Technische Universität Braunschweig

Suresoft
SUSTAINABLE RESEARCH SOFTWARE

# Scientific software development is not a Jenga game!

## Software Engineering to the rescue

HeFDI Code School – Sustainable Research Software

# Who are we?

Dr. Jan Linxweiler
j.linxweiler@tu-braunschweig.de

Sören Peters
soe.peters@tu-braunschweig.de

# Agenda

- Part 1

  **Requirements and characteristics of maintainable and sustainable research software**

- Part 2

  **Software Development Principles and Practices (SOLID)**

- Part 3

  **Design Patterns**

- Part 4

  **Clean Code and Refactoring**

- (Part 5)

  **(Introduction to Test-Driven Development (TDD))**

# How to follow along?


Technische Universität Braunschweig



Suresoft
SUSTAINABLE RESEARCH SOFTWARE

# What is your educational background?

a) (Human) Medicine
b) Physics
c) Electrical Engineering
d) Computer Science
e) Biology
f) other

Join at
slido.com
#123 444

# Which programming language do you prefer?

a) C++
b) Java
c) Python
d) Matlab
e) R
f) other

Join at
**slido.com**
**#123 444**

## How much experience do you have in OOP - Object-Oriented Programming?

a) I have zero experience
b) I have just a little experience
c) I feel comfortable in programming
d) I'm an expert
e) I would rather not say
f) other

Join at
slido.com
#123 444

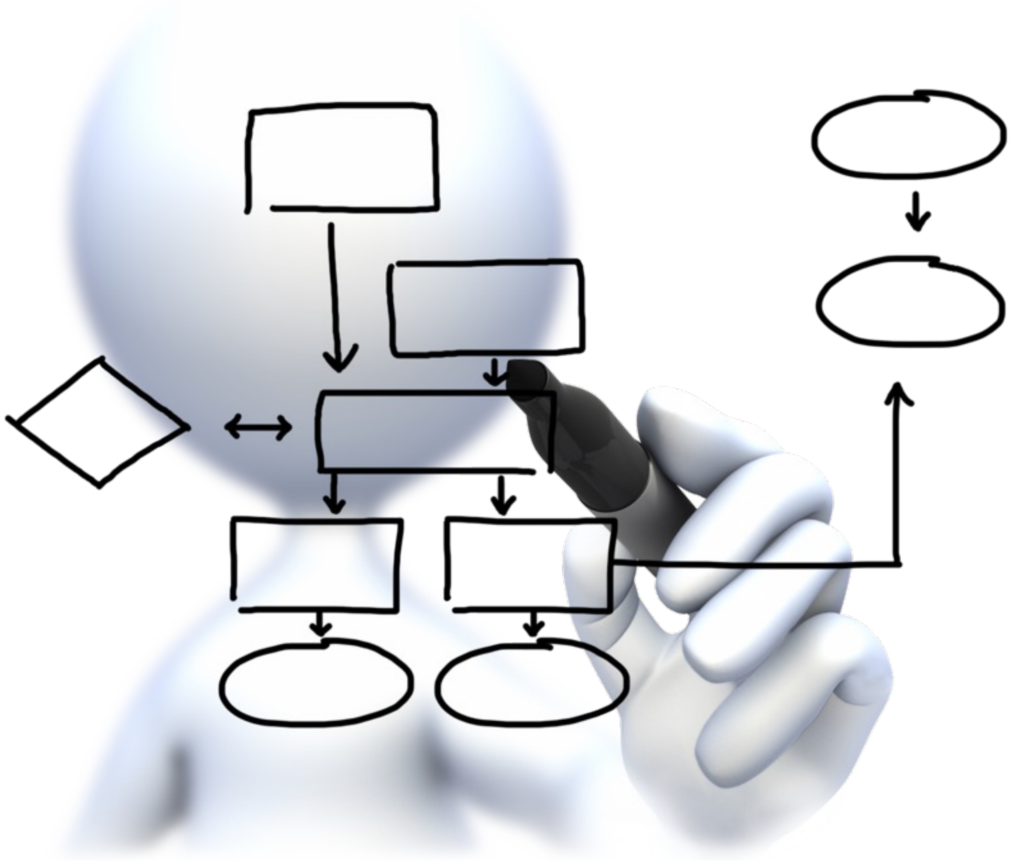
Technische Universität Braunschweig



Suresoft
SUSTAINABLE RESEARCH SOFTWARE

# How familiar are you with UML class diagrams?

a) What the heck are you talking about?
b) It would be nice to refresh my knowledge a bit
c) I feel comfortable in reading UML
d) I'm an expert
e) I would rather not say
f) other



Join at
slido.com
#123 444

# Who uses the code that you write?

a) Only you
b) Your direct colleagues
c) Other researching groups you collaborate with
d) The whole scientific community in your field
e) Even more
f) other

Join at
slido.com
#123 444

# Software provides the ultimate flexibility …

# Relationships - Dependencies - Coupling

# One of the first scientific software developers

https://bit.ly/3kYfXWR

# One of the first scientific software developers

“One of our difficulties will be the maintenance of an **appropriate discipline**, so that we **do not lose track of what we are doing**.”

- 1947

[Lecture to London Mathematical Society, February 20, 1947]

# One of the first scientific software developers

“One of our difficulties will be the maintenance of an **appropriate discipline**, so that we **do not lose track of what we are doing**.”

- 1947

[Lecture to London Mathematical Society, February 20, 1947]

