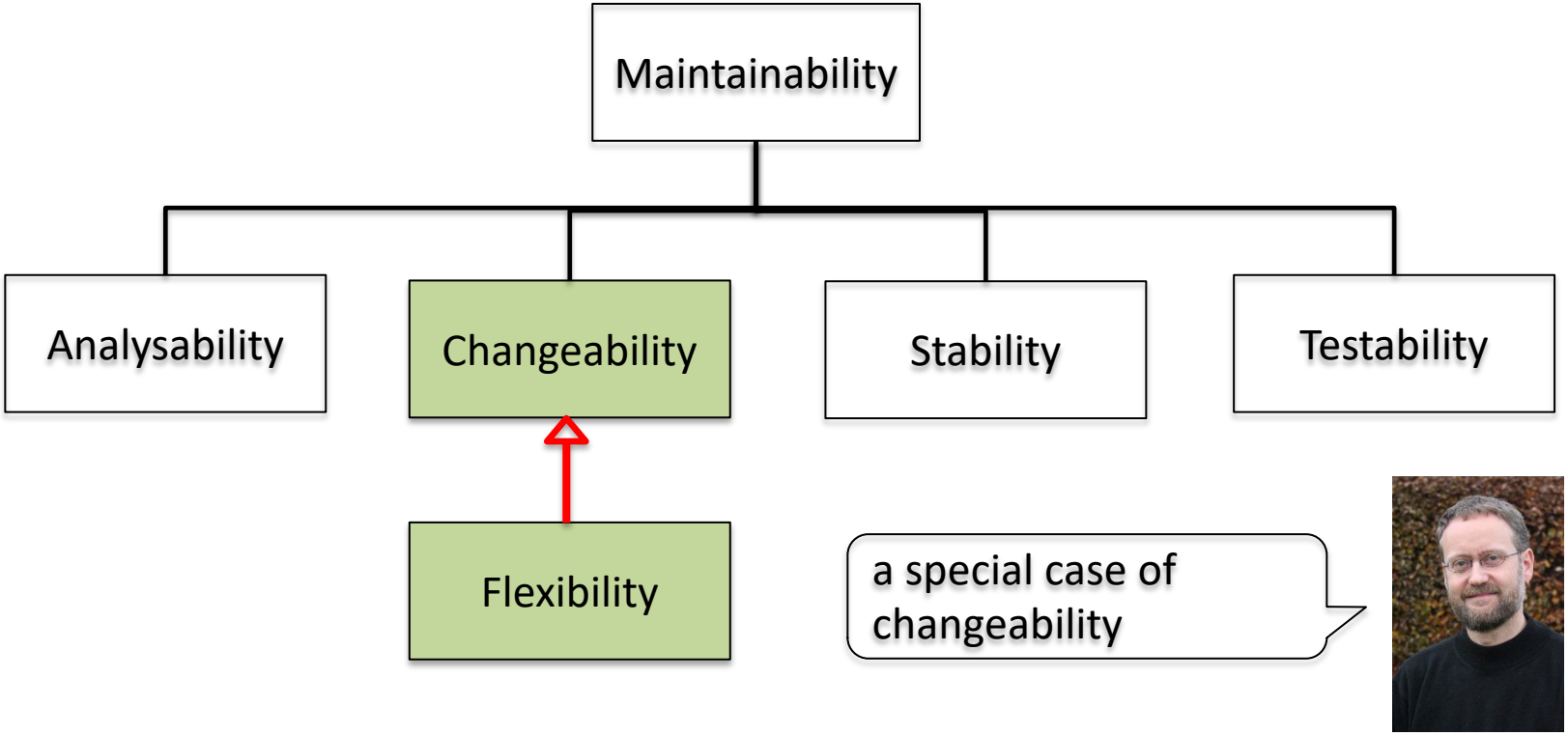


Software is changeable if it allows you to

- implement a specific modification
- add, modify, or enhance a feature
  at a reasonable cost

"almost all **Design Patterns** are geared towards **increasing design's changeability**"





"Software is flexible if you can **add/enhance functionality** **purely by adding software** units and specifically **not by modifying existing software** units"

Changeability

Changeability is a desirable quality,
but it relies on
Change by Modification

Change by Modification

Behavioural changes that are introduced by **modifying** existing production code

Modifications carry the **risk of introducing defects**, and the necessary cost of avoiding them: testing, code reviewing, etc.

"The less I ever modify a class, the higher the **probability that it will remain free of defects**"

Contrast

Change by Modification

Behavioural changes that are introduced by **modifying** existing production code

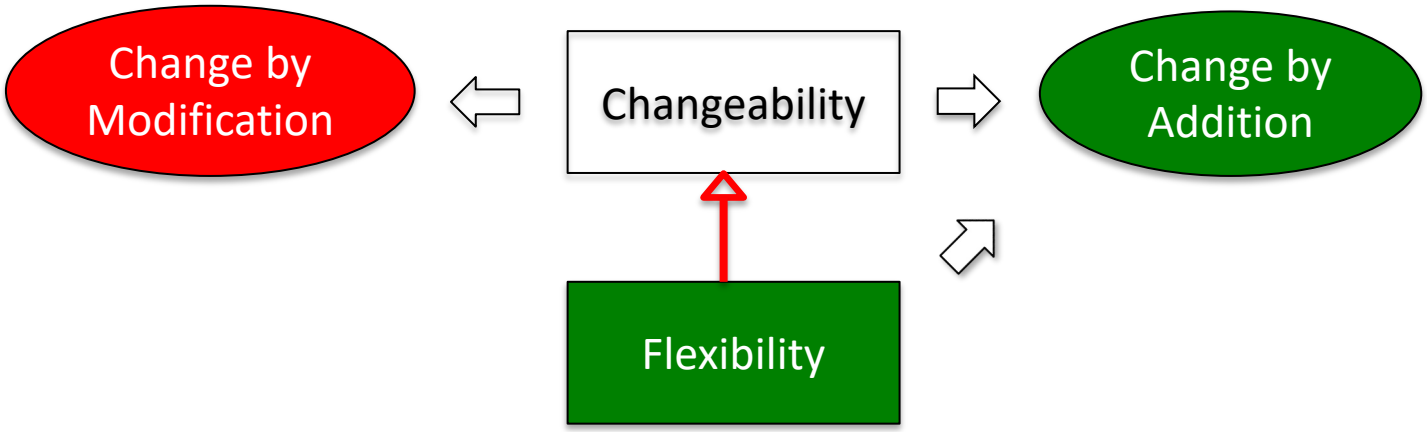
with

Change by Addition

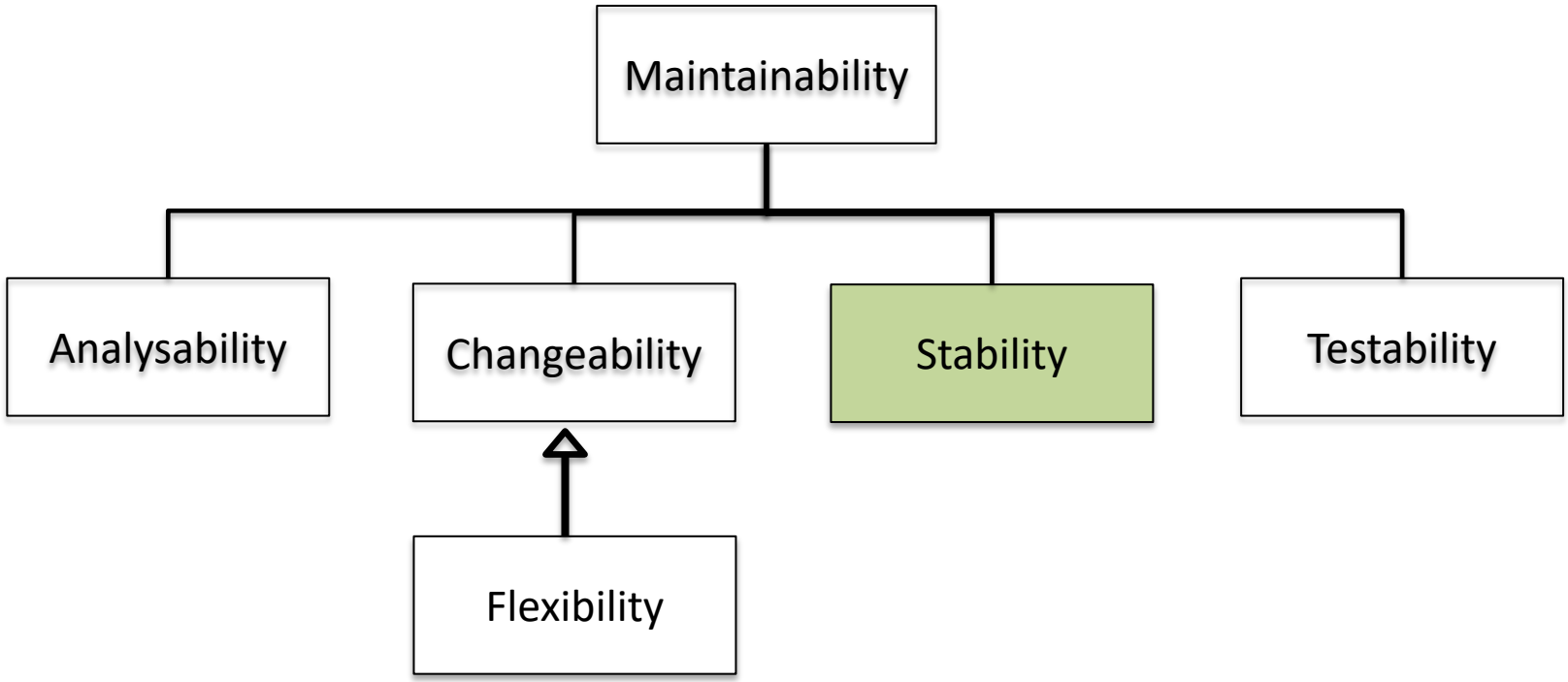
Behavioural changes that are introduced by **adding** new production code **instead of modifying** existing code

Change by Addition **avoids (risky) modifications** altogether

Changeability **does not take a stand point** with regards to the way a specified modification is implemented



In contrast, Flexibility **does take this stand point** and **requires that no modifications are made**

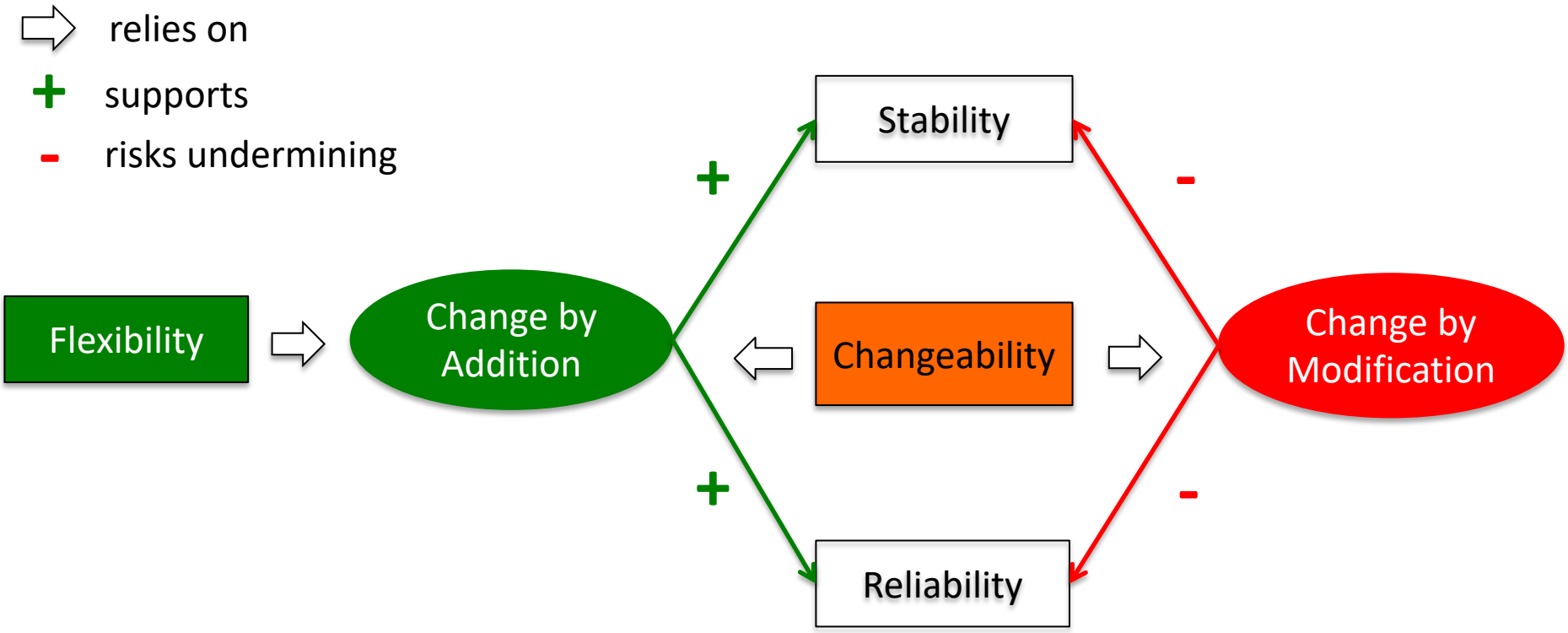


Software is **Stable** if it avoids unexpected effects when modified

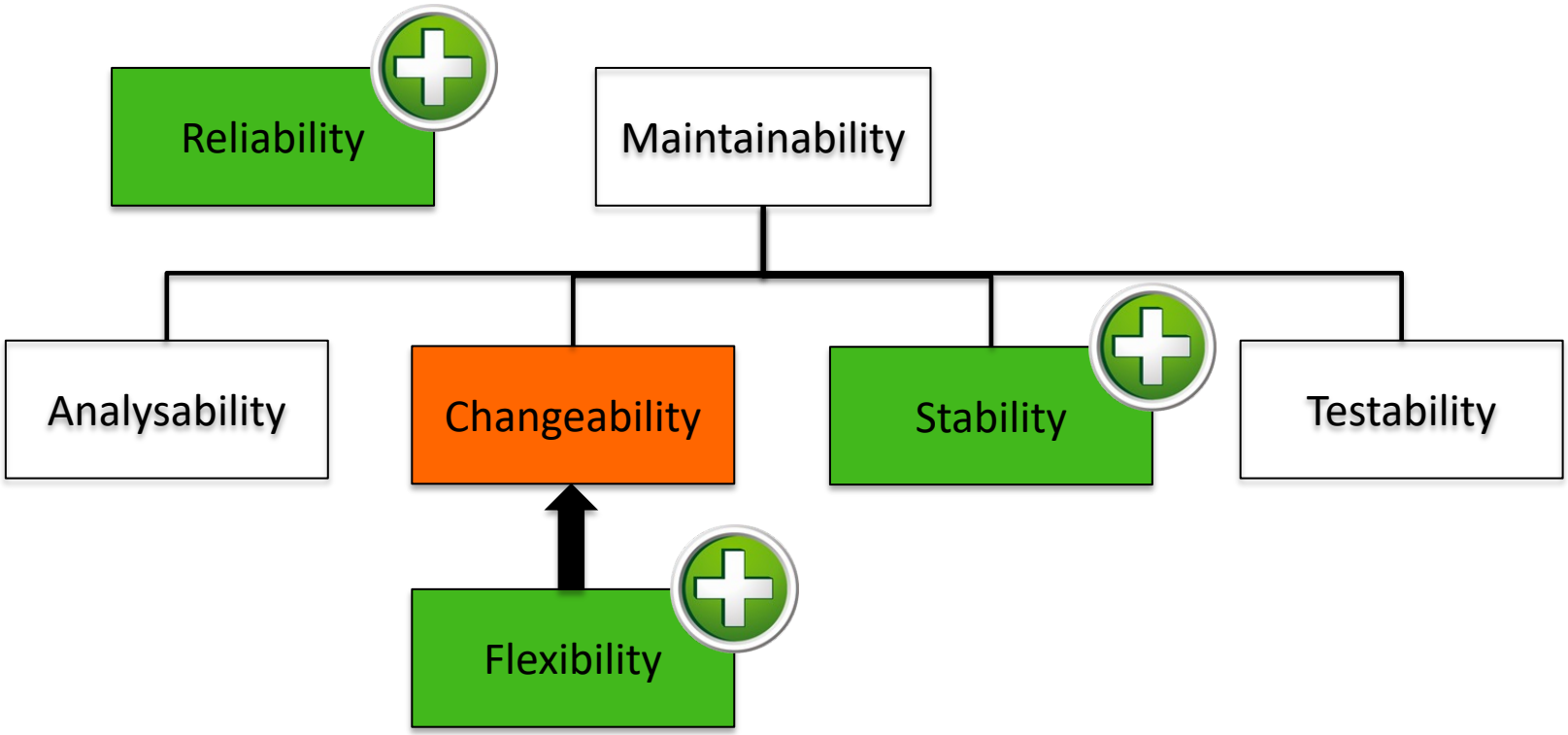
"any change to existing software carries a risk of introducing defects"

"I advocate the practice to **avoid modifying existing code** but preferably **add features or modify existing ones by other means**"

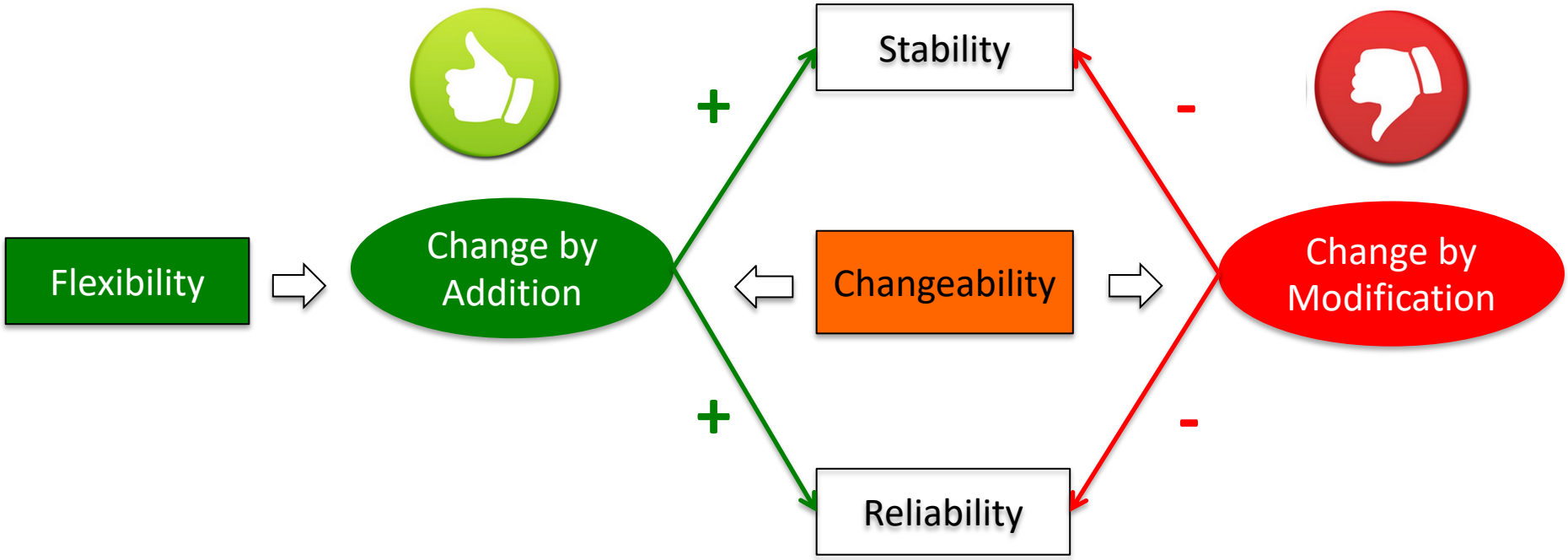
Summary


relies on
+ supports
- risks undermining
Flexibility
Change by Addition
Stability
Changeability
Reliability
Change by Modification
+
+
-
-

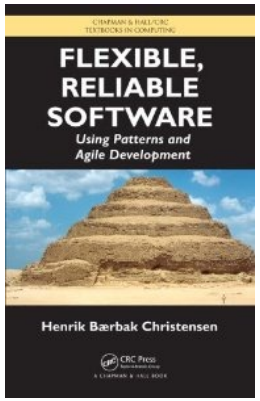
Question: how do we promote flexibility, reliability and stability in our software?



Answer: we favour '**Change by Addition**' over '**Change by Modification**'



if you require that s/w can adapt to changing requirements



Christensen

without modifying the production code

then you need to employ a **SPECIAL set of design and programming techniques** as well as adopt a **SPECIAL mindset**

I cover **techniques that focus on flexibility**

How do we achieve Change by Addition ?

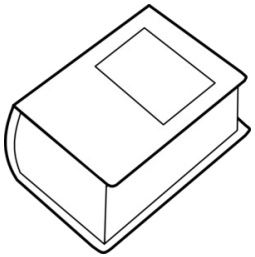
"Another way of characterising **Change by Addition** that you may come across is the **Open Closed Principle**"

1988

Bertrand Meyer

Object Oriented
Software Construction

The most comprehensive, definitive OO reference ever published



The Open-Closed Principle

Modules should be both open and closed

Isn’t there a **contradiction** between the two terms?

Modules should be both **OPEN** and **CLOSED**

$$\neg(p \wedge \neg p)$$

The **contradiction** is only **apparent**

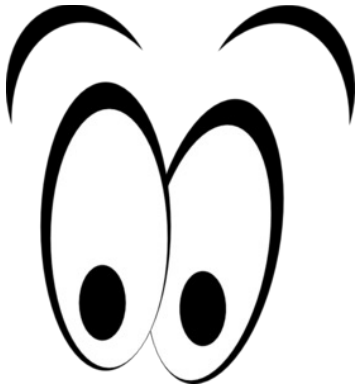
It is a

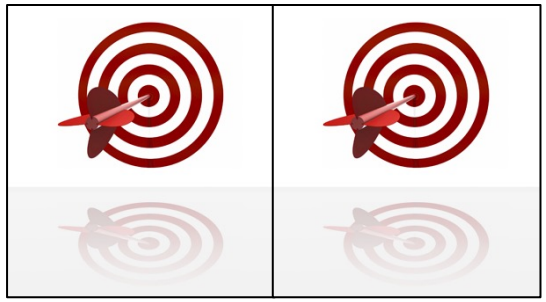
PARADOX:
a statement or proposition that seems self-contradictory or absurd but in reality expresses a possible truth.

The two terms correspond to **goals of a different nature**

e.g. a Dutch door can be both OPEN and CLOSED

So let’s look at the goals of the OCP



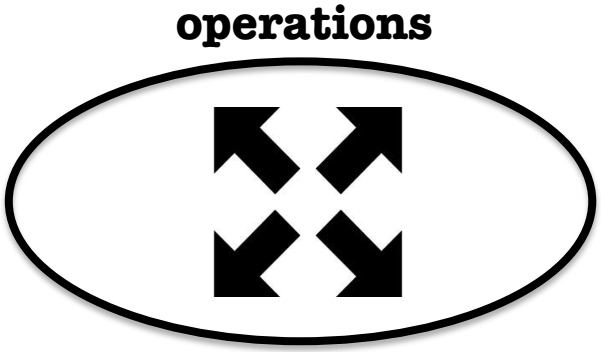


OCP

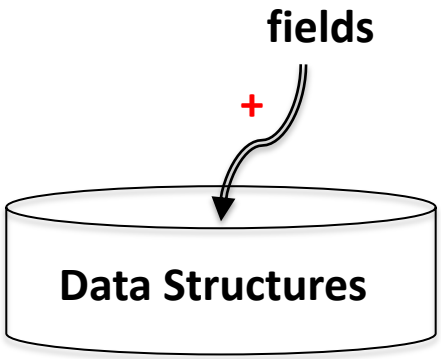
A module is said to be **OPEN**

if it is still **available for extension**

e.g. it should be possible to **expand** its set of **operations**



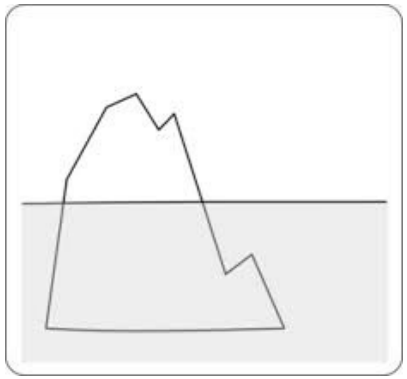
or **add** **fields** to its **data structures**



A module is said to be **CLOSED**

If it is **available for use** by other modules

1



public part

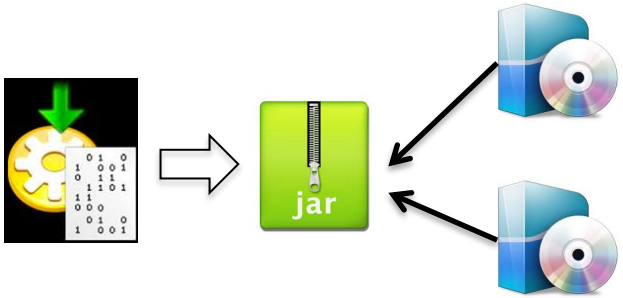
interface

secret part

Has a well-defined, **stable** description
(its **interface** – in information hiding sense)

2

can be compiled, stored in a library, and made **available for clients to use**



3

in the case of a design or specification module:

- **approved**
- **baselined** in version control
- its **interface published** for benefit of other module authors

Recap – A module is…

OPEN

if it is still **available for extension**

CLOSED

If it is **available for use** by other modules

The need for 

ness

It is impossible to foresee all the elements that a module will need in its lifetime





so developers **wish to keep the module open for as long as possible**

so that they can address changes, and extensions

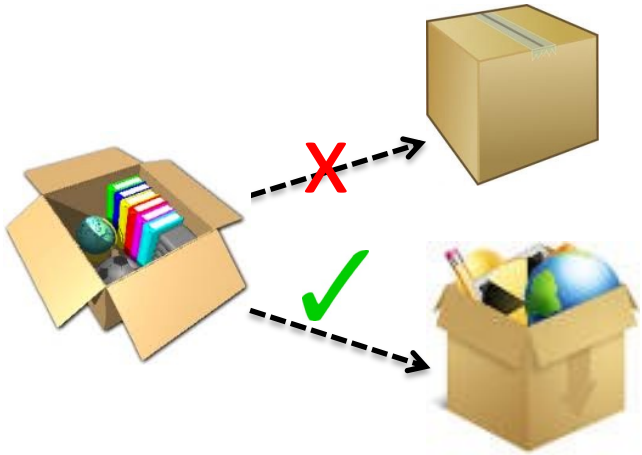

by changing elements or adding new elements

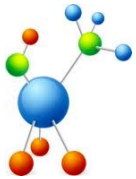
The need for 

ness

but it is also necessary to close modules

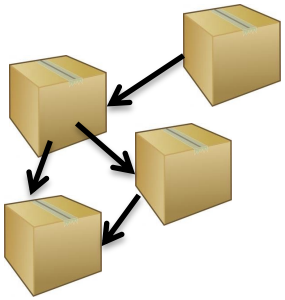
if a module is never closed until it is certain that it contains all the needed features





then multi-module s/w can never reach completion

in a system consisting of many modules, most modules will depend on some others





every developer would always be waiting for completion of another developer's job

Modules should be both open and closed

We want modules to be both 

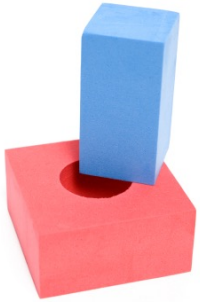
for extension and 

for modification

with **traditional techniques**

the two goals of openness and closedness

are **incompatible**

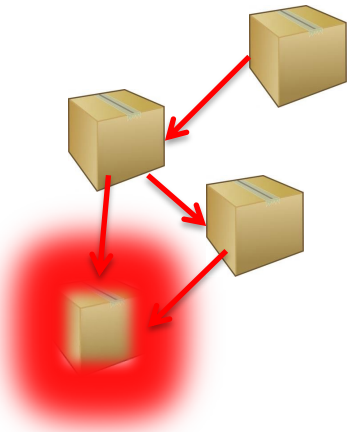


either you keep a module open

and others cannot use it yet

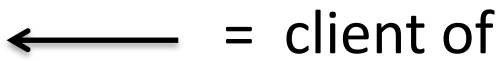
or you close it

and any change or extension can trigger a **painful chain reaction of changes**

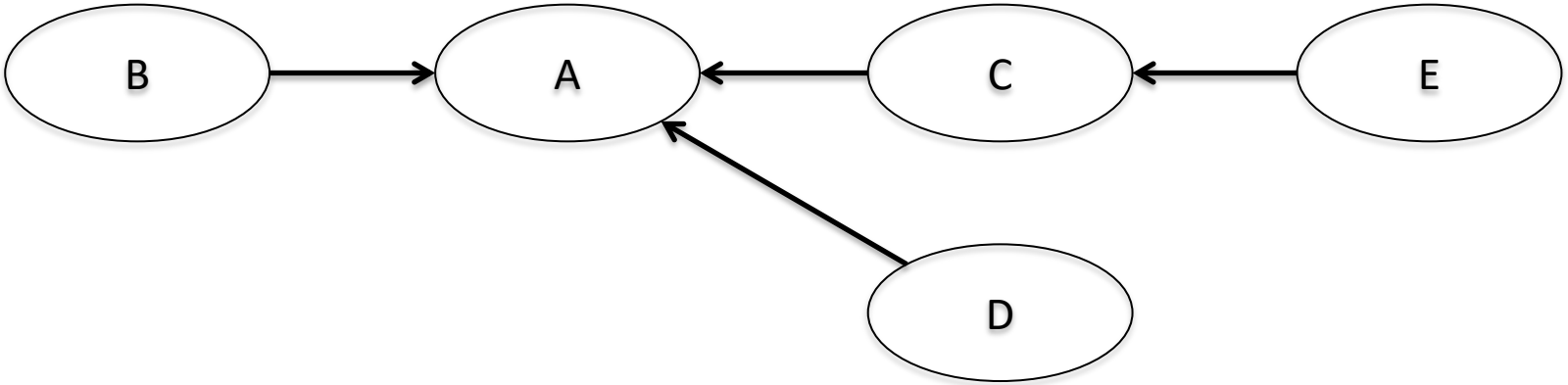


in many other modules which relied on the original module directly or indirectly

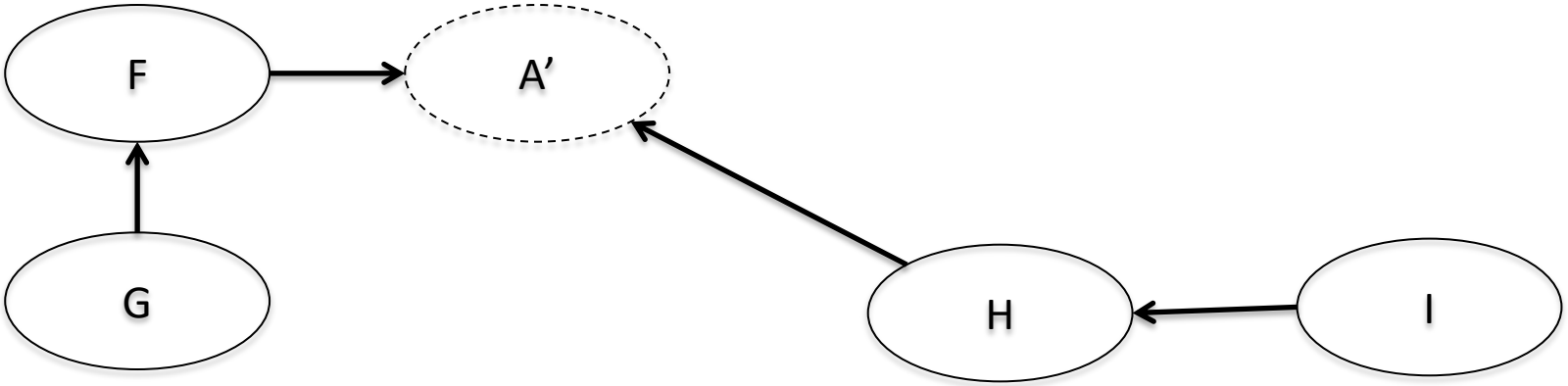
Typical situation where the needs for Open and Closed modules are **hard to reconcile**