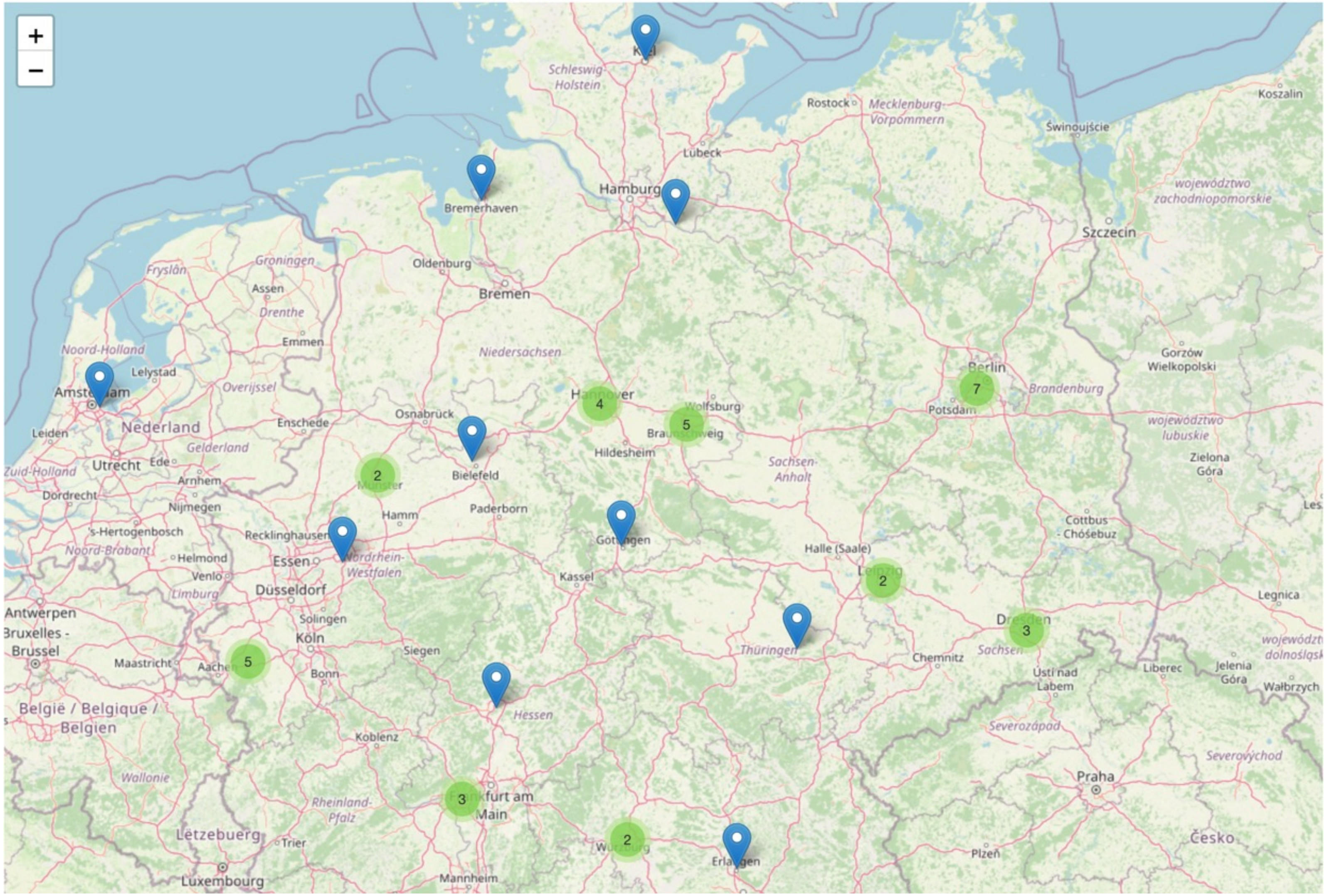
# German Society for Research Software

GESELLSCHAFT FÜR FORSCHUNGSSOFTWARE

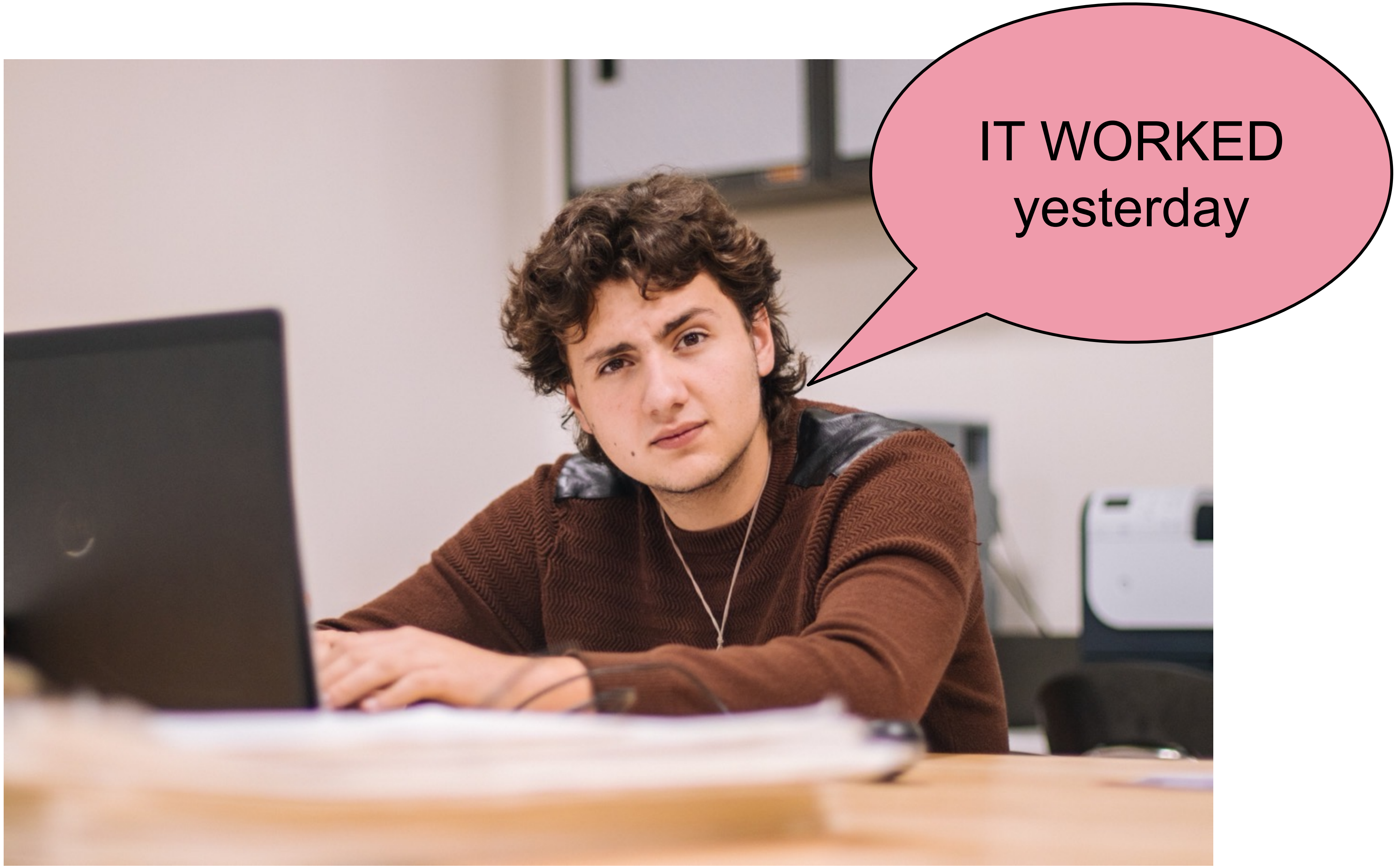
https://de-rse.org/

- recognition for software development as a scientific achievement
- anchoring in the scientific reputation system
- availability and usability of scientific software
- competence in software engineering
- quality standards for the development and review of scientific software
- reproducibility, e.g. of simulation results
- regional chapters https://de-rse.org/chapter/

# New career path in science

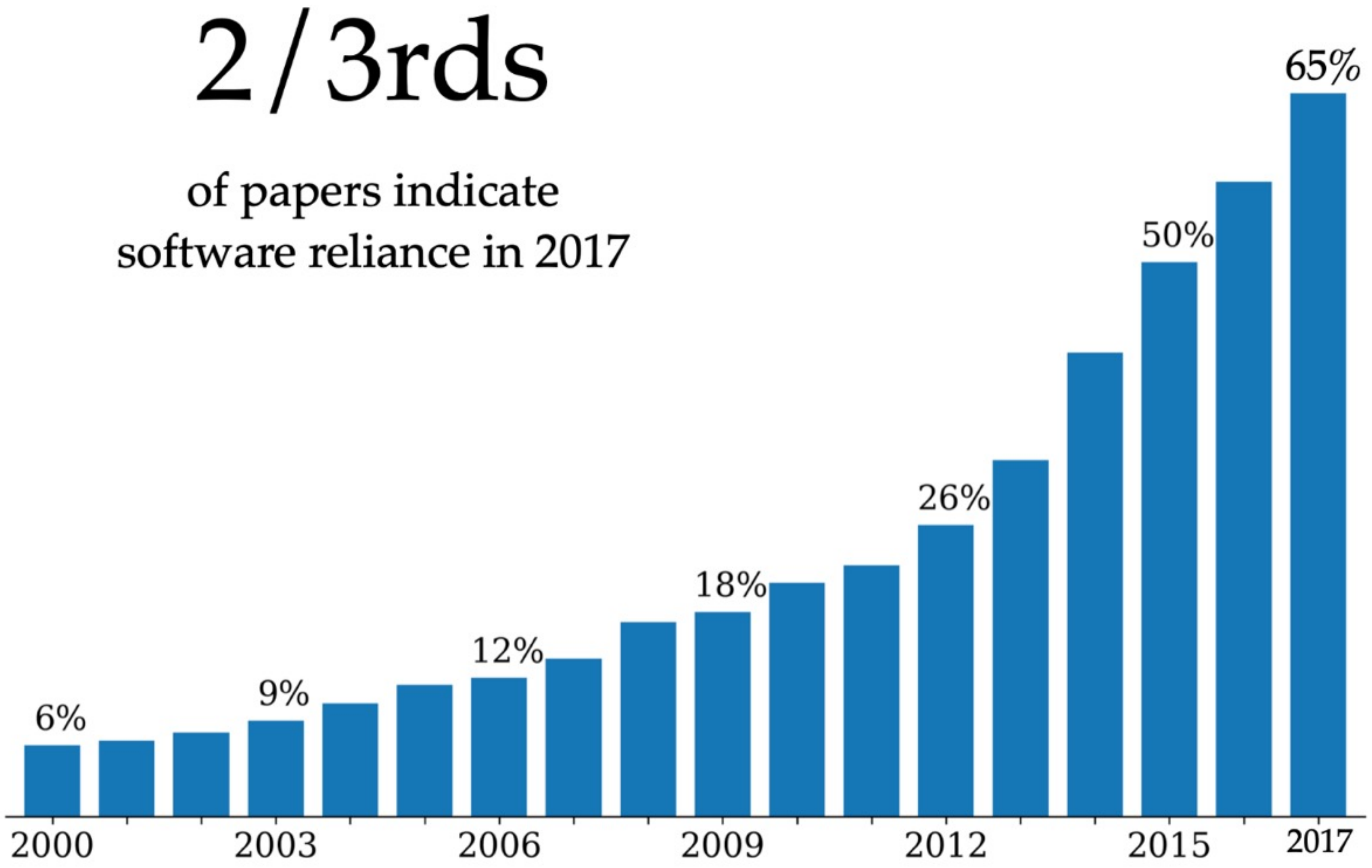



https://de-rse.org/




[photo: pixabay.com]

# Publications relying on software




https://bit.ly/37XEJ2u

# Use of research software

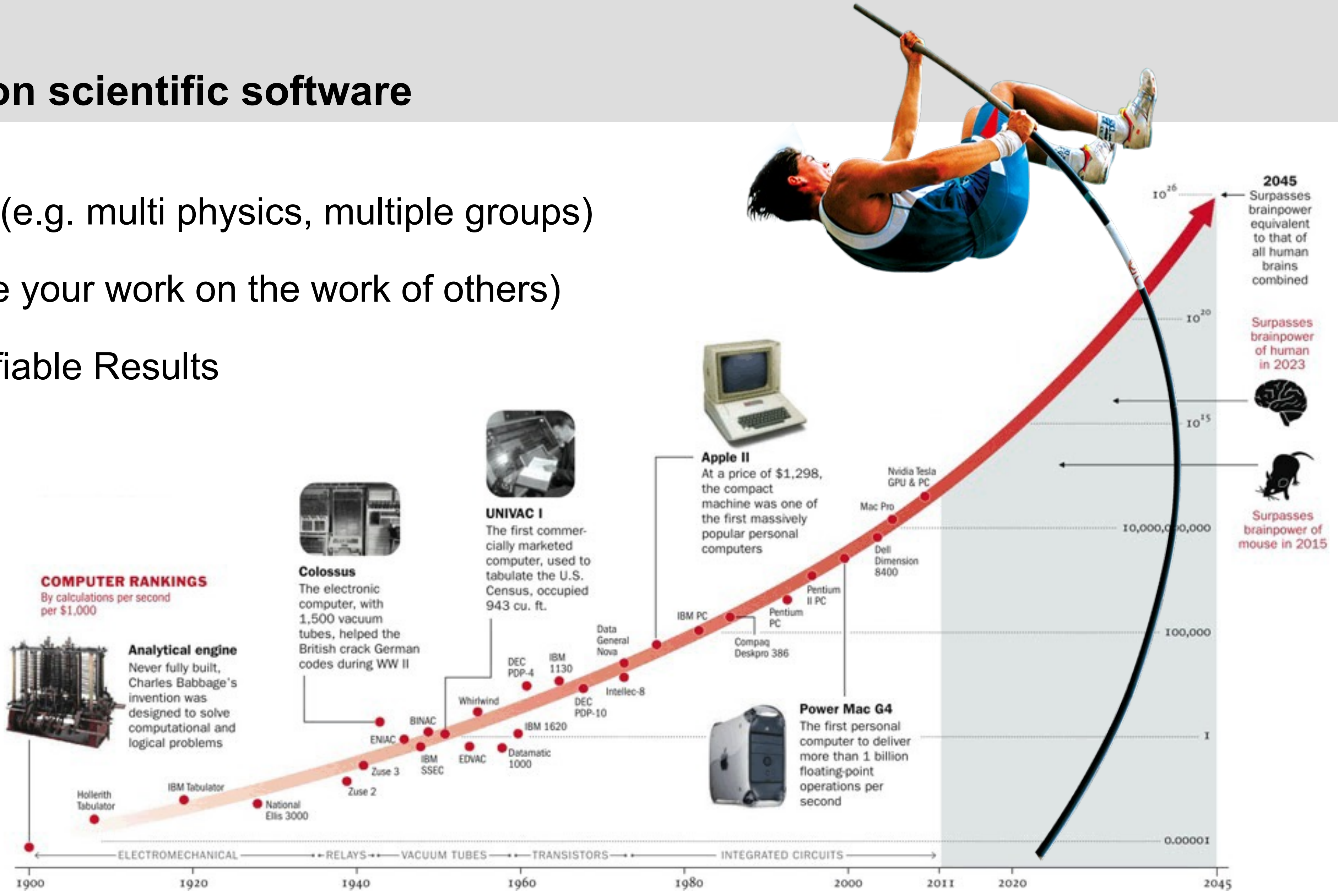
- 92% of academics use research software
- 69% say that their research would not be practical without it
- 56% develop their own software

https://bit.ly/2zZPhSa


https://bit.ly/2BAvzwQ

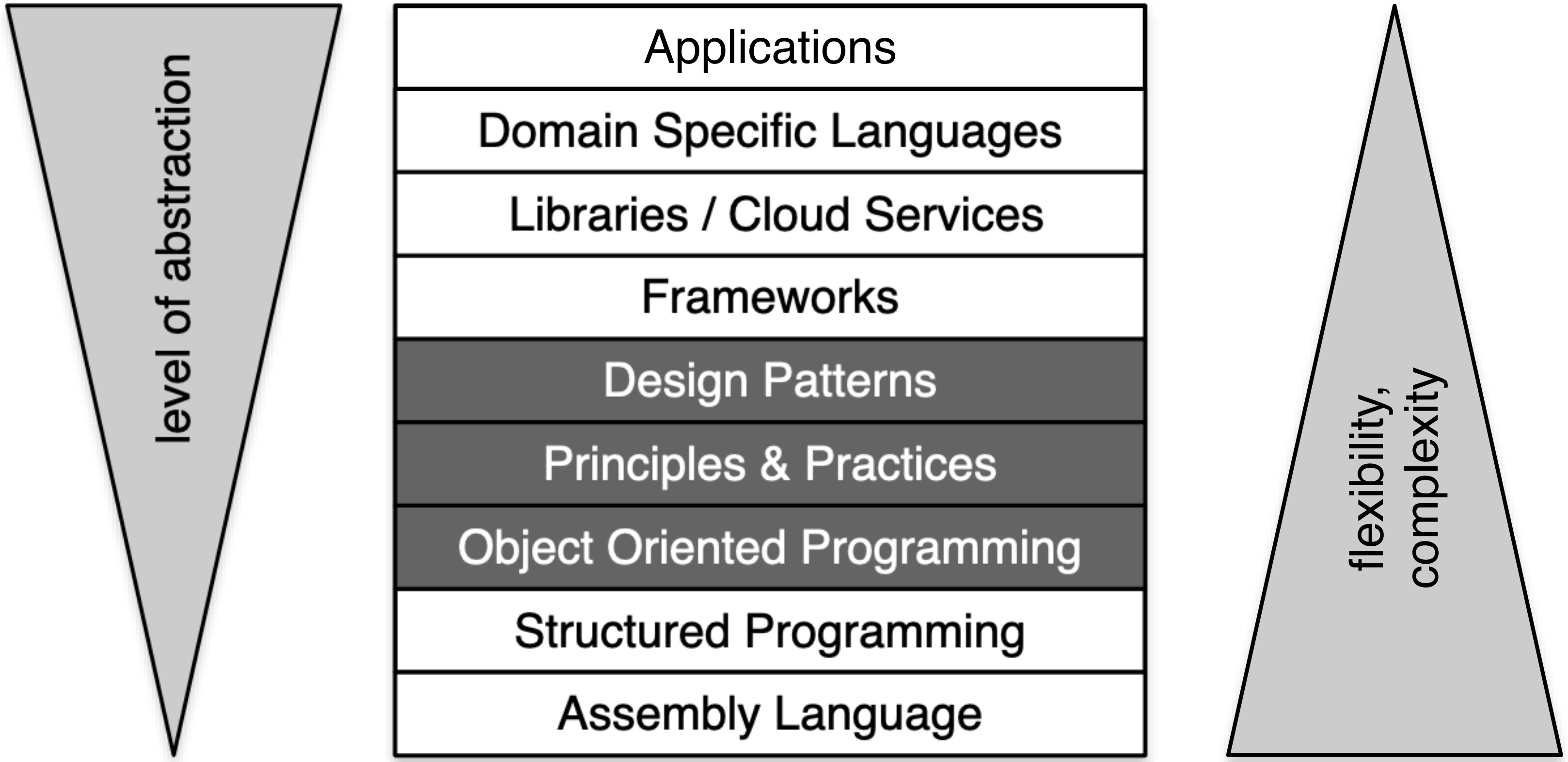
# Growing demands on scientific software

- Increasing Complexity (e.g. multi physics, multiple groups)
- Longer Life Span (base your work on the work of others)
- Reproducible and Verifiable Results



“The function of good software is to make the complex appear to be simple. ”

Grady Booch



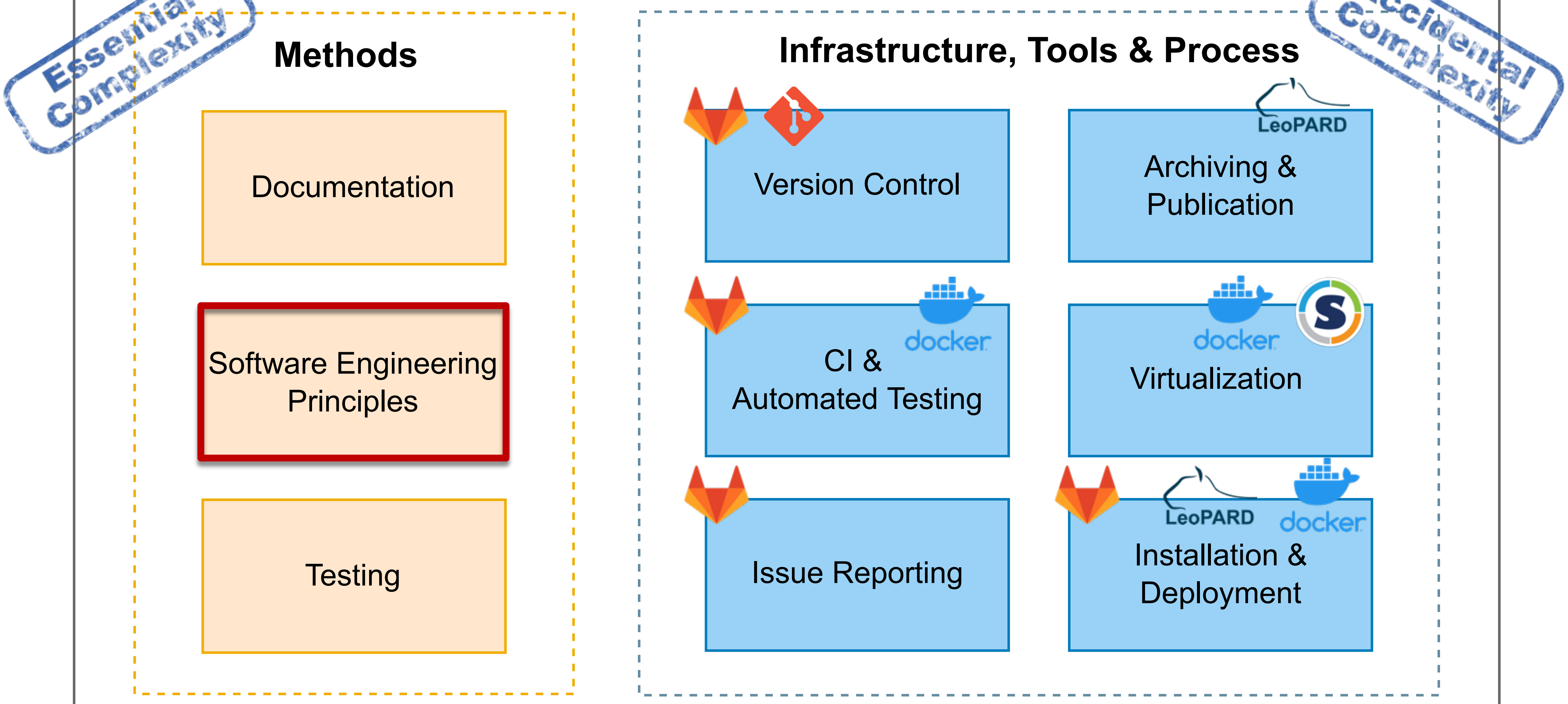
“ The art of programming is the art of organizing complexity. ”

Edsger W. Dijkstra


[Notes On Structured Programming, Edsger W. Dijkstra, 1970]

[Frederick P. Brooks, No Silver Bullet: Essence and Accidents of Software Engineering, 1987]