A module and its clients

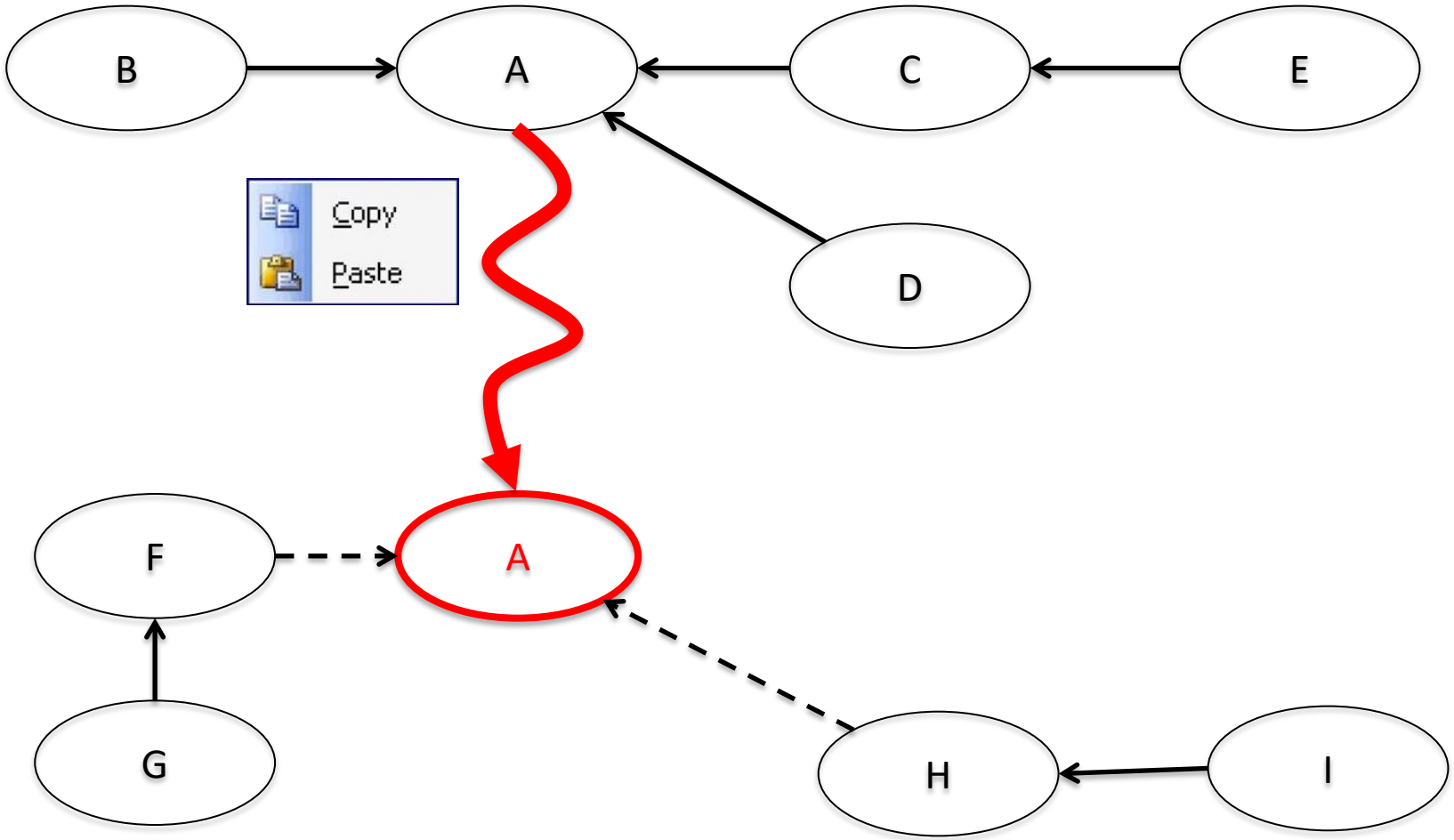


New clients which need A', an adapted or extended version of A

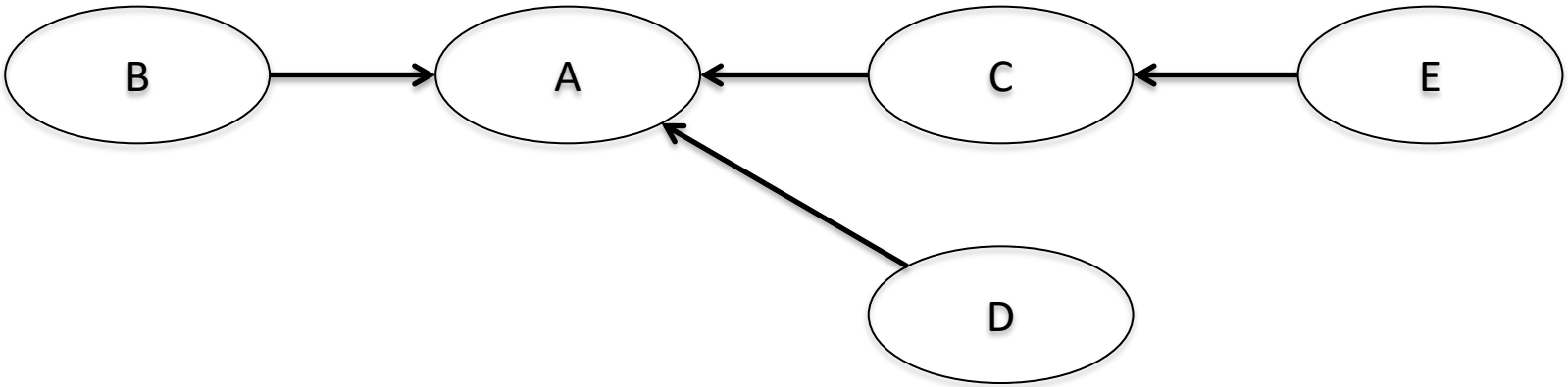


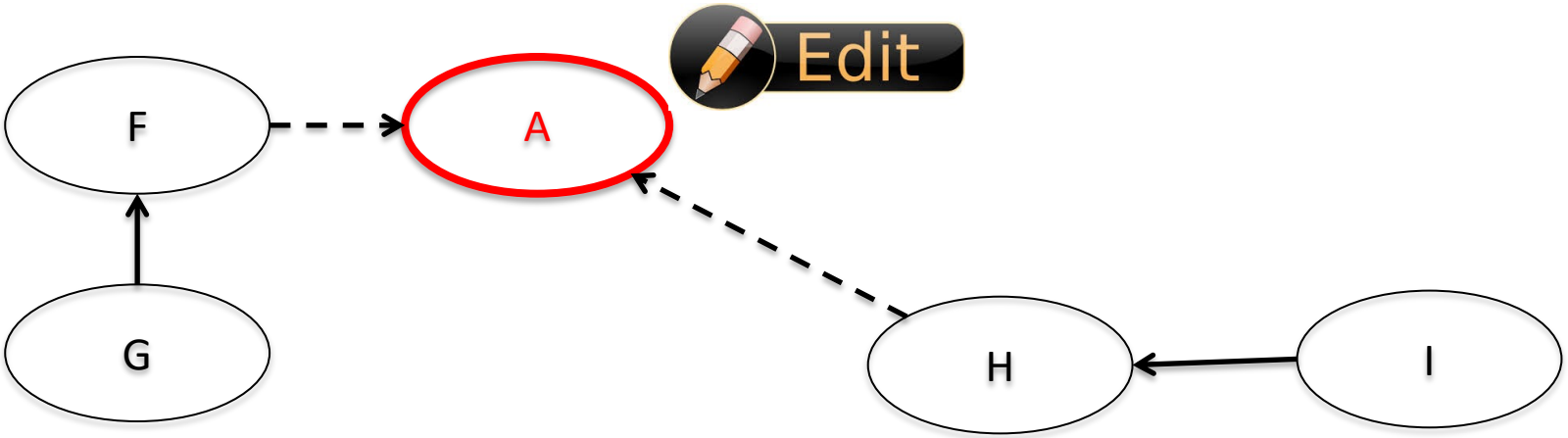
With non-OO methods,
there seem to be **only 2 solutions**,
equally unsatisfactory

Solution 1



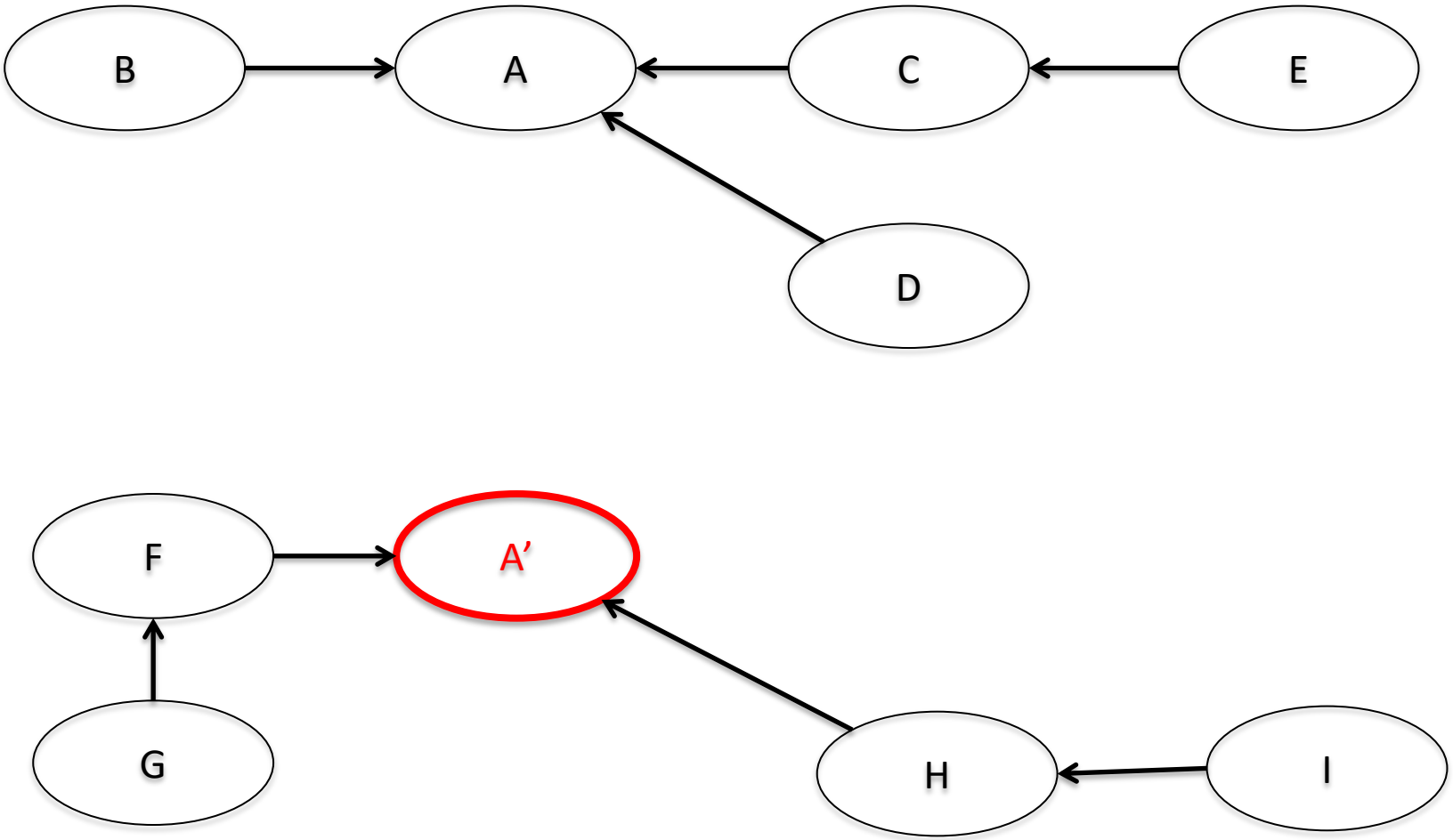
Solution 1





Solution

We have taken a copy of A and modified it, turning it into the desired A’

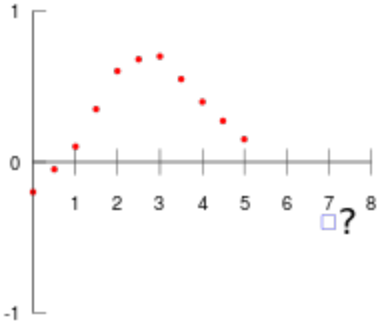


Solution

Meyer's Assessment

if you extrapolate its effects to

- **many** modules
- **many** modification requests
- a **long** period of time

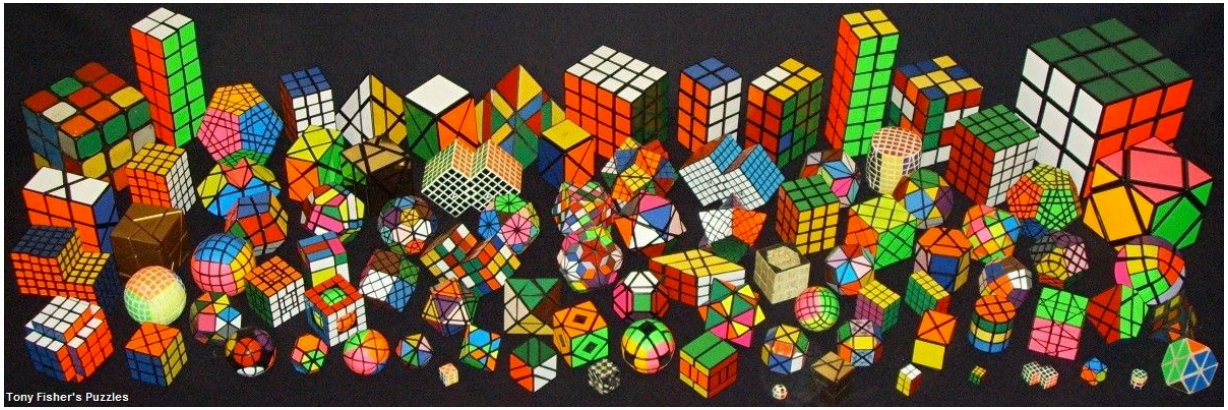


the consequences are appalling

an **explosion of variants**
of the original module,

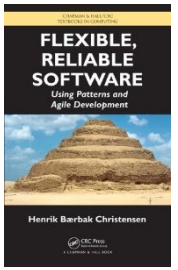
many of them very similar,

but **never quite identical**

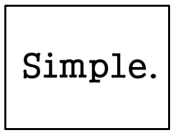


Solution (Source tree copy)

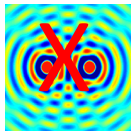
Christensen's Assessment



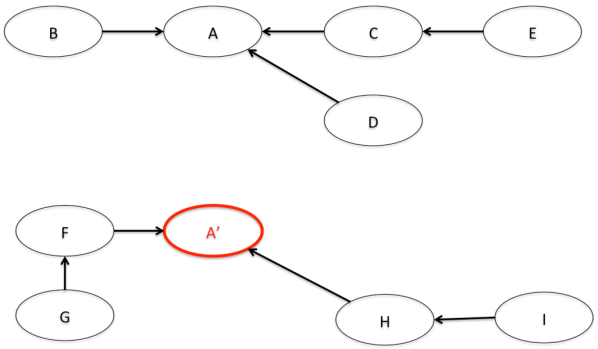
Probably the main reason it is **encountered so often in practice**



Easy to explain to colleagues & new developers



No implementation interference



Solution (Source tree copy)

Christensen's Assessment

Cons

Quick

Quick but **very dangerous**

The solution has **severe limitations**.