# SURESOFT Approach for Sustainable Software

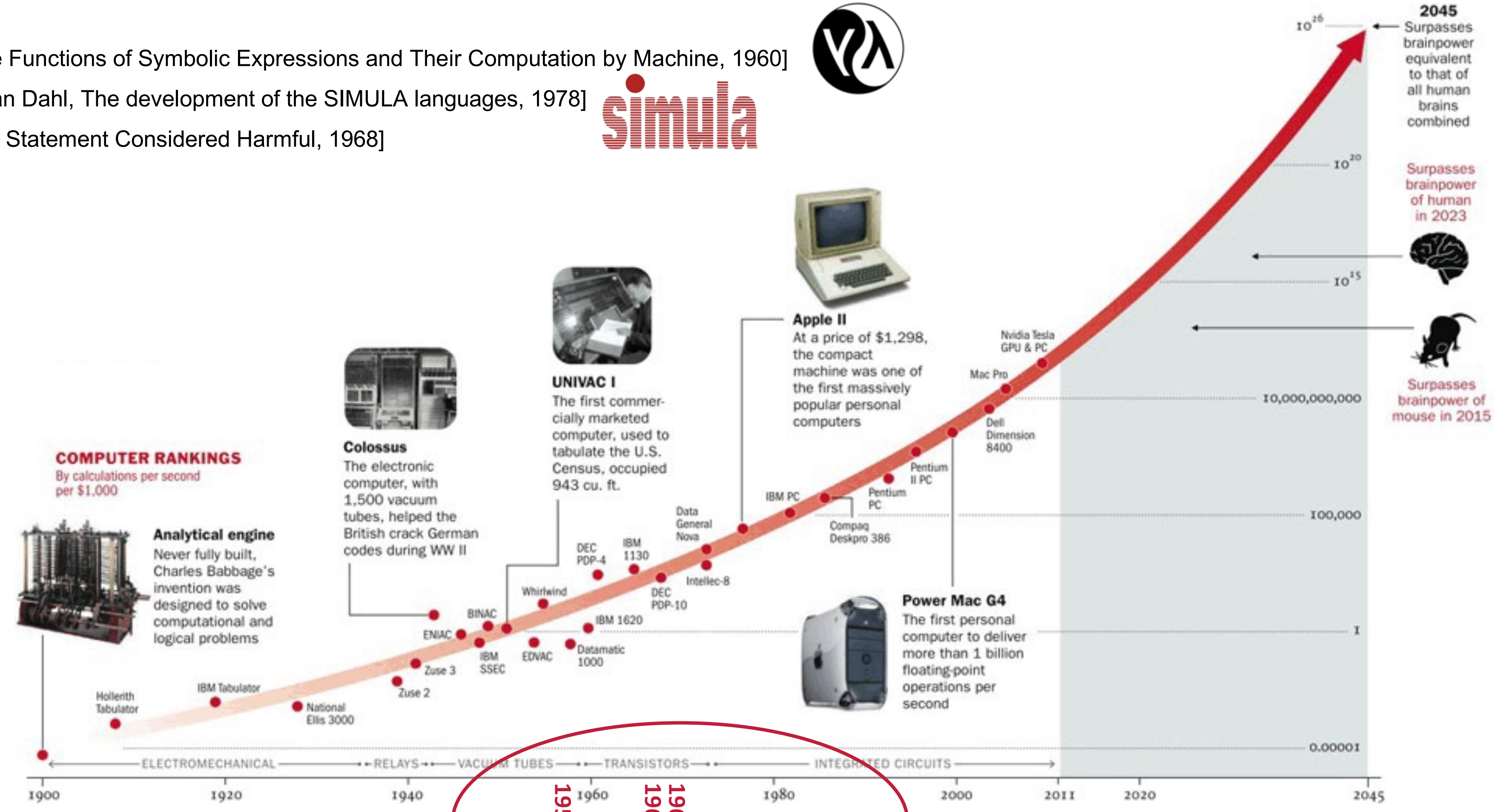


# Growth in technology vs. software development paradigms

[1] [John McCarthy, Recursive Functions of Symbolic Expressions and Their Computation by Machine, 1960]

[2] [Kristen Nygaard, Ole-Johan Dahl, The development of the SIMULA languages, 1978]

[3] [Edsger W. Dijkstra, Go To Statement Considered Harmful, 1968]




1958 1966 1968

LISP[1] OOP[2] Structured Programming[3]

# Take Home Messages

In accordance to Wirth's law one can argue:

“**Software systems grow faster in size and complexity than methods to handle complexity are invented**.”

[Niklaus Wirth, "A Plea for Lean Software”, 1995]

We need to **make the best possible use of the software development techniques available** to cope with the growth in complexity.

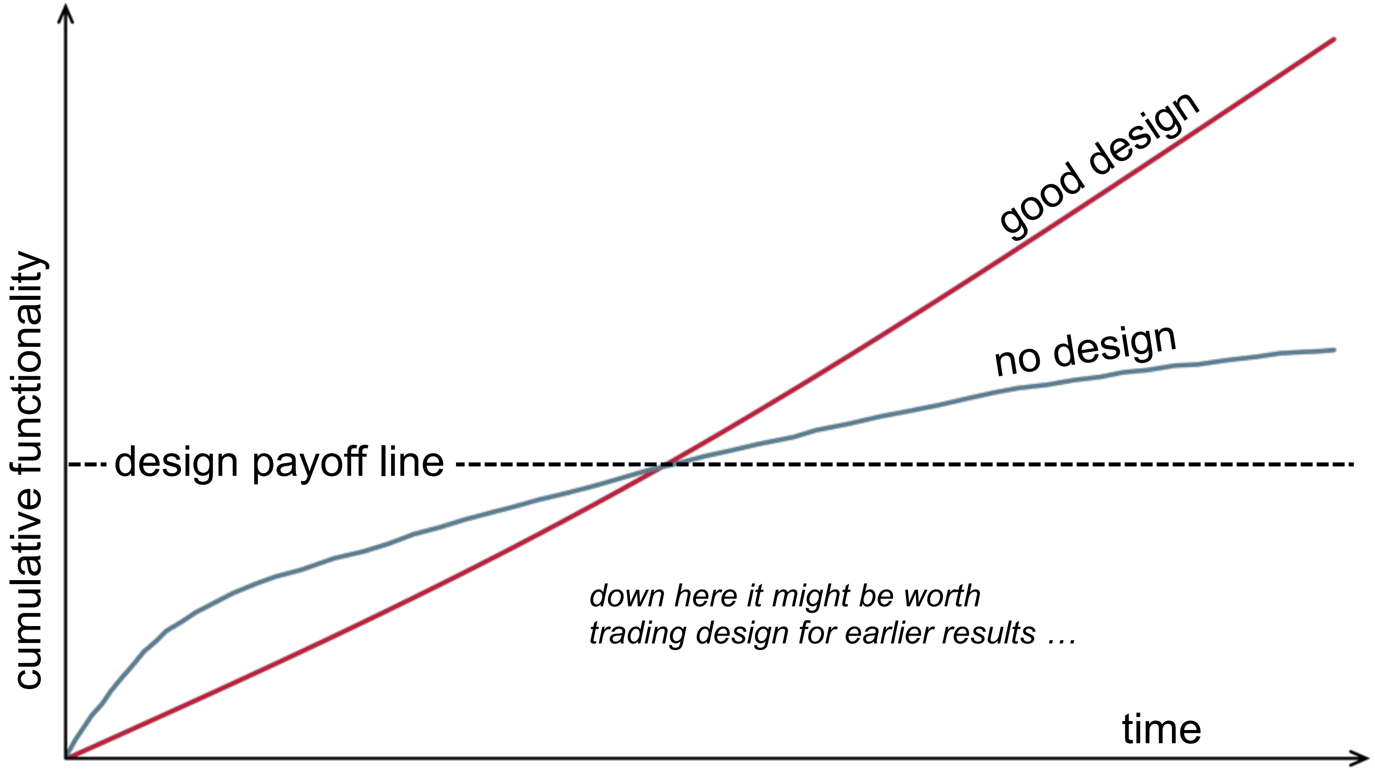
“The gap between the best software engineering practice and the average practice is very wide – perhaps wider than in any other engineering discipline. [...] **The difference between the the great and the average approach an order of magnitude**.”

[Frederick P. Brooks, No Silver Bullet: Essence and Accidents of Software Engineering, 1987]

# Productivity Crisis

- floating point performance is constantly rising
- time-to-solution is inceasing
- scientists spend 50% of the time finding bugs

[P. Prabhu, A Survey of the Practice of Computational Science, 2011]




Design Stamina Hypothesis - https://bit.ly/2A64CAR

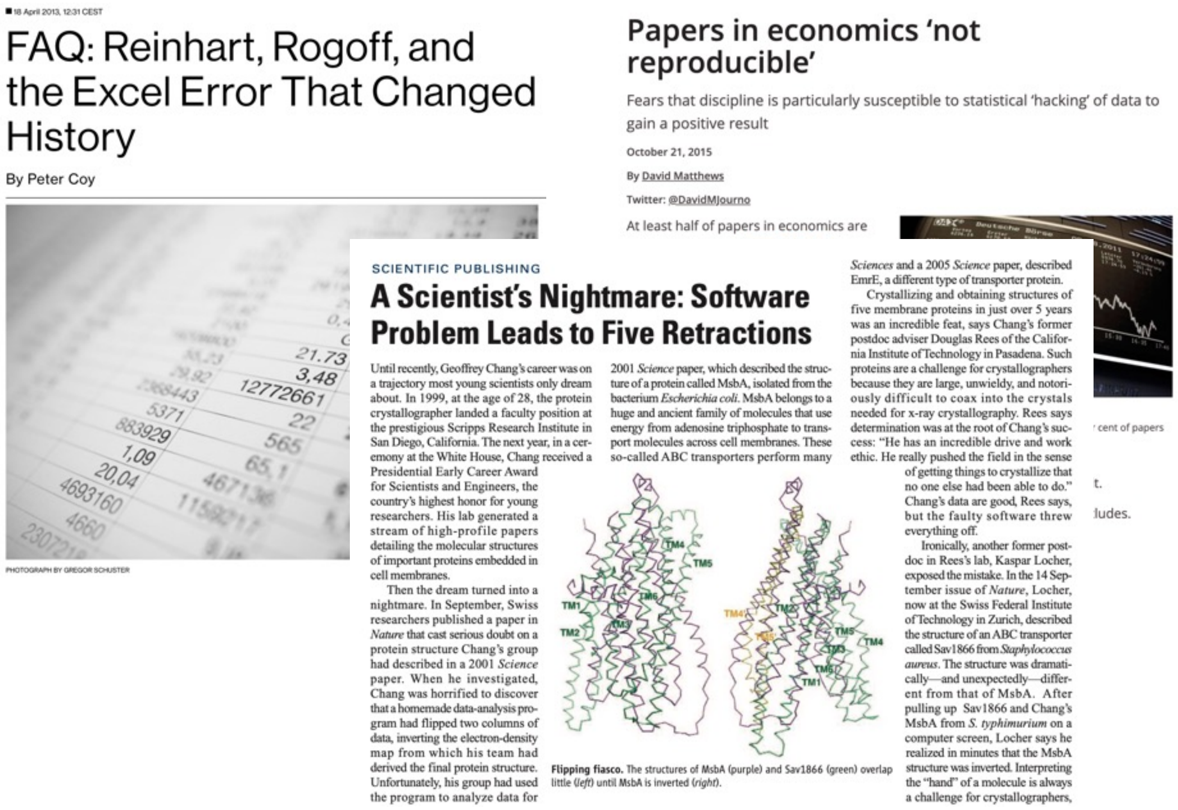
# Productivity Crisis

“ The only way to go **fast** is to go **well**.”

Robert C. Martin

# Credibility Crisis

Questionable reliability, accuracy, reproducibility and verifiability of the results …



18 April 2013, 12:31 CEST

**FAQ: Reinhart, Rogoff, and the Excel Error That Changed History**

By Peter Coy

PHOTOGRAPH BY GREGOR SCHUSTER

**Papers in economics 'not reproducible'**

Fears that discipline is particularly susceptible to statistical 'hacking' of data to gain a positive result

October 21, 2015

By David Matthews

Twitter: @DavidMJourno

At least half of papers in economics are

SCIENTIFIC PUBLISHING

**A Scientist's Nightmare: Software Problem Leads to Five Retractions**

Until recently, Geoffrey Chang's career was on a trajectory most young scientists only dream about. In 1999, at the age of 28, the protein crystallographer landed a faculty position at the prestigious Scripps Research Institute in San Diego, California. The next year, in a ceremony at the White House, Chang received a Presidential Early Career Award for Scientists and Engineers, the country's highest honor for young researchers. His lab generated a stream of high-profile papers detailing the molecular structures of important proteins embedded in cell membranes.

Then the dream turned into a nightmare. In September, Swiss researchers published a paper in *Nature* that cast serious doubt on a protein structure Chang's group had described in a 2001 *Science* paper. When he investigated, Chang was horrified to discover that a homemade data-analysis program had flipped two columns of data, inverting the electron-density map from which his team had derived the final protein structure. Unfortunately, his group had used the program to analyze data for

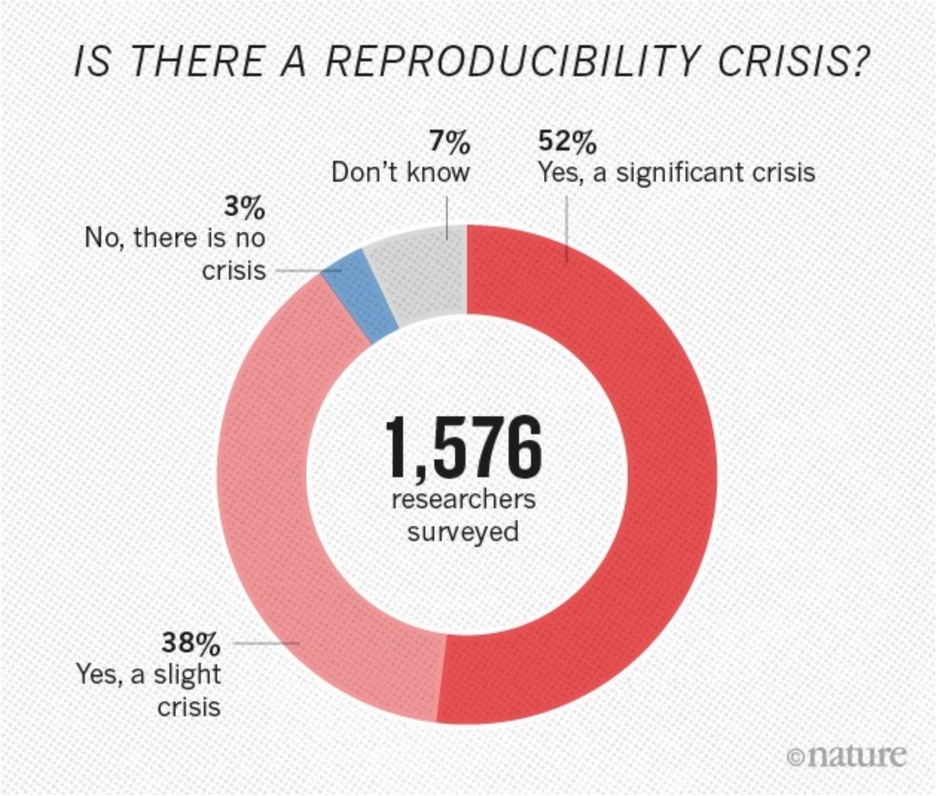
2001 *Science* paper, which described the structure of a protein called MsbA, isolated from the bacterium *Escherichia coli*. MsbA belongs to a huge and ancient family of molecules that use energy from adenosine triphosphate to transport molecules across cell membranes. These so-called ABC transporters perform many

**Flipping fiasco.** The structures of MsbA (purple) and Sav1866 (green) overlap little (*left*) until MsbA is inverted (*right*).

*Sciences* and a 2005 *Science* paper, described EmrE, a different type of transporter protein.

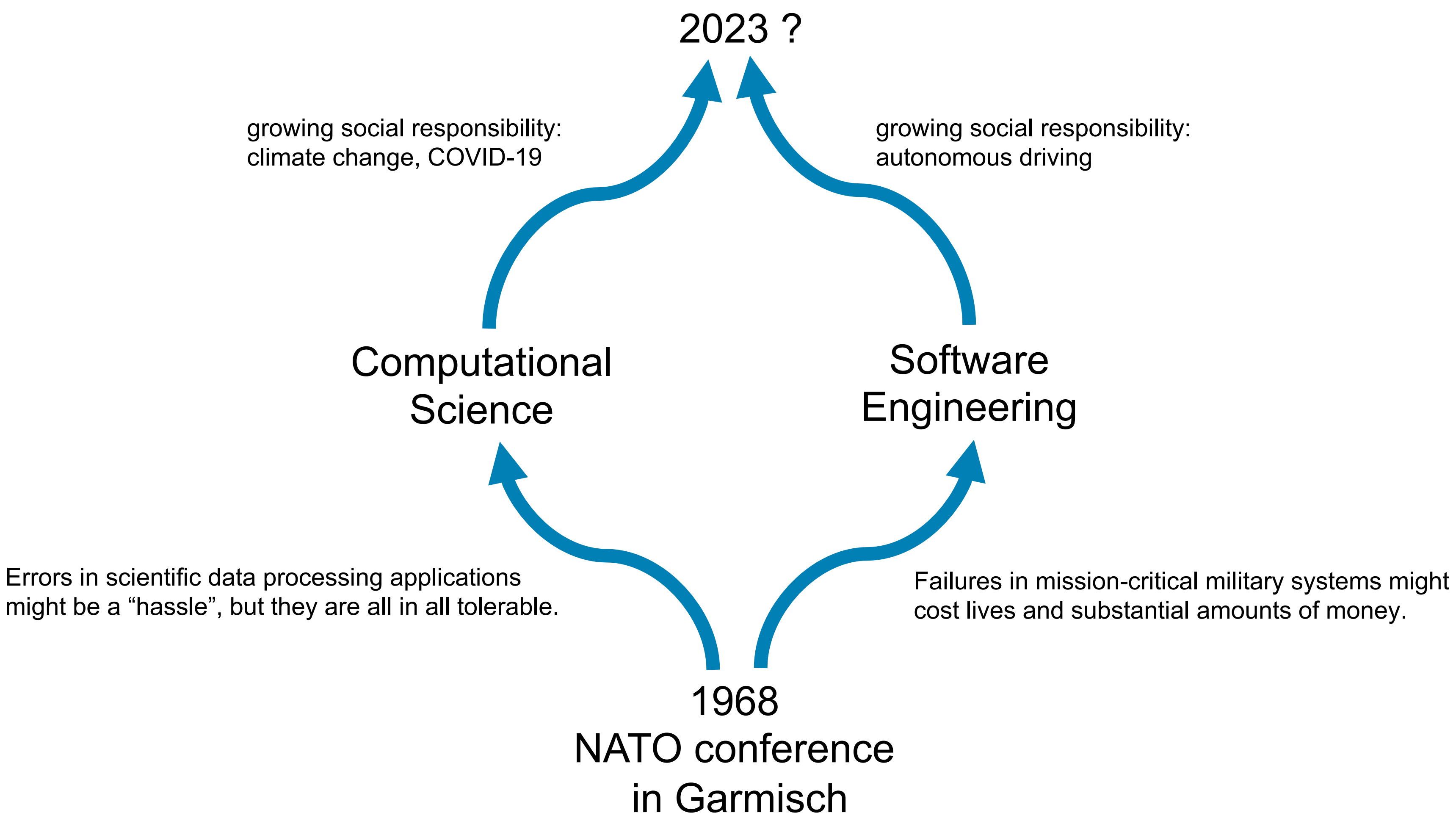
Crystallizing and obtaining structures of five membrane proteins in just over 5 years was an incredible feat, says Chang's former postdoc adviser Douglas Rees of the California Institute of Technology in Pasadena. Such proteins are a challenge for crystallographers because they are large, unwieldy, and notoriously difficult to coax into the crystals needed for x-ray crystallography. Rees says determination was at the root of Chang's success: "He has an incredible drive and work ethic. He really pushed the field in the sense of getting things to crystallize that no one else had been able to do." Chang's data are good, Rees says, but the faulty software threw everything off.

Ironically, another former postdoc in Rees's lab, Kaspar Locher, exposed the mistake. In the 14 September issue of *Nature*, Locher, now at the Swiss Federal Institute of Technology in Zurich, described the structure of an ABC transporter called Sav1866 from *Staphylococcus aureus*. The structure was dramatically—and unexpectedly—different from that of MsbA. After pulling up Sav1866 and Chang's MsbA from *S. typhimurium* on a computer screen, Locher says he realized in minutes that the MsbA structure was inverted. Interpreting the "hand" of a molecule is always a challenge for crystallographers,




https://go.nature.com/2DgtDKR

# Birth of Software Engineering discipline in 1968




[Naur, Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 1968]

# Start caring about your code

“Clean code always looks like
it was written by someone who cares.”

Michael Feathers



[Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, 2008]

# Software Entropy - Broken Window Effect

Broken window effect: https://bit.ly/2BddXYh

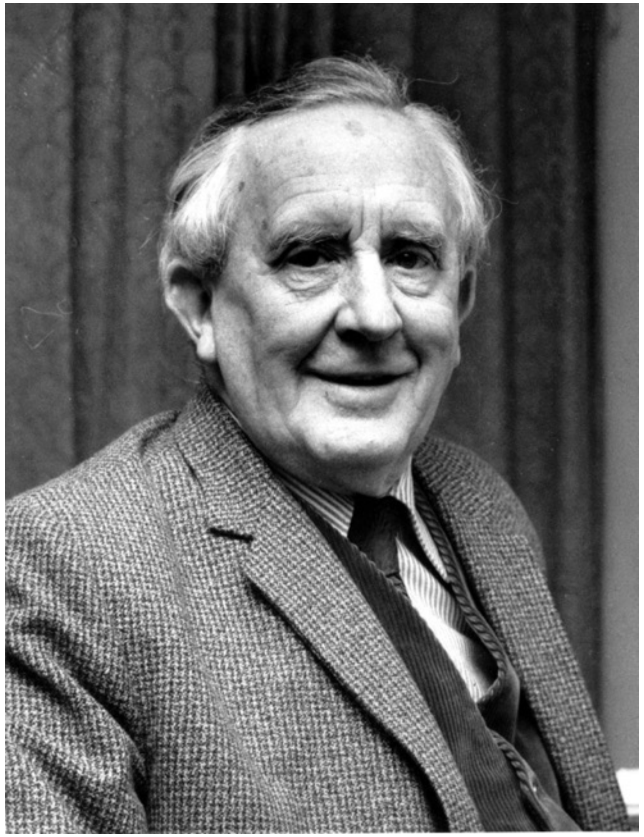
[A. Hunt, The Pragmatic Programmer: From Journeyman to Master, 1999]

# The Boy Scout Rule

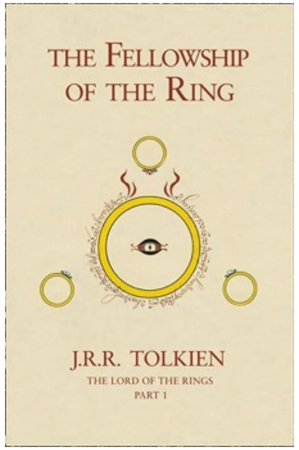
"Leave the campground cleaner than you found it."


[Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, 2008]

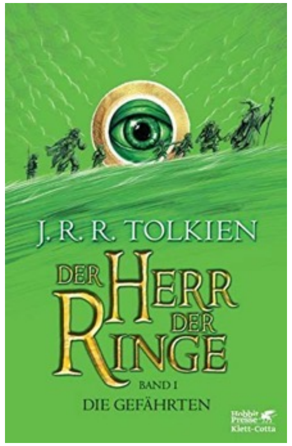
# Today it's all about concepts…

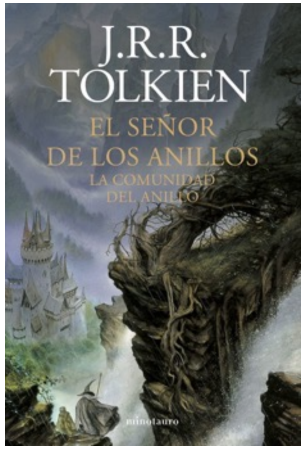


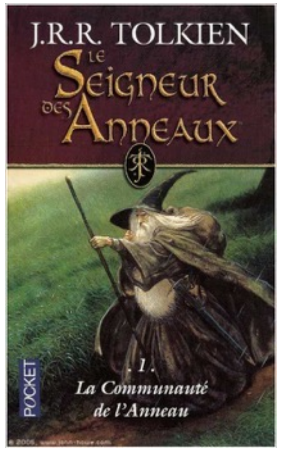
J.R.R. Tolkien (1892 – 1973)








It needs more than knowing a language to be a successful author!

# External and Internal Software Quality

## Two Categories of Quality

### External Quality Factors

- aim to the needs of a user

### Internal Quality Factors

- aim to the needs of the developers

Implicit dependencies of several quality factors prevent the maximization of all factors.

**Engineering task:** Optimal balancing of quality goals.

[James McCall, Factors in Software Quality, Technical Report, General Electric, 1977]

[C.A.R. Hoare, The Quality of Software, in Software, Practice and Experience, 1972]

[Barry W. Boehm, J.R. Brown, G. McLeod, Myron Lipow and M. Merrit: Characteristics of Software Quality, 1978]

# External Quality Factors - J. McCall, 1977

- **Correctness** - The degree to which a system is free from faults in its specification, design, and implementation.
- **Usability** - The ease with which users can learn and use a system.
- **Efficiency** - Minimal use of system resources, including memory and execution time.
- **Reliability** - The ability of a system to perform its required functions under stated conditions whenever required—having a long mean time between failures.
- **Integrity** - The degree to which a system prevents unauthorized or improper access to its programs and its data.
- **Adaptability** - The extent to which a system can be used, without modification, in applications or environments other than those for which it was originally designed.
- **Accuracy** - The degree to which a system, as built, is free from error, especially with respect to quantitative outputs. Accuracy differs from correctness; it is a determination of how well a system does the job it's built for rather than whether it was built correctly.
- **Robustness** - The degree to which a system continues to function in the presence of invalid inputs or stressful environmental conditions.

# Internal Quality Factors - J. McCall, 1977

**“The key to achieving these external factors is in the internal ones.”**

– B. Meyer, 1988

- **Maintainability** - The ease with which you can modify a software system to change or add capabilities, improve performance, or correct defects.
- **Flexibility** - The extent to which you can modify a system for uses (or environments) other than those for which it was specifically designed.
- **Portability** - The ease with which you can modify a system to operate in an environment different from that for which it was specifically designed.
- **Reusability** - The extent to which and the ease with which you can use parts of a system in other systems.
- **Readability** - The ease with which you can read and understand the source code of a system, especially at the detailed-statement level.
- **Testability** - The degree to which you can unit-test and system-test a system; the degree to which you can verify that the system meets its requirements.
- **Understandability** - The ease with which you can comprehend a system at both the system-organizational and detailed-statement levels. Understandability has to do with the coherence of the system at a more general level than readability does.

[Bertrand Meyer, Object-Oriented Software Construction, 1988]

# Quality characteristics valued by scientists – survey by J. Carver, 2007

- functional correctness
- performance
- portability
- maintainability

[Jeffrey Carver, Software Development Environments for Scientific and Engineering Software: A Series of Case Studies, 2007]

# What do you value the most?

a) functional correctness
b) performance
c) portability
d) maintainability
e) other