THE LONG RUN

In the long run it is often a **real disaster…**

because it leads to the

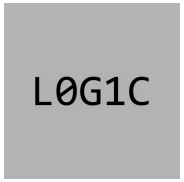
multiple maintenance problem

Solution (Source tree copy)

Christensen's Assessment



when you want to add/modify logic

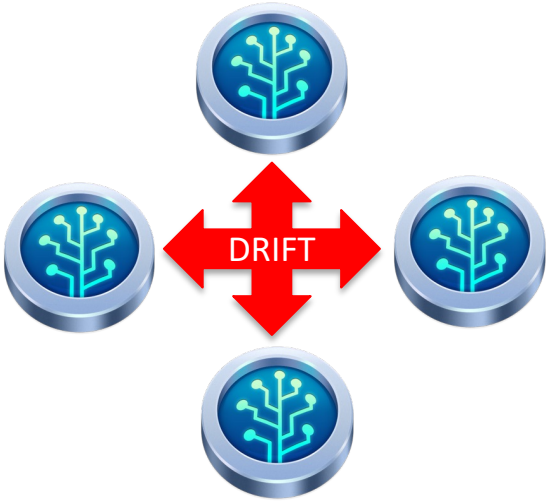



you have to:

- Do it for each source tree
- Write the same test cases for each source tree

when you need to remove a defect

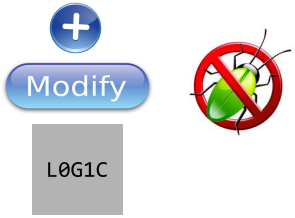
you have to do it in each source tree



Practice shows that over time the source trees evolve into completely different directions: **they drift apart**.

After a while it is more or less like **maintaining** a set of **completely different applications**





At that point, before you do any of the operations, **you have to first analyse <u>each</u> source tree!!**

Example

SAWOS - Semi Automatic Weather Observation System

Used in airports to generate reports of local weather

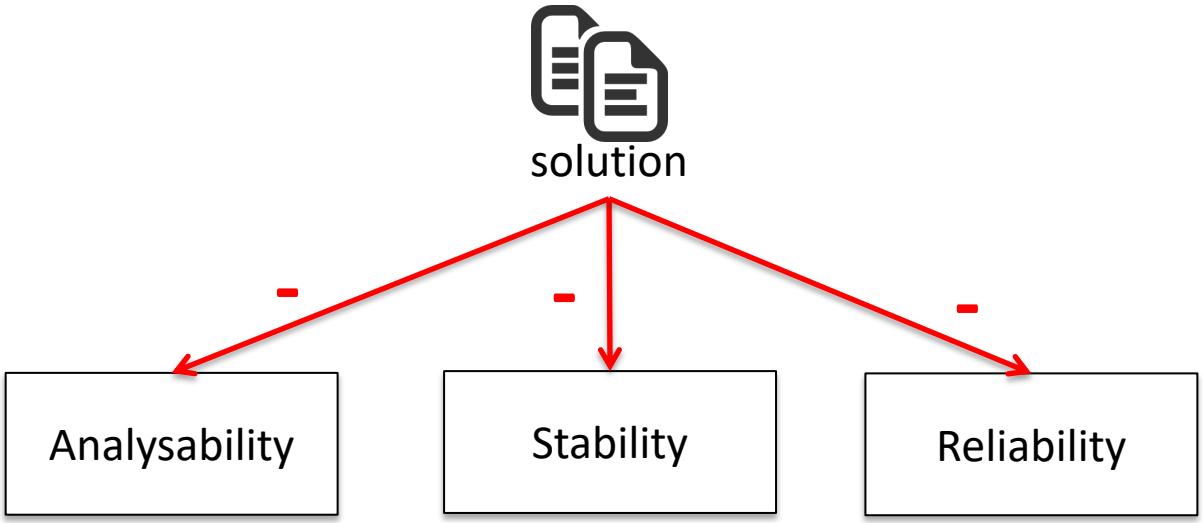
init and config code

8 copies!!!

one variant for each airport in Denmark

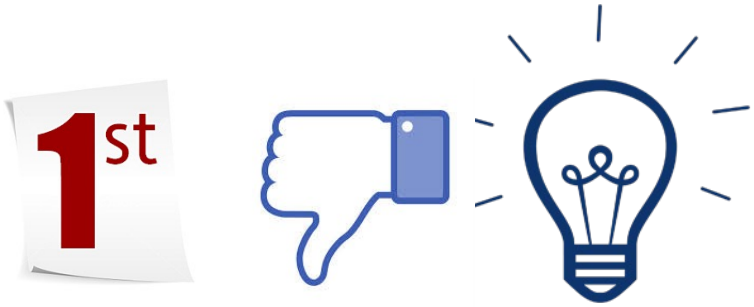
However, SAWOS evolved and many programmers participated in its development. When I left the company the setup and configuration code had grown into about 20 source code files containing 425 KB code (roughly 13,000 lines of code) and these were now maintained in eight different copies. Any defect or new requirement that was realized by code in these parts had to be analyzed and potentially coded in each of the eight copies. It required extreme care to do this properly. I