Join at
slido.com
#123 444

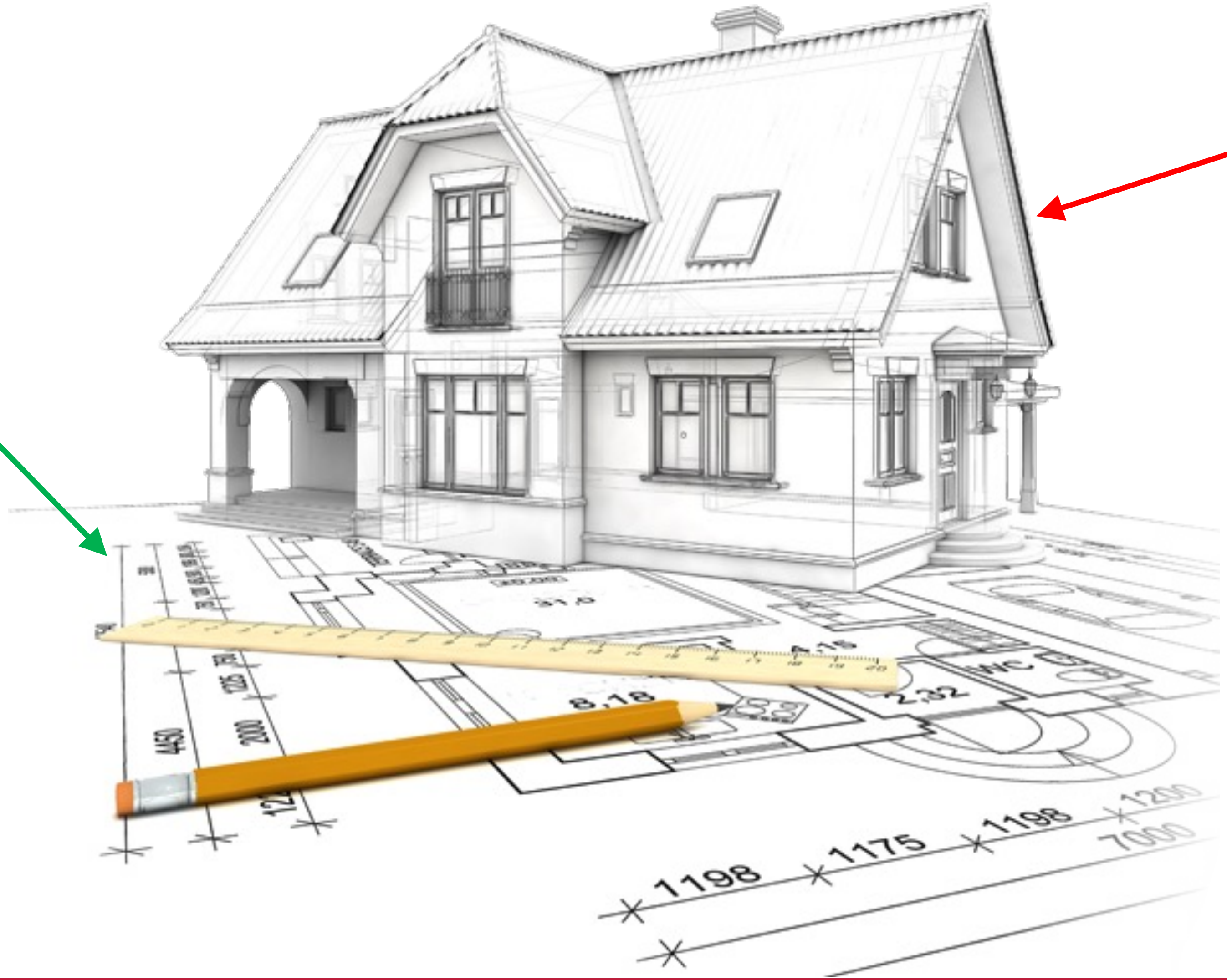
Technische Universität Braunschweig



Suresoft
SUSTAINABLE RESEARCH SOFTWARE

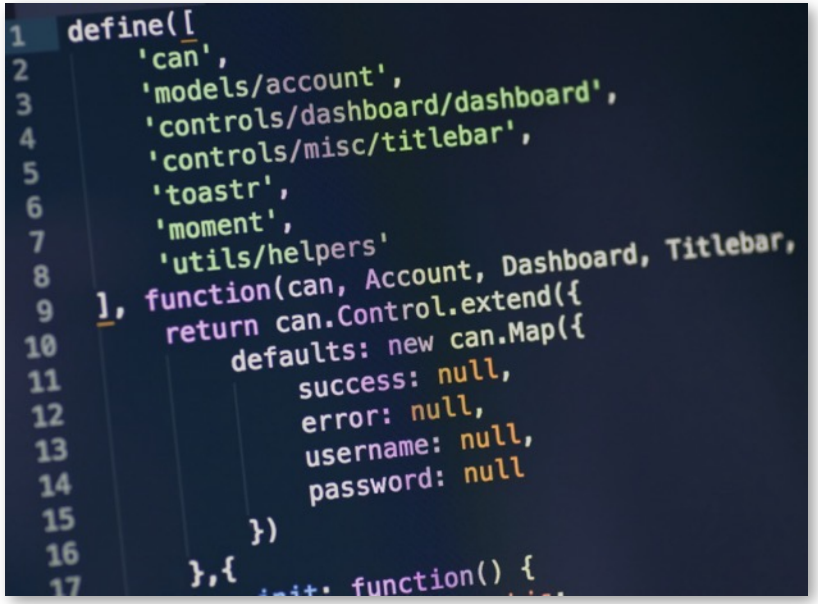
# Traditional vs. Agile Processes

**Design**
cheap



**Product**
expensive

# Traditional vs. Agile Processes



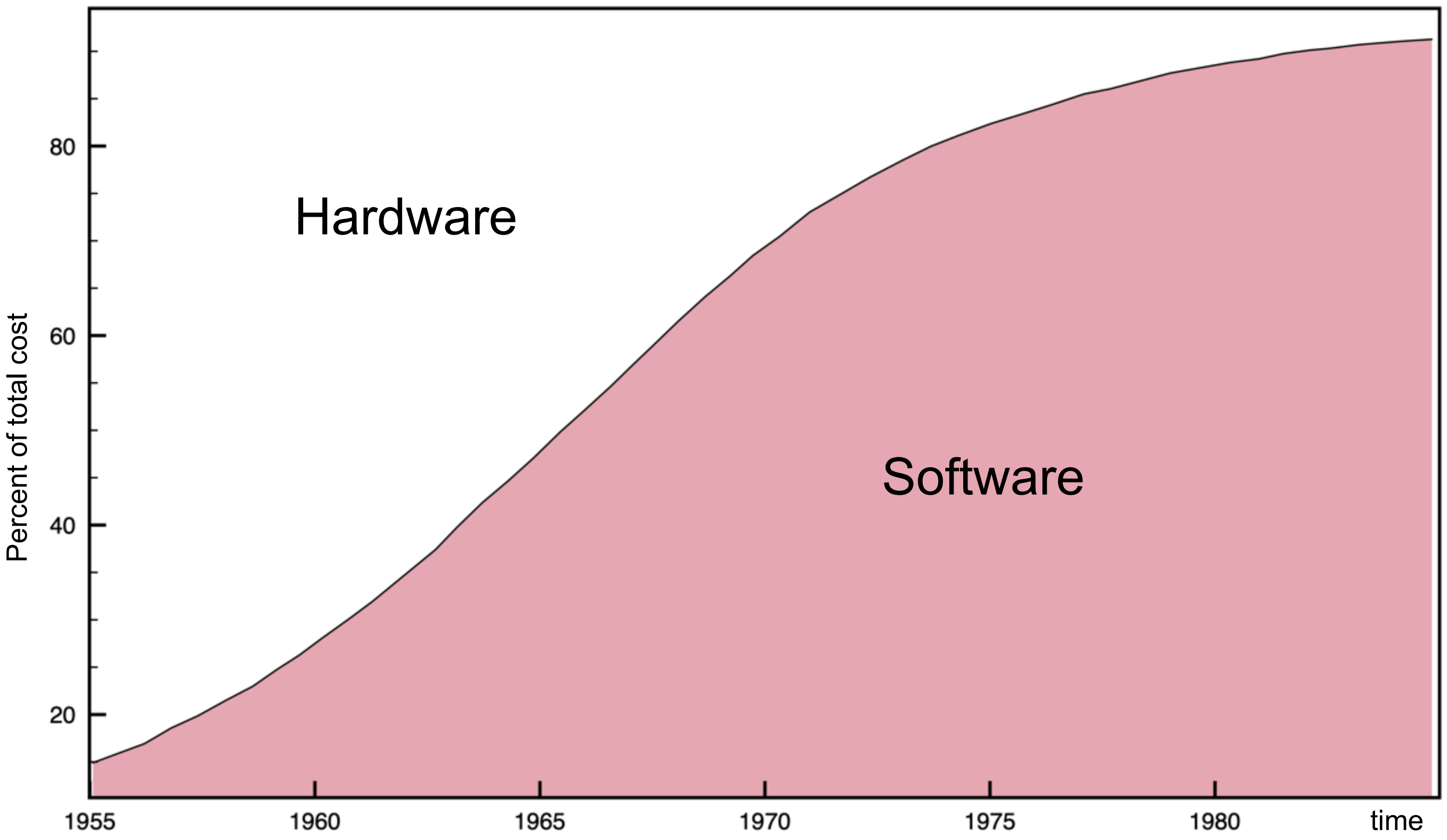


**Design**
expensive

**Product**
cheap

[photo: pixabay.com]

# Evolution of costs for Hardware vs. Software

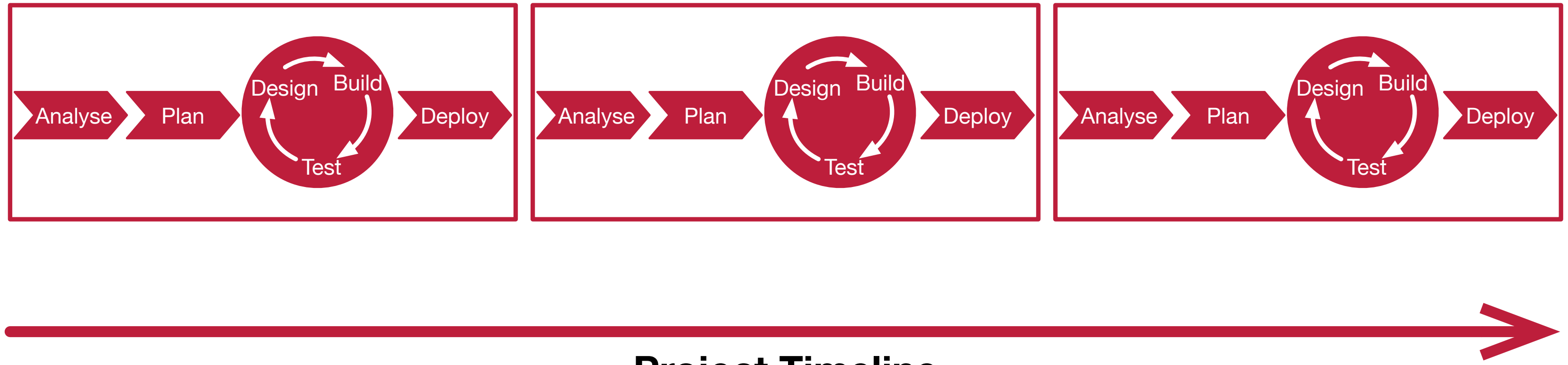



[Barry W. Boehm, Software and its impact: A quantitative assessment, 1973]

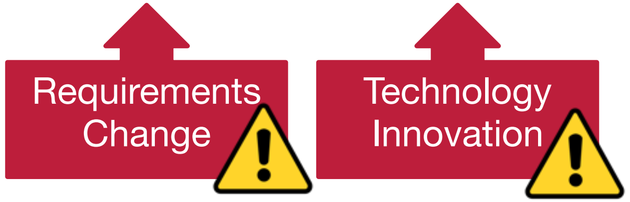
# Traditional vs. Agile Processes

## Plan-Driven (Waterfall)


## Agile




**Project Timeline**



# Quality characteristics valued by scientists – survey by J. Carver, 2007

- functional correctness
- performance
- portability
- maintainability

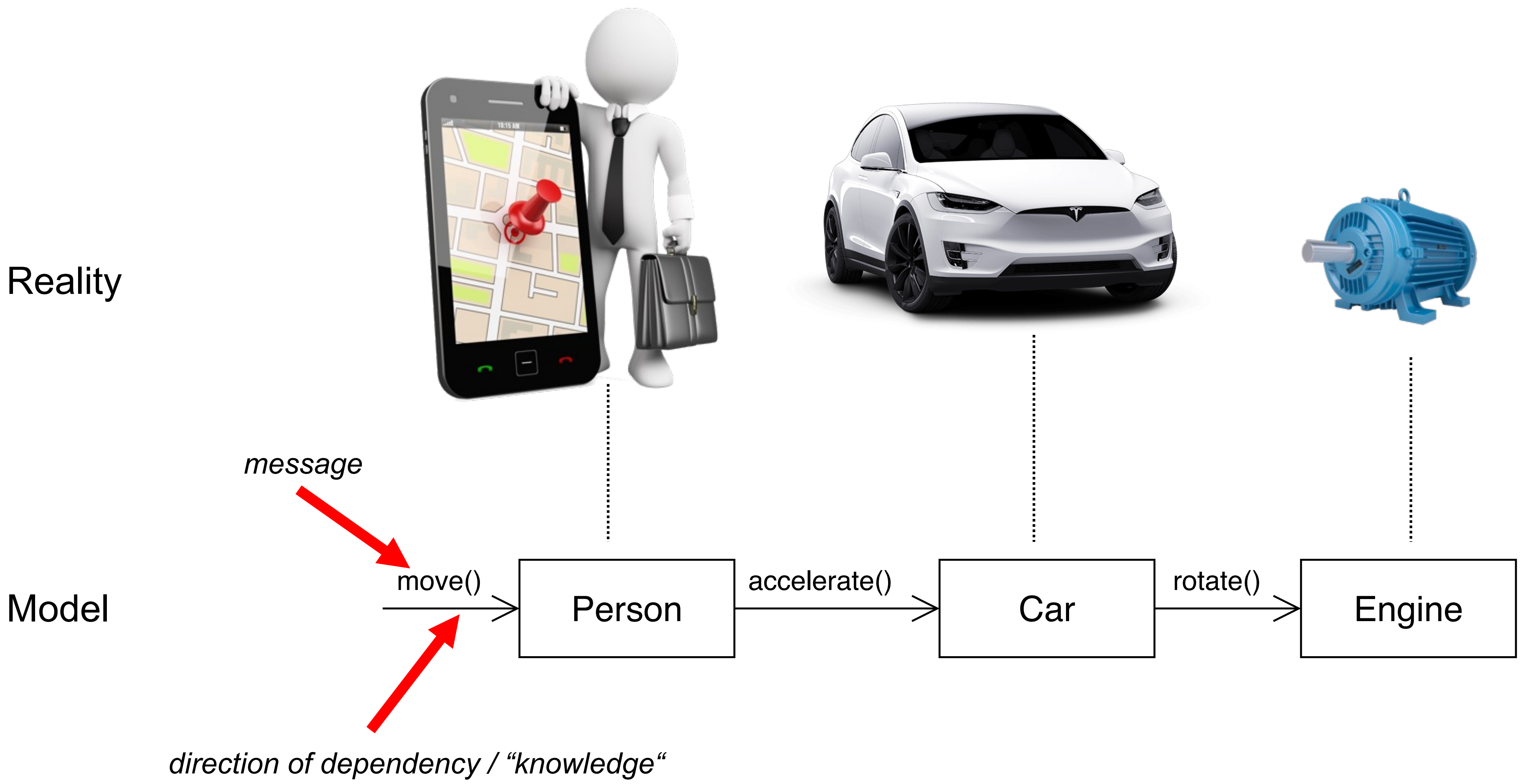
[Jeffrey Carver, Software Development Environments for Scientific and Engineering Software: A Series of Case Studies, 2007]