Summary

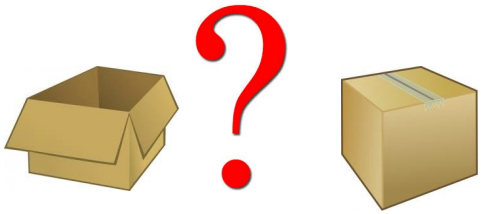


multiple maintenance problem

That was Meyer's first **unsatisfactory** solution

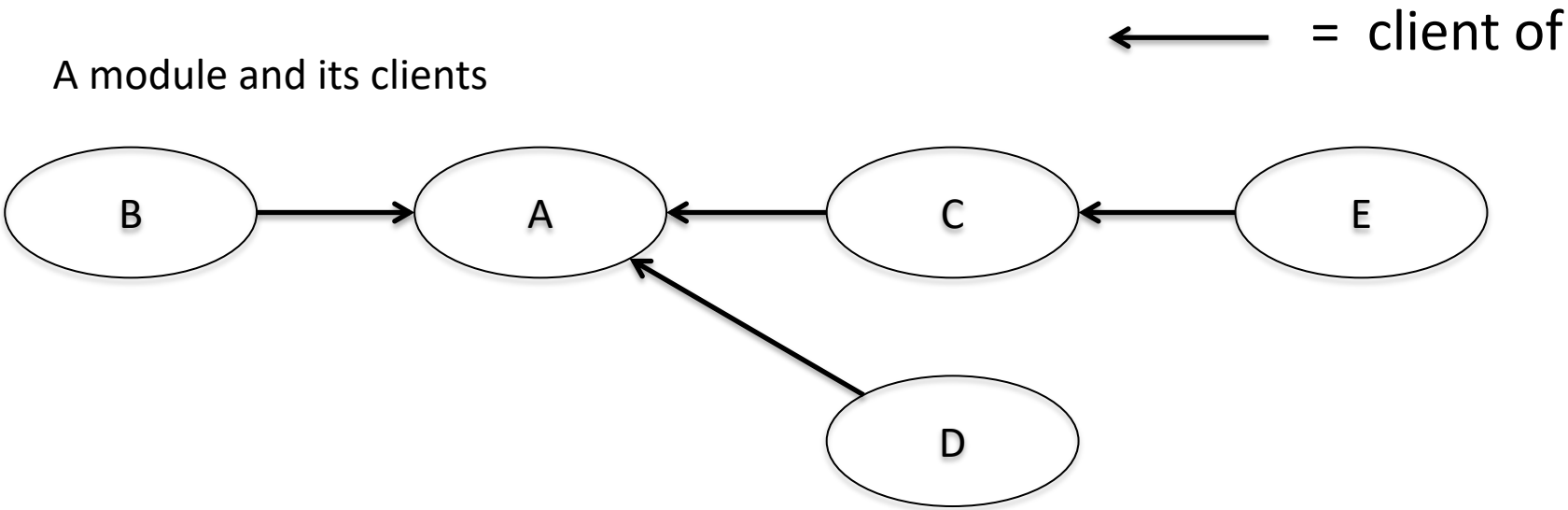


to the problem of making modules both OPEN and CLOSED

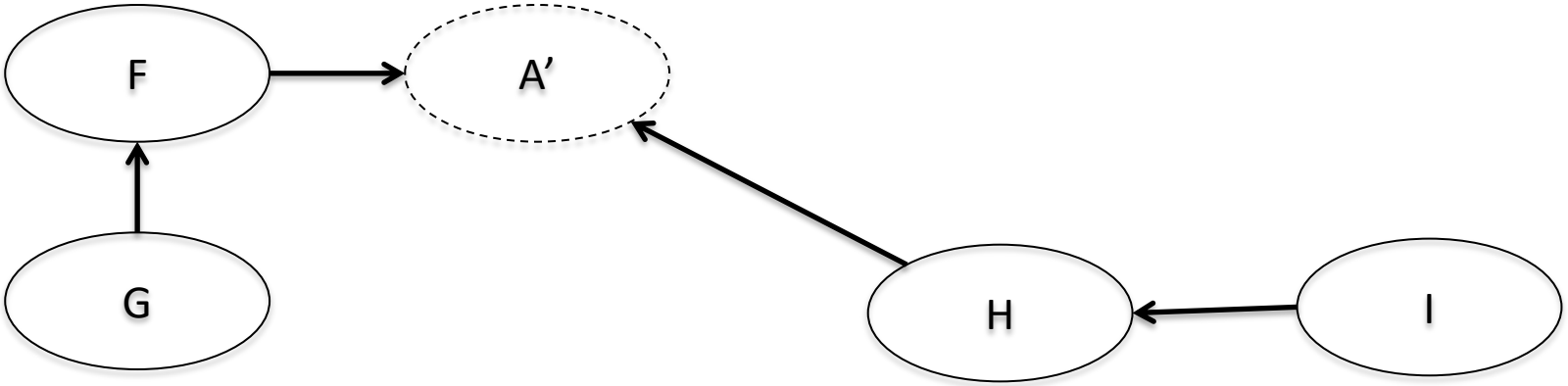


Let's turn to the second one

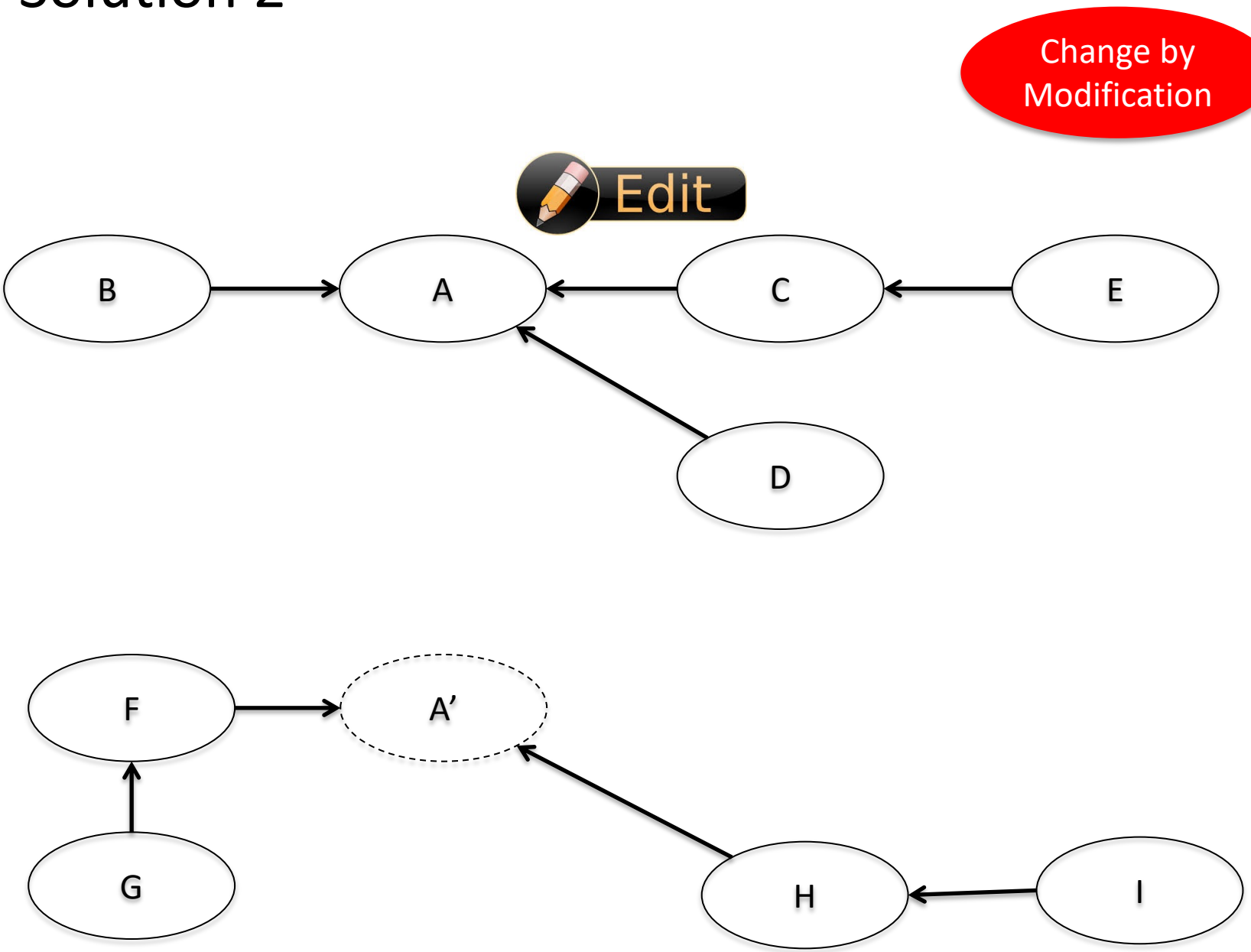
Problem



New clients which need A', an adapted or extended version of A



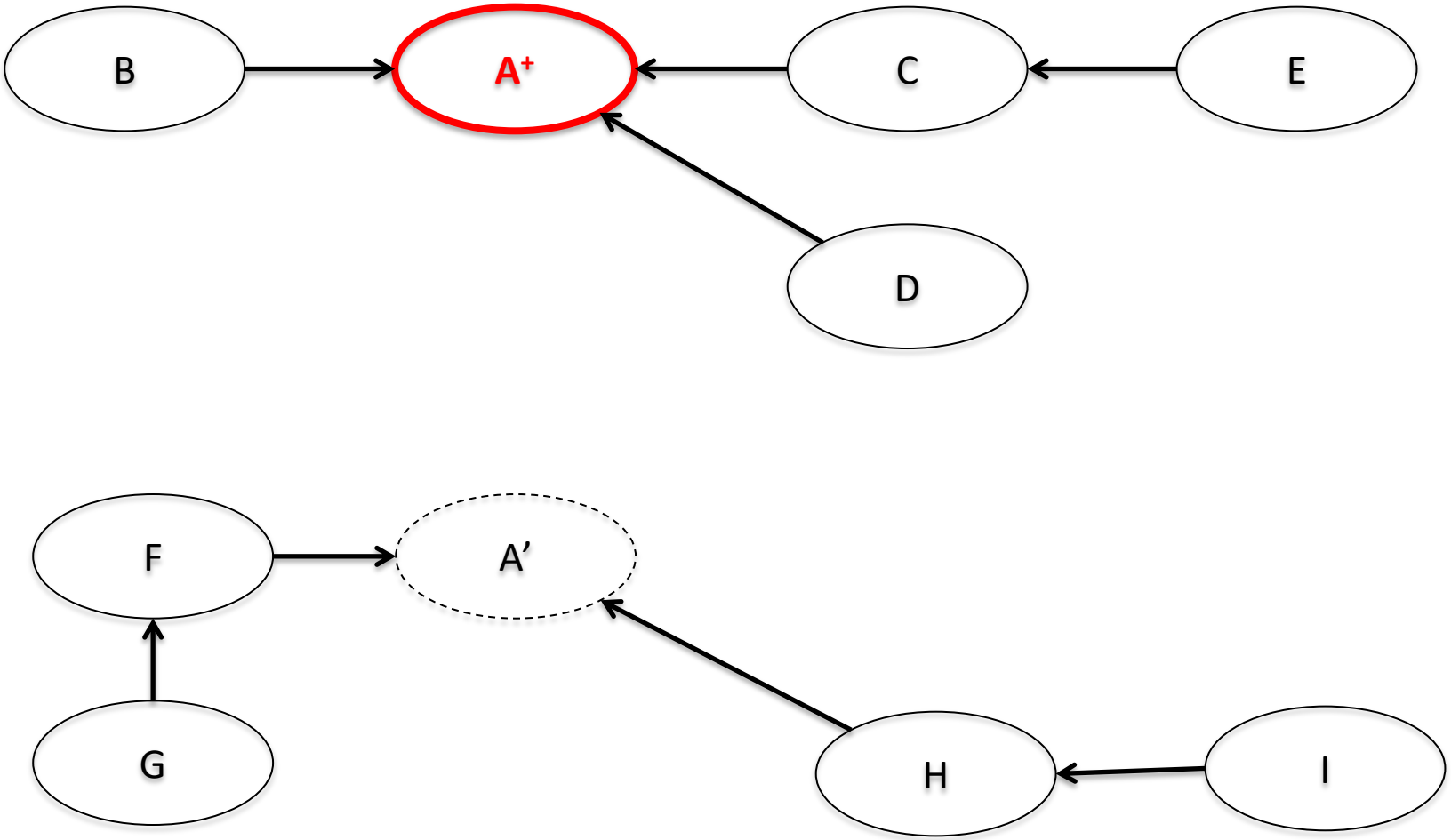
Solution 2



Solution

We have modified A into A+, which can switch between two modes of execution

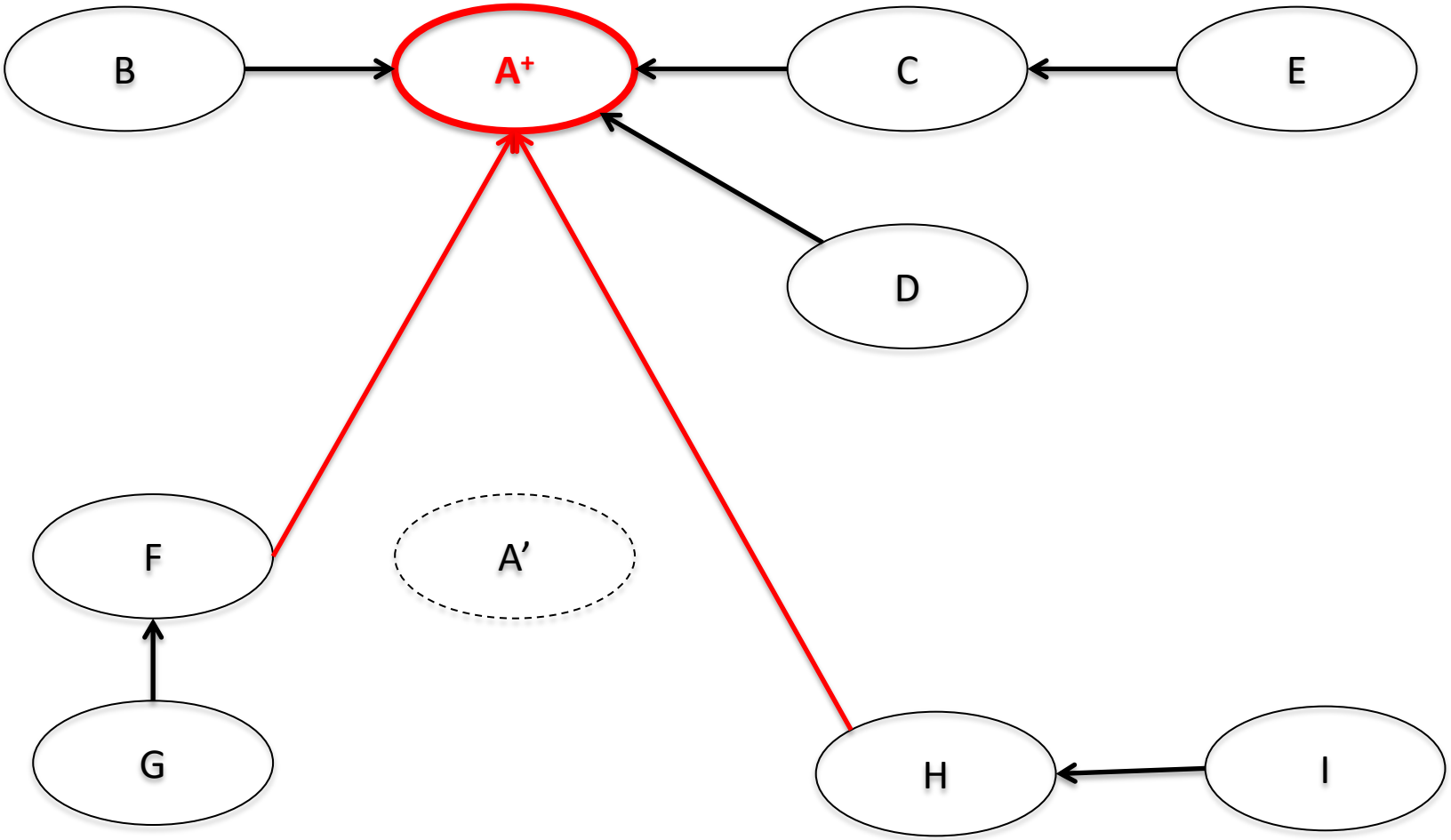
In one mode it behaves like A, and in the other it behaves as expected of A’



Solution

We have modified A into A+, which can switch between two modes of execution

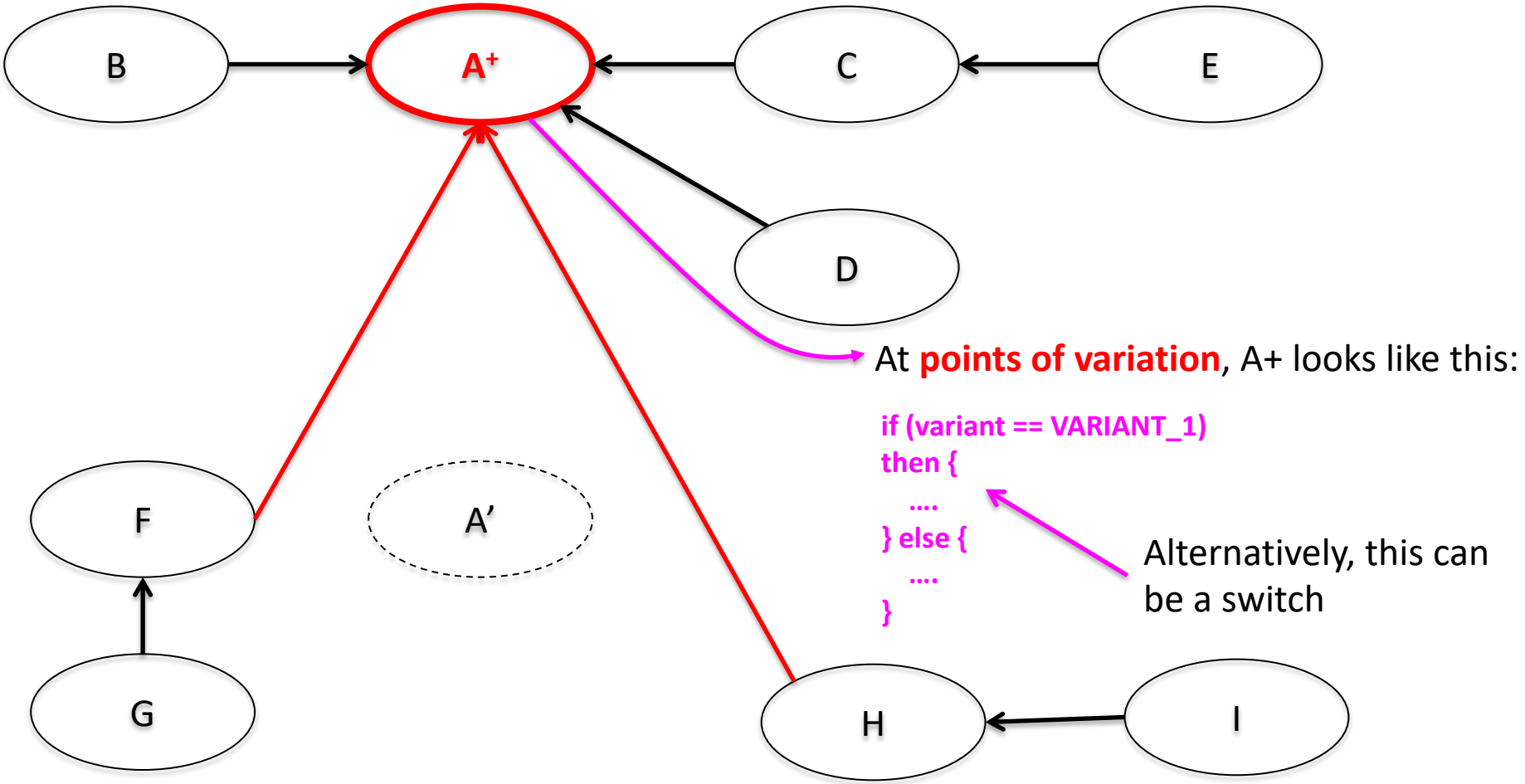
In one mode it behaves like A, and in the other it behaves as expected of A’



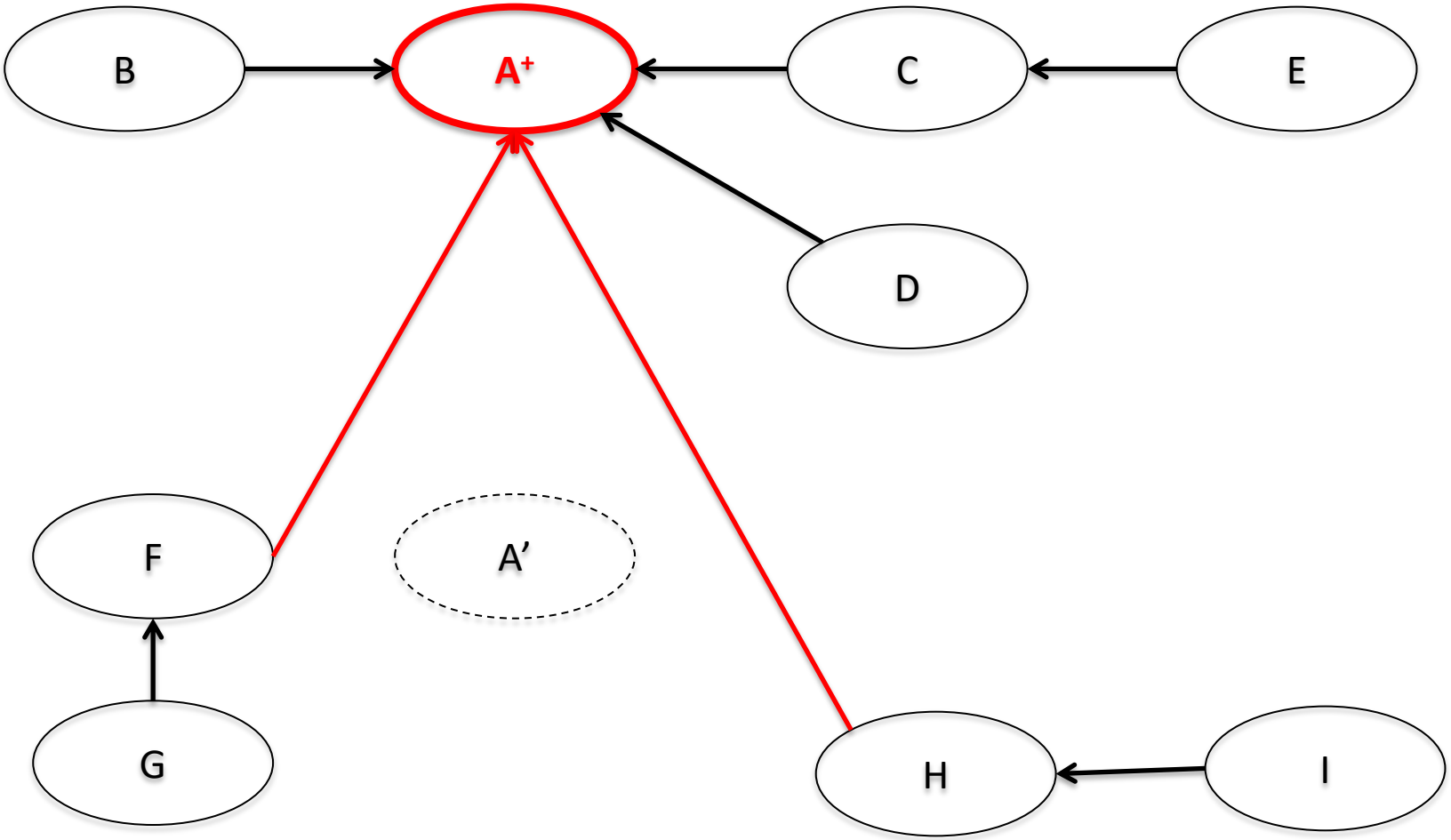
Solution

We have modified A into A+, which can switch between two modes of execution

In one mode it behaves like A, and in the other it behaves as expected of A'



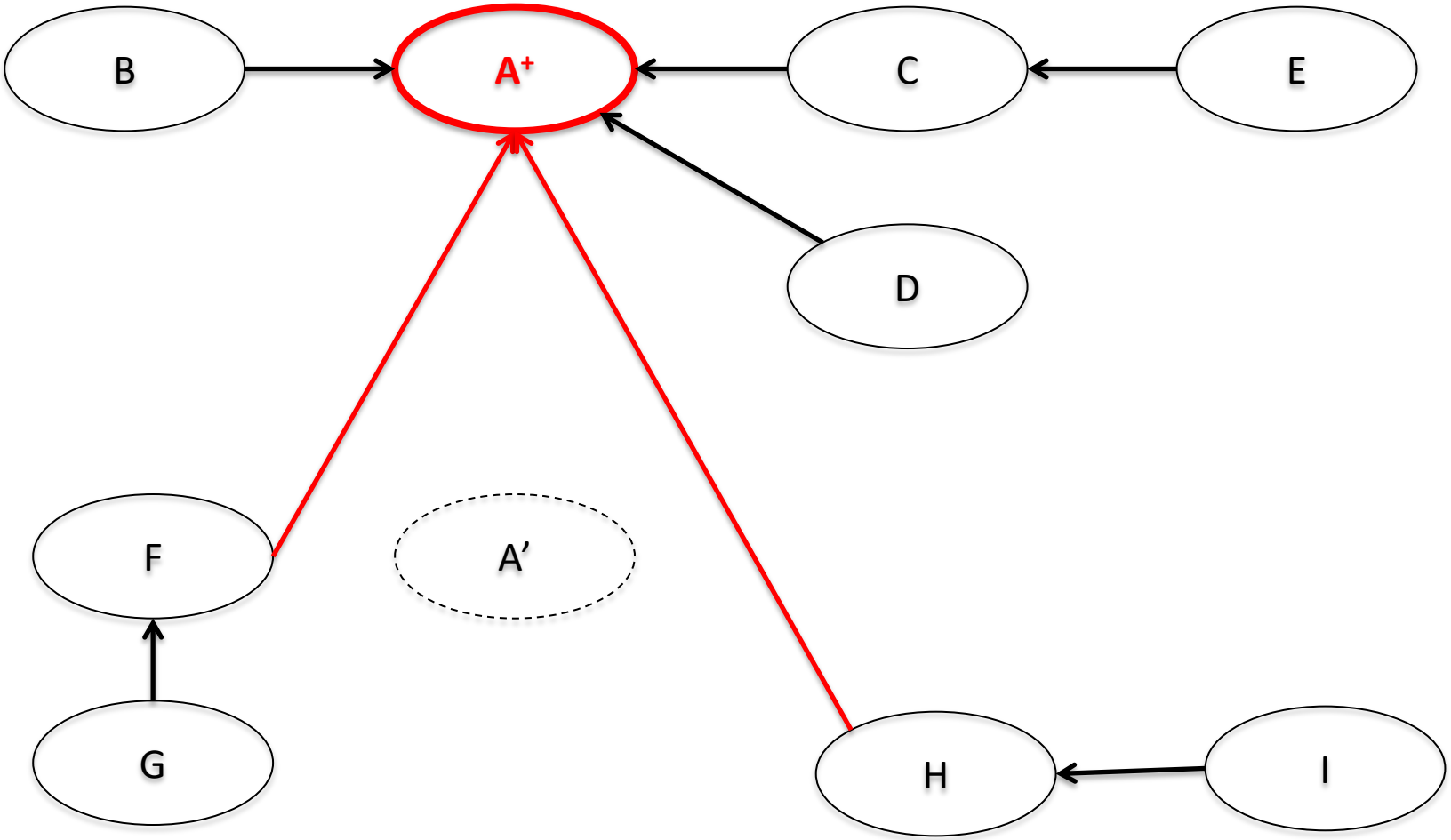
Solution – **Meyer's Assessment**



Solution – **Meyer's Assessment**

The **potential for disaster** is obvious: changes to A may invalidate the assumptions on the basis of which the old clients used A.

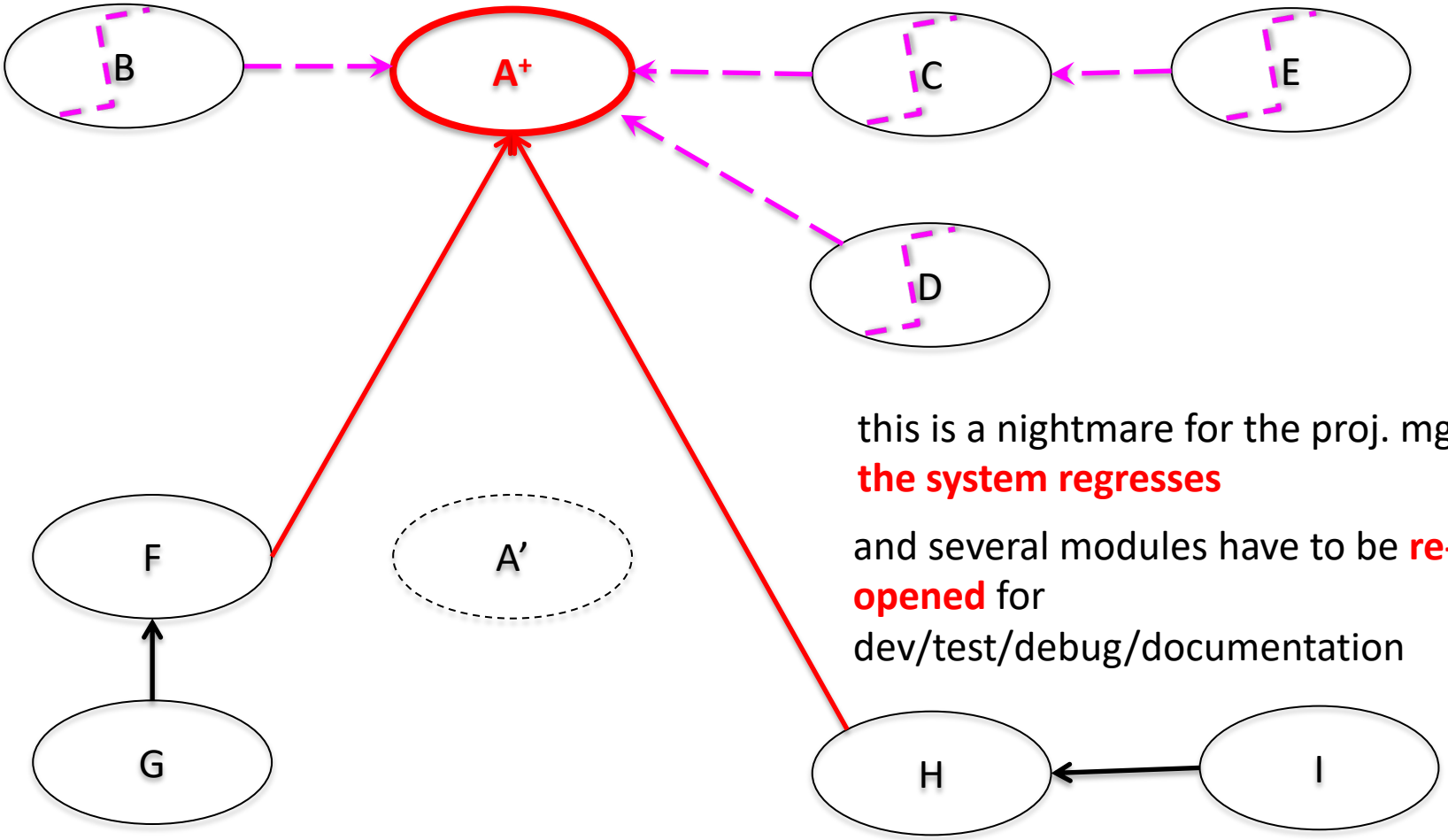
So the changes may start a dramatic **series of changes in clients, client of clients**....etc



Solution – **Meyer's Assessment**

The **potential for disaster** is obvious: changes to A may invalidate the assumptions on the basis of which the old clients used A.

So the changes may start a dramatic **series of changes in clients, client of clients**....etc



this is a nightmare for the proj. mgr.
the system regresses

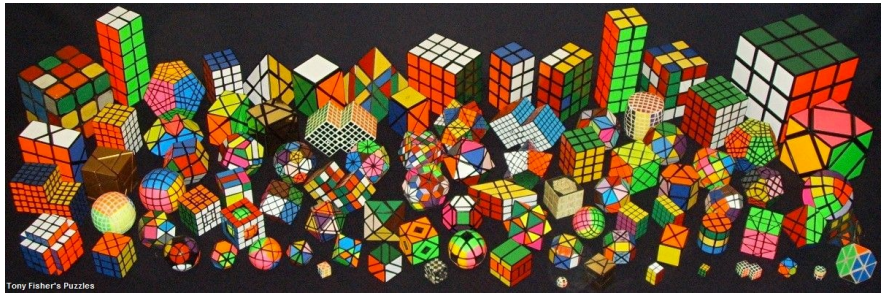
and several modules have to be **re-opened** for
dev/test/debug/documentation

Solution – **Meyer's Assessment**

Even though the Change solution has this **problematic ripple effect**, it is still **better than the Copy solution.**

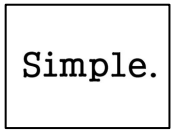
On the surface, the copy solution seems better because it avoids the **ripple effect of change** but in fact it may even be more catastrophic…it only postpones the **day of reckoning**

We saw earlier the risks of an explosion of variants, many of them very similar, but **never quite identical**:

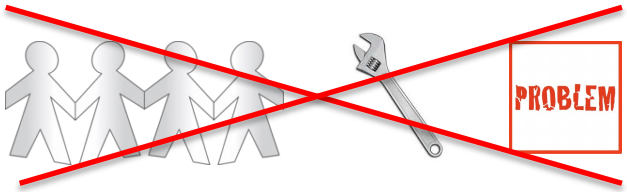


Solution (Parametric solution)

Christensen's Assessment



Conditionals are **easy to understand**. So approach is easy to describe to other developers.



Avoids **Multiple Maintenance Problem**

Only one code base to maintain

solution

solution

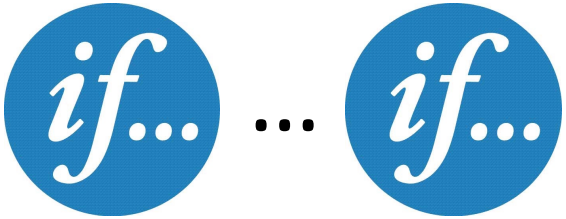
Solution (Parametric solution)

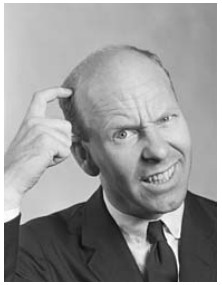
Christensen's Assessment

Cons Liabilities, most of which deal with long term maintainability

Reliability Concerns – solution relies on **Change by Modification** with risk of introducing new defects

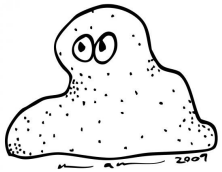
Analysability concerns – as more and more requirements are handled by parameter switching, the code becomes **less easy to analyse**





Responsibility erosion – the software has, without much notice, been given an extra responsibility

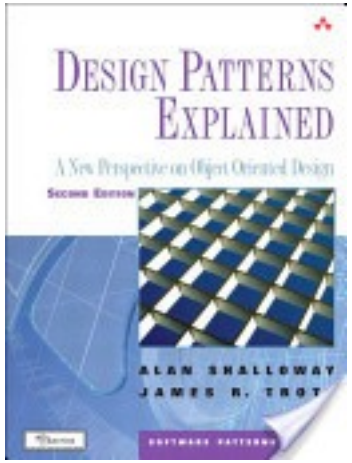
drives towards **Procedural Design**



Blob aka God Class

Solution (Switches)

Shalloway's Assessment



2004

A **reasonable approach at first**, but one with **serious problems** for applications that need to grow over time

Not too bad as long as you just keep adding cases…

```
// Handle Tax switch (myNation) {
   case US:
    // US Tax rules here break;
   case Canada:
    // Canadian Tax rules here break;
}
```

```
// Handle Currency switch (myNation) {
  case US:
    // US Currency rules here break;
  case Canada:
    // Canadian Currency rules here break;
}
```

```
// Handle Date Format switch (myNation){
   case US:
    // use mm/dd/yy format break;
   case Canada:
    // use dd/mm/yy format break;
}
```

but soon you need to introduce fall-throughs…

```
// Handle Tax switch (myNation) {
   case US:
      // US Tax rules here break;
   case Canada:
      // Canadian Tax rules here break;
   case Germany:
      // Germany Tax rules here break;
}
```

```
// Handle Currency switch (myNation) {
   case US:
      // US Currency rules here break;
   case Canada:
      // Canadian Currency rules here break;
   case Germany:
      // Euro Currency rules here break;
}
```

```
// Handle Date Format switch (myNation) {
   case US:
      // use mm/dd/yy format break;
   case Canada:
   case Germany:
      // use dd/mm/yy format break;
}
```

```
// Handle Language switch (myNation) {
   case US:
   case Canada:
      // use English break;
   case Germany:
      // use German break;
}
```

…and then the switches are **not as nice as they used to be**

Eventually you need to start adding **variations within a case.**

Suddenly **things get bad in a hurry**.

```
// Handle Language switch (myNation) {
    case Canada:
        if ( inQuebec) {
            // use French break;
        }
    case US:
        // use English break;
    case Germany:
        // use German break;
}
```

The flow of the switches themselves becomes **confusing, hard to read, hard to decipher**.

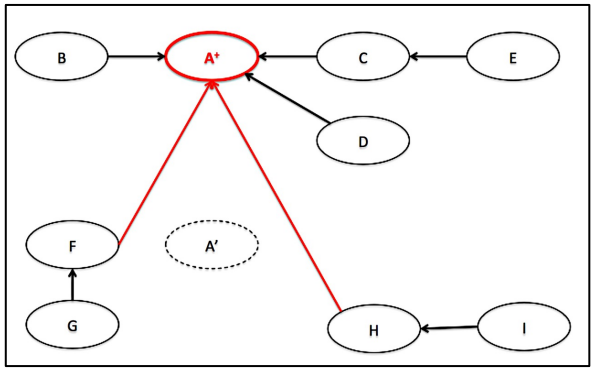
When a new case comes in the programmer **must find every place it can be involved** (**often finding all but one of them**).

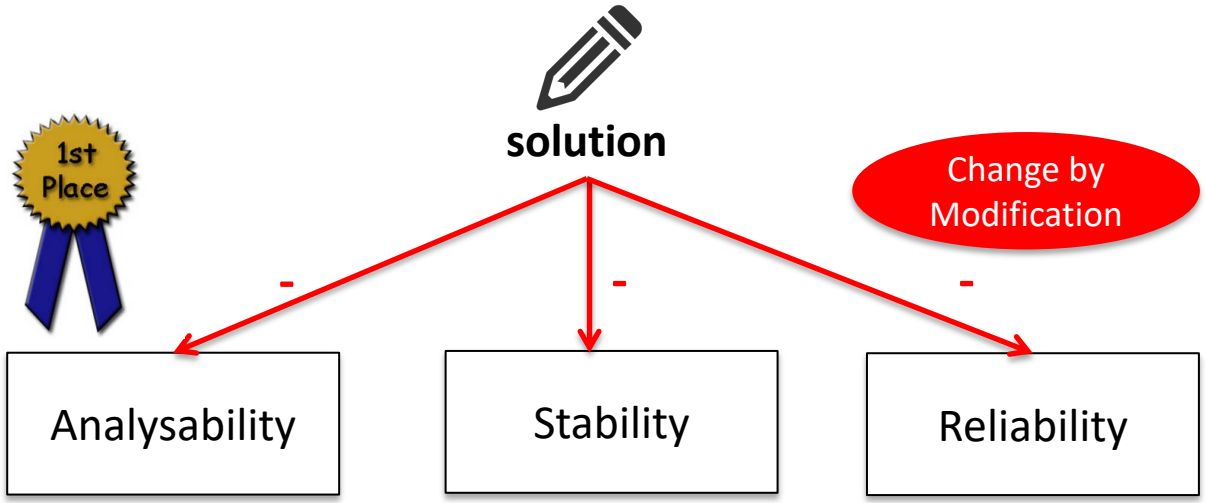
I like to call this **`switch`**

Summary

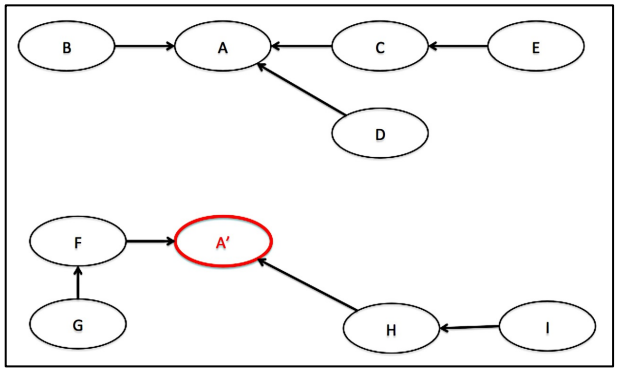
With non-OO methods, there are **only only 2 solutions** available to us,
BOTH UNSATISFACTORY

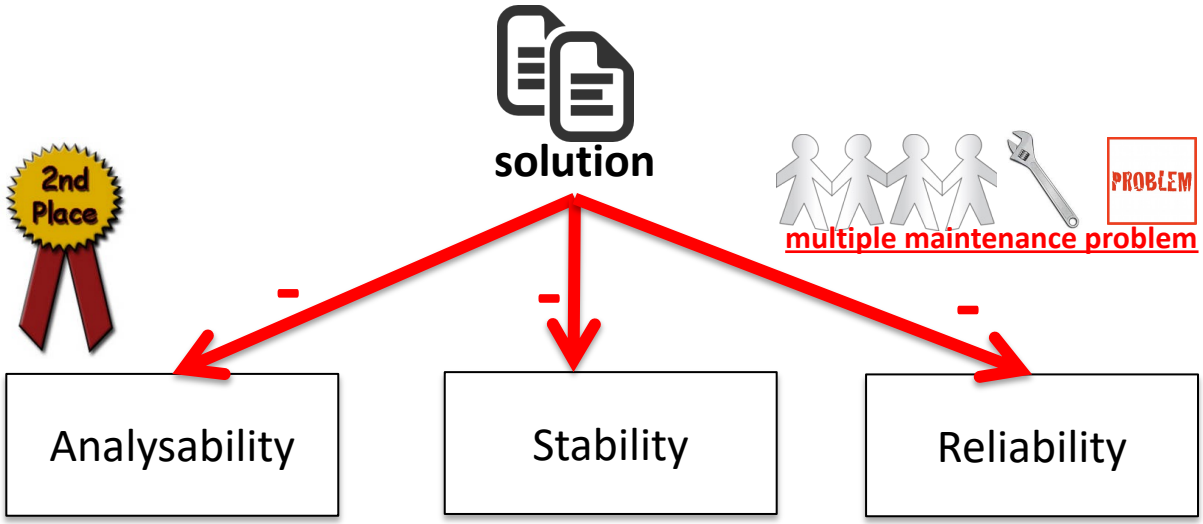
CHANGE





COPY





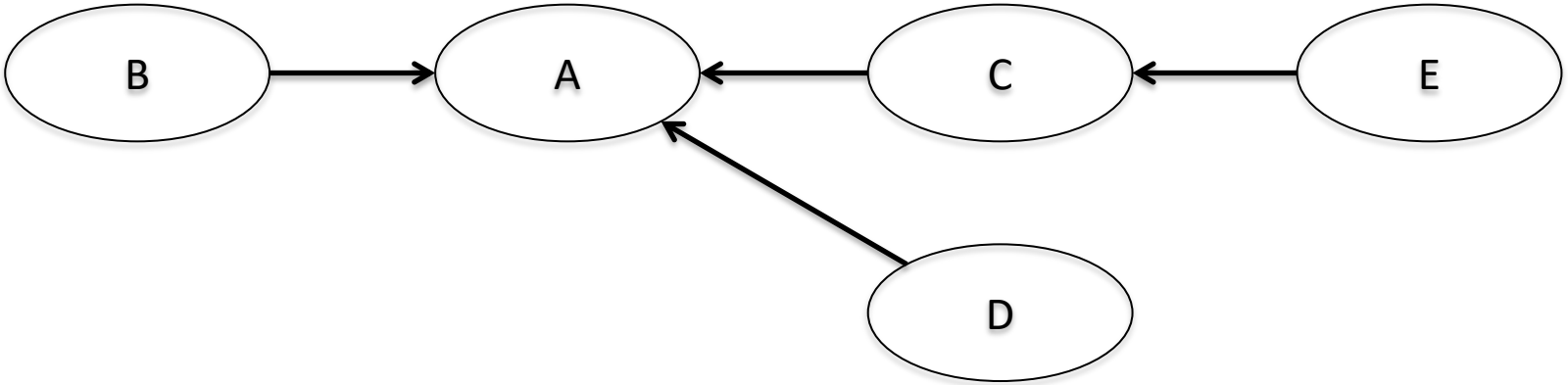
If non-OO methods are all we have, then Meyer says we face a **change or copy dilemma**

So how can we have modules that are both 

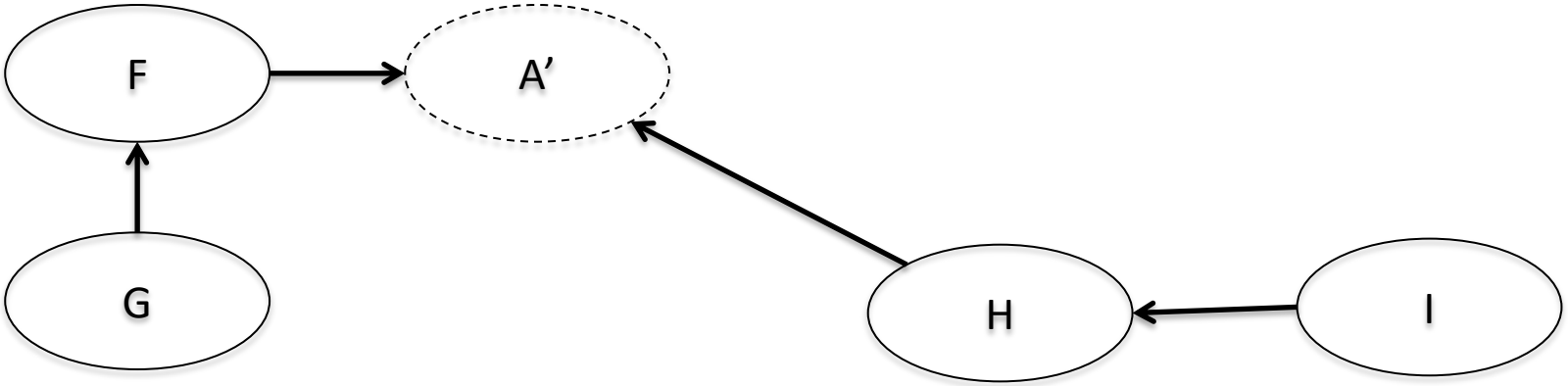
and 

?

How can we keep A and everything in the top part of the figure unchanged, …



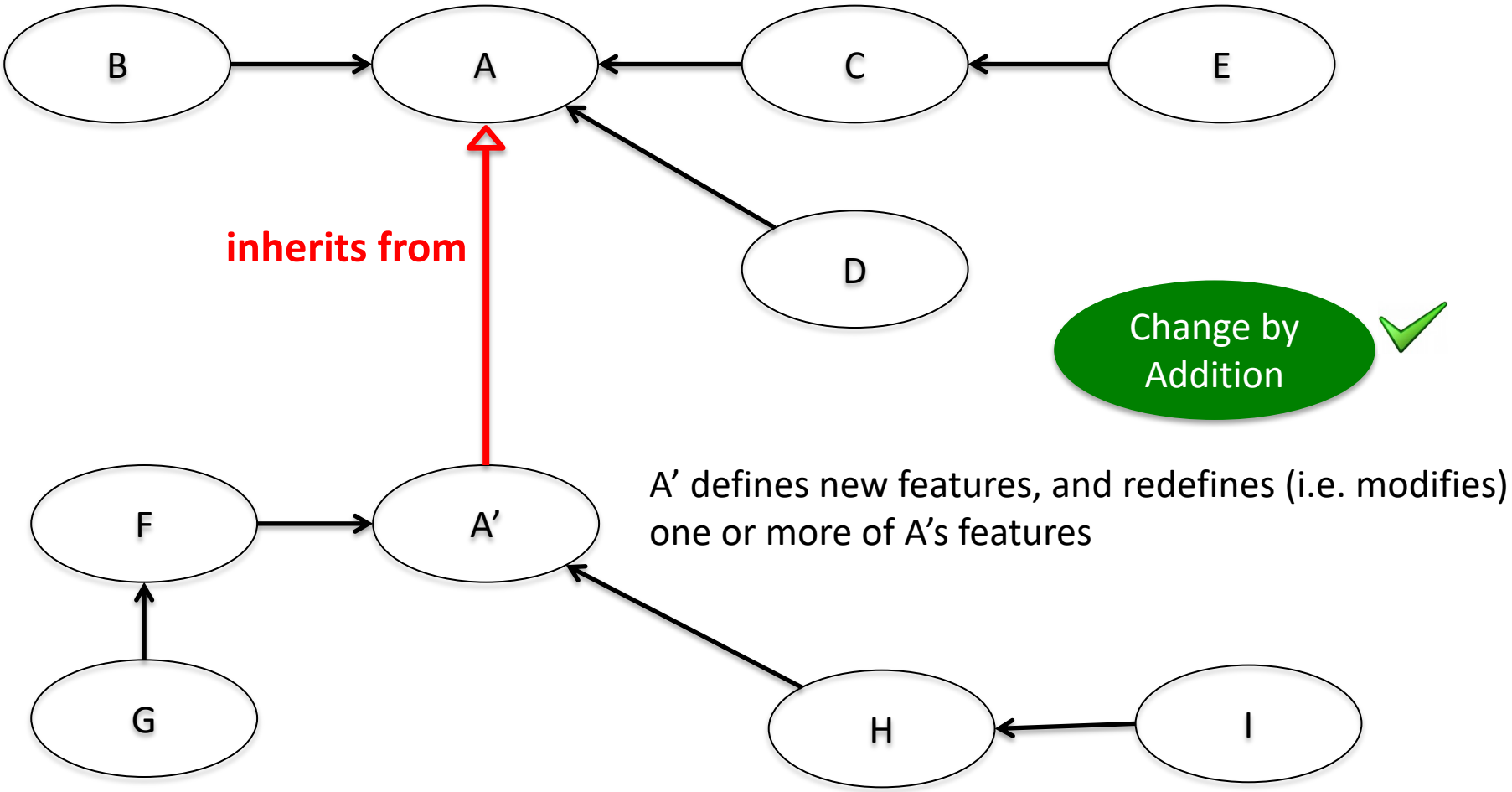
…while providing A' to the bottom clients, and **avoiding duplication of software**?

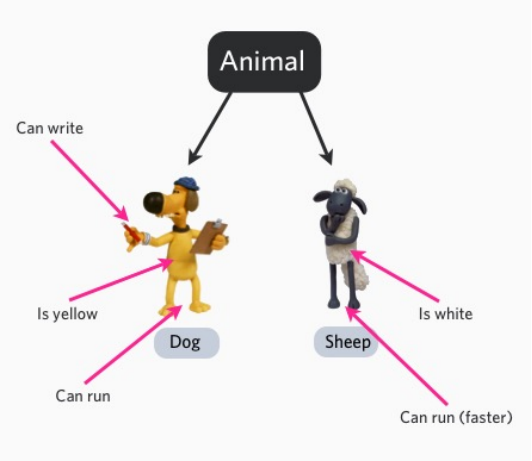


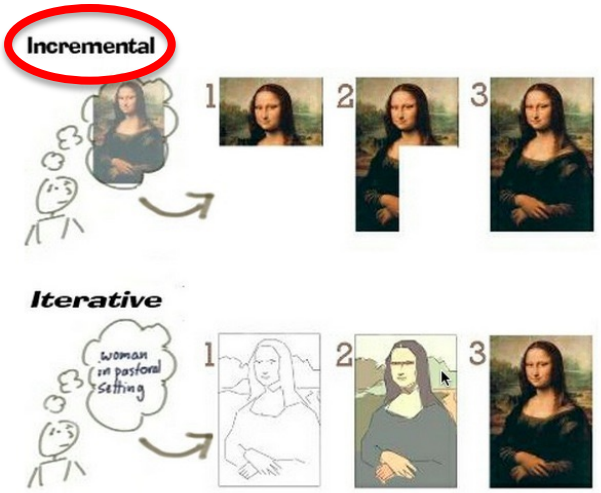
With the OO concept of inheritance

Inheritance allows us to **get out of the CHANGE OR COPY dilemma**…

…because inheritance allows us to **define a new module A' in terms of an existing module A**, …**by stating only the differences between the two**

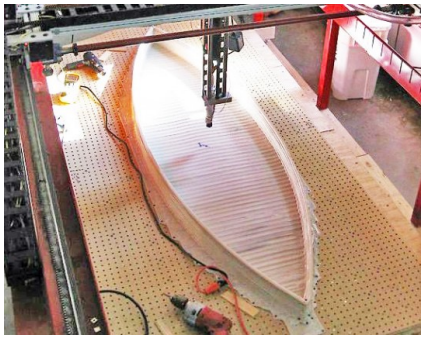






Thanks to **inheritance**, OO developers can adopt a much more incremental approach to software development than used to be possible with earlier methods

OO inheritance



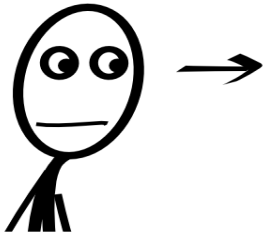
Hacking = **Slipshod** approach to building and modifying code

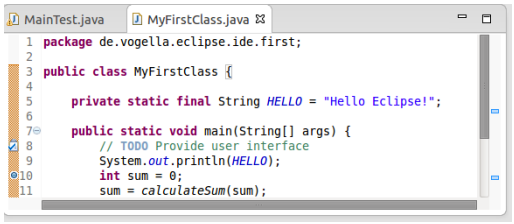
Slipshod = Done poorly or too quickly; careless.

The Hacker may seem bad

but often his heart is pure.

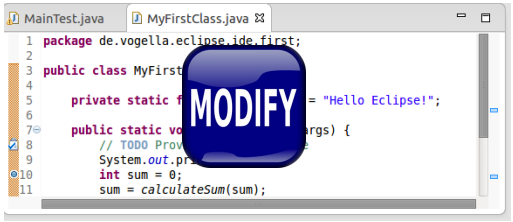
Hacker





Useful

He sees a useful piece of software, which is almost able to address the needs of the moment, more general than the software's original purpose.



solution

Spurred by a laudable desire not to redo what can be reused, our hacker starts modifying the original to add provisions for new cases

The impulse is good but the effect is often to pollute the software with many clauses of the form **if** that_special_case **then**…

switch

if (<special case A>) **then** …

if (<special case B>) **then** …

if (<special case C>) **then** …

if (<special case D>) **then** …

```
public class TcpClientSample
    c void Main()
                                    ng input, stringData;
    TcpClient se
    try{
        server = new
                                    . . ", port);
                                    ketException){
                                    eLine("Unable to connect to server");
    }
    NetworkStream ns = server.GetStream();
    int recv = ns.Read(data, 0, data.Length);
    stringData  = Encoding.
      ASCII.GetString(data, 0, recv);
    Console.WriteLine(stringData);
    while(true){
        input = Console.ReadLine();
        if (input == "exit") break;
                                    newchild.Properties["ou"].Add
                                    ("Auditing Department");
                                    newchild.CommitChanges();
                                    newchild.Close();
```

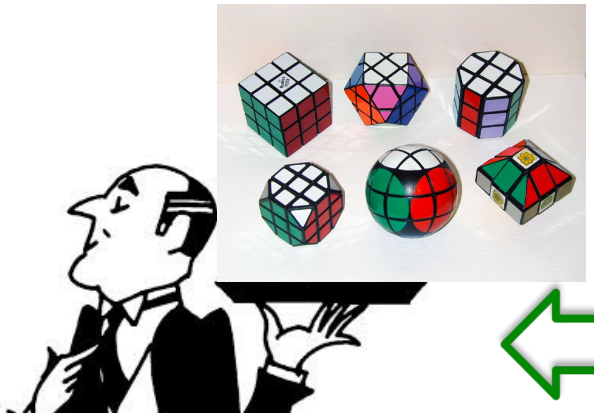
so that after a few rounds of hacking, perhaps by different hackers,

the software starts resembling a chunk of Swiss cheese that has been left outside for too long in August **– it has both holes and growth**

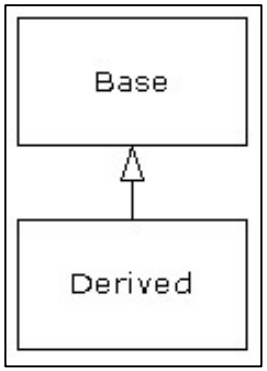
One way to describe the **OCP** and the consequent OO techniques is to think of them as organised hacking

~~Hacking~~

Open-Closed Principle = Organised Hacking



Change by Addition



Inheritance

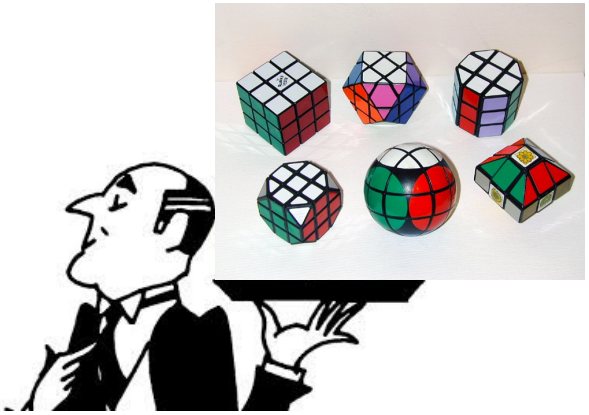
~~Change by Modification~~

consistency

The **organised form of hacking** will **enable us to cater to the variants without affecting the consistency of the original version**.

Caveats

if you have control over
original s/w and
can rewrite it



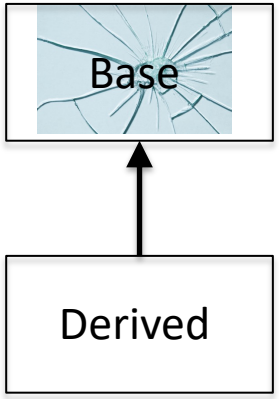
so that it will address the needs
of several kinds of clients

at no extra complication

…you should do so

The OCP principle and associated techniques are intended for the **adaptation of healthy modules**

If there is something wrong with a module you should fix it…



…not leave the original alone and try to correct the problem in the derived module

neither OCP nor redefinition in inheritance is a way to address design flaws, let alone bugs

So Let's Recap...

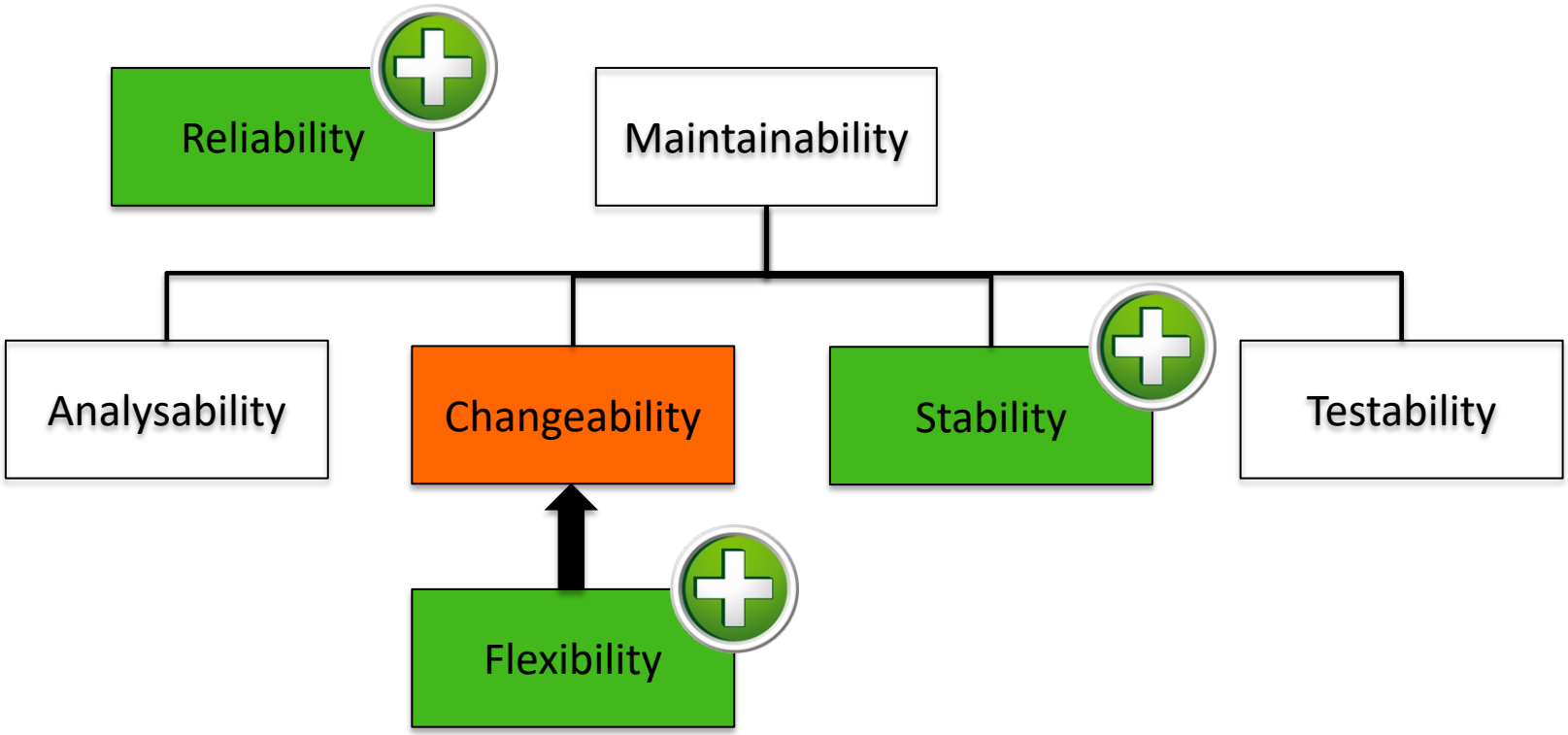
design

its purpose, its entire reason for being, is to **reduce the cost of change**.

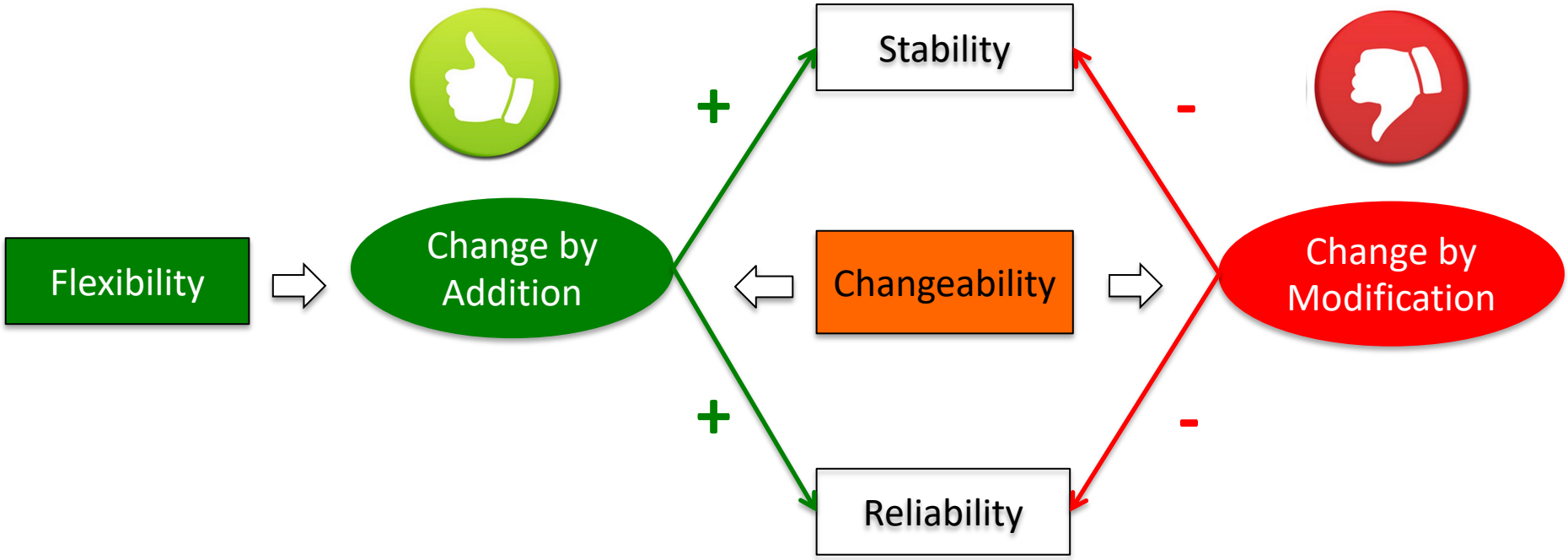
raison d'être

(n.) a reason for existing

Question: how do we promote flexibility, reliability and stability in our software?



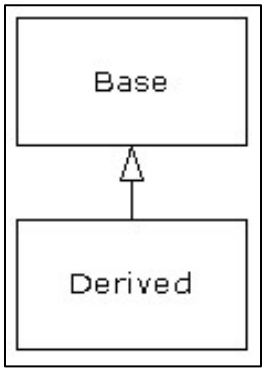
Answer: we favour '**Change by Addition**' over '**Change by Modification**'



How do we achieve Change by Addition ?

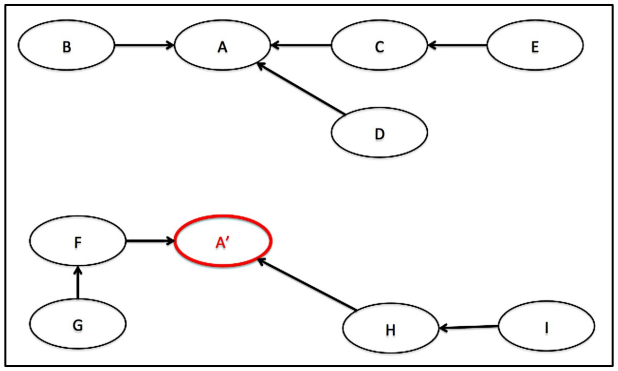
We apply the **Open-Closed Principle**

which uses **OO inheritance**



Inheritance

COPY solution

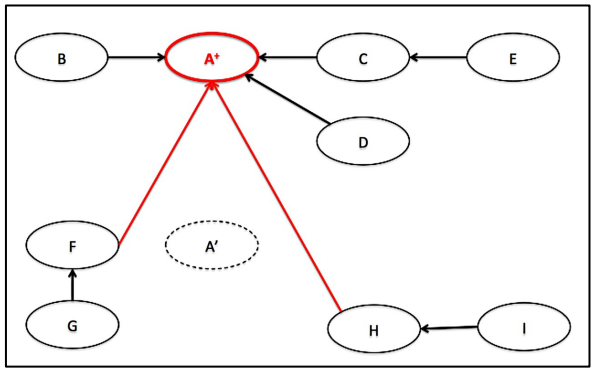


multiple maintenance problem

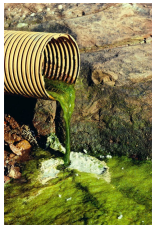
Hacker

Chooses

CHANGE solution



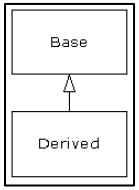
Change by Modification



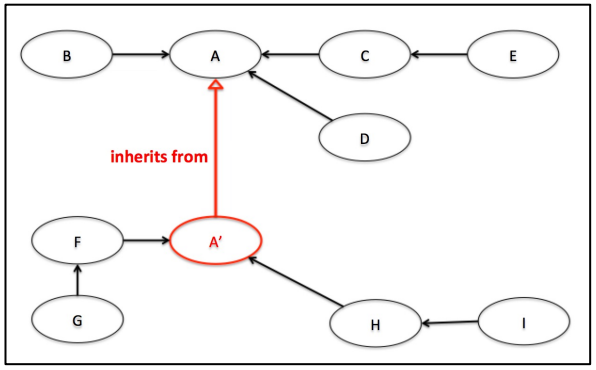
switch

Organised Hacking

Chooses



OCP solution



Change by Addition

I hope you enjoyed that.

See you in part two.