The one constant in **Soft**ware development is

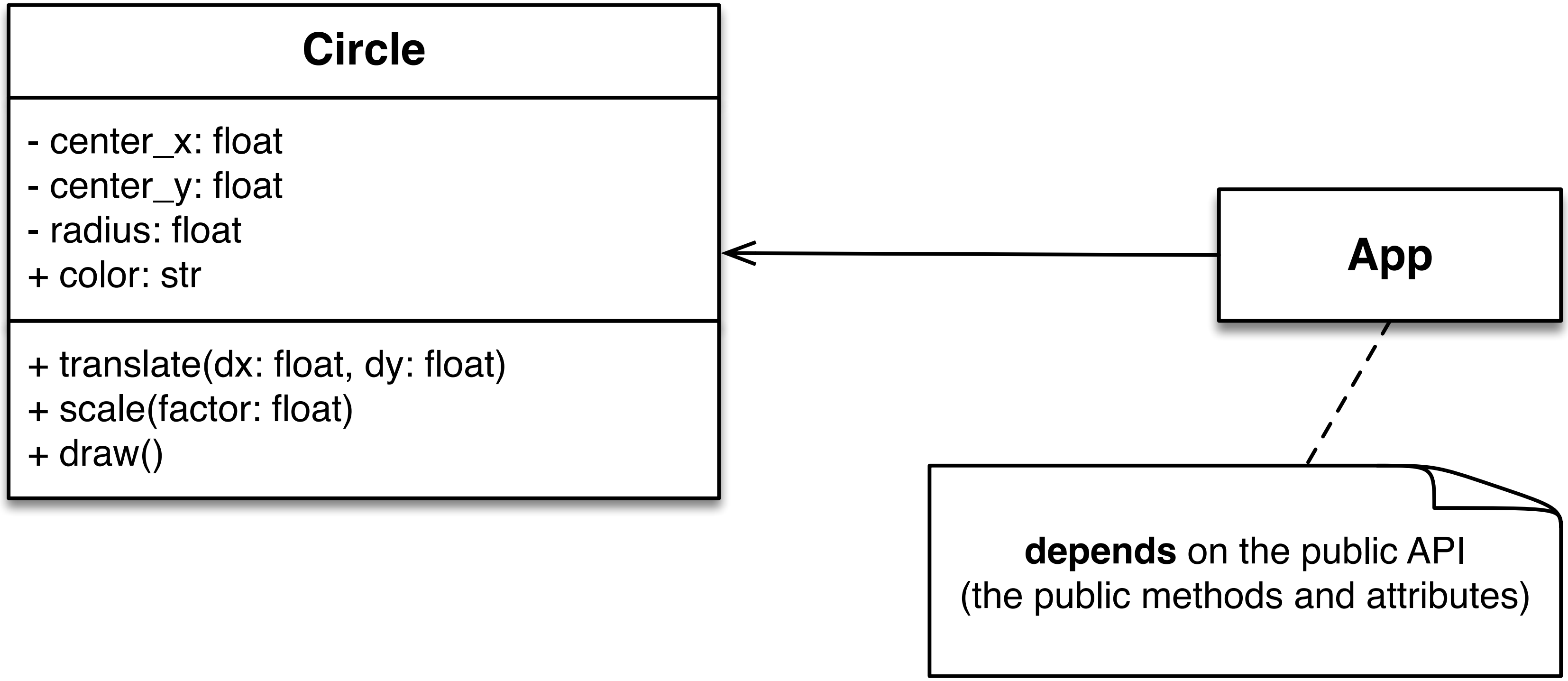
# CHANGE

https://bit.ly/3BGi1cU

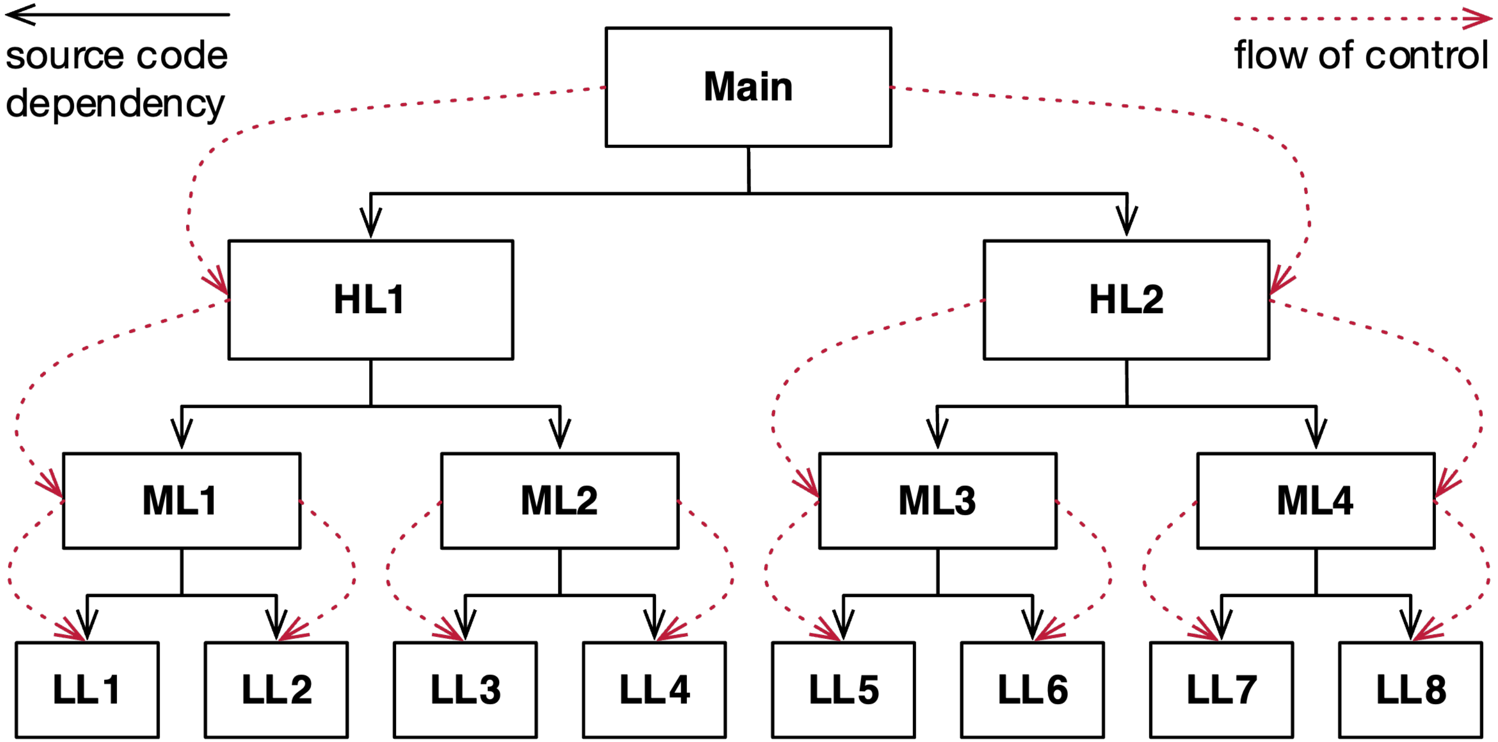
# Object Oriented Programming – It‘s all about messages



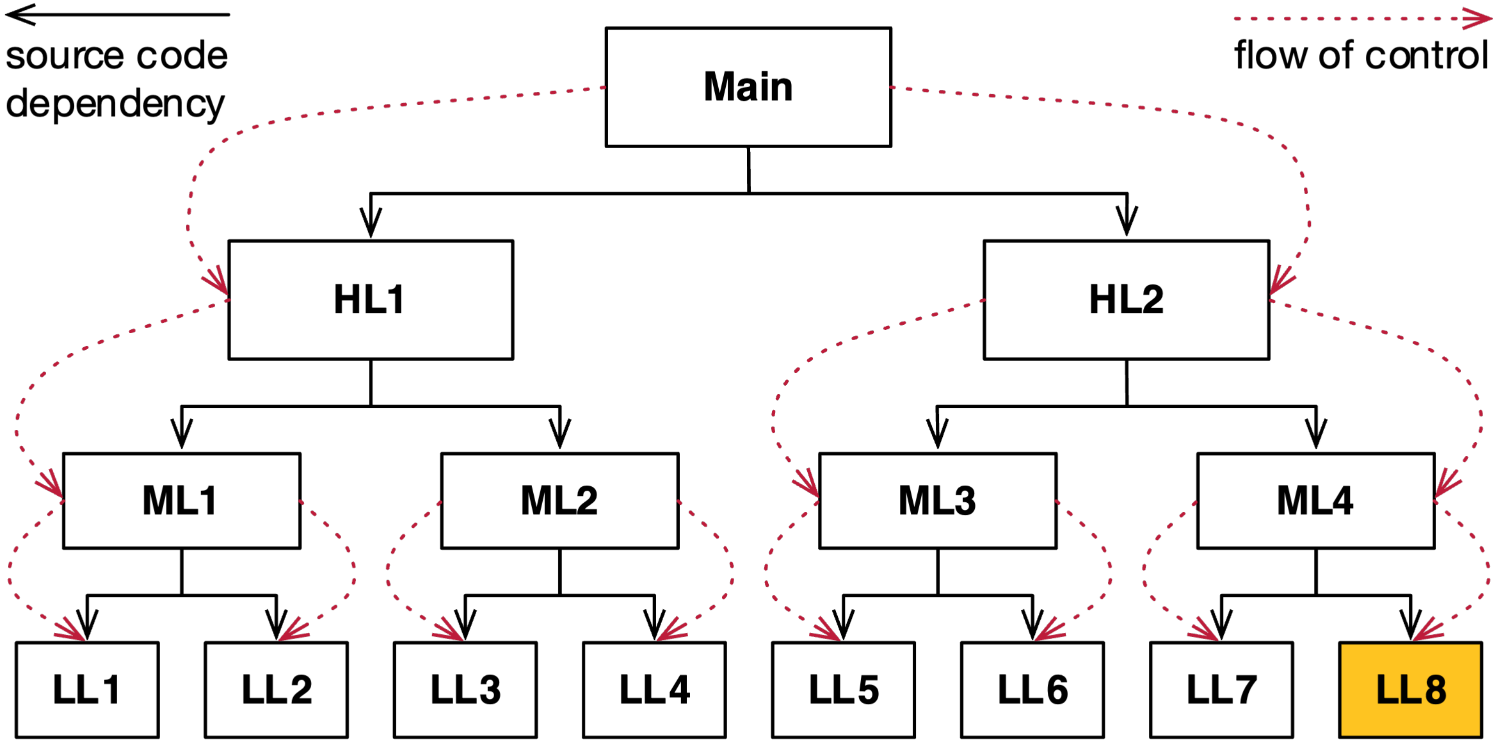
# Dependencies in code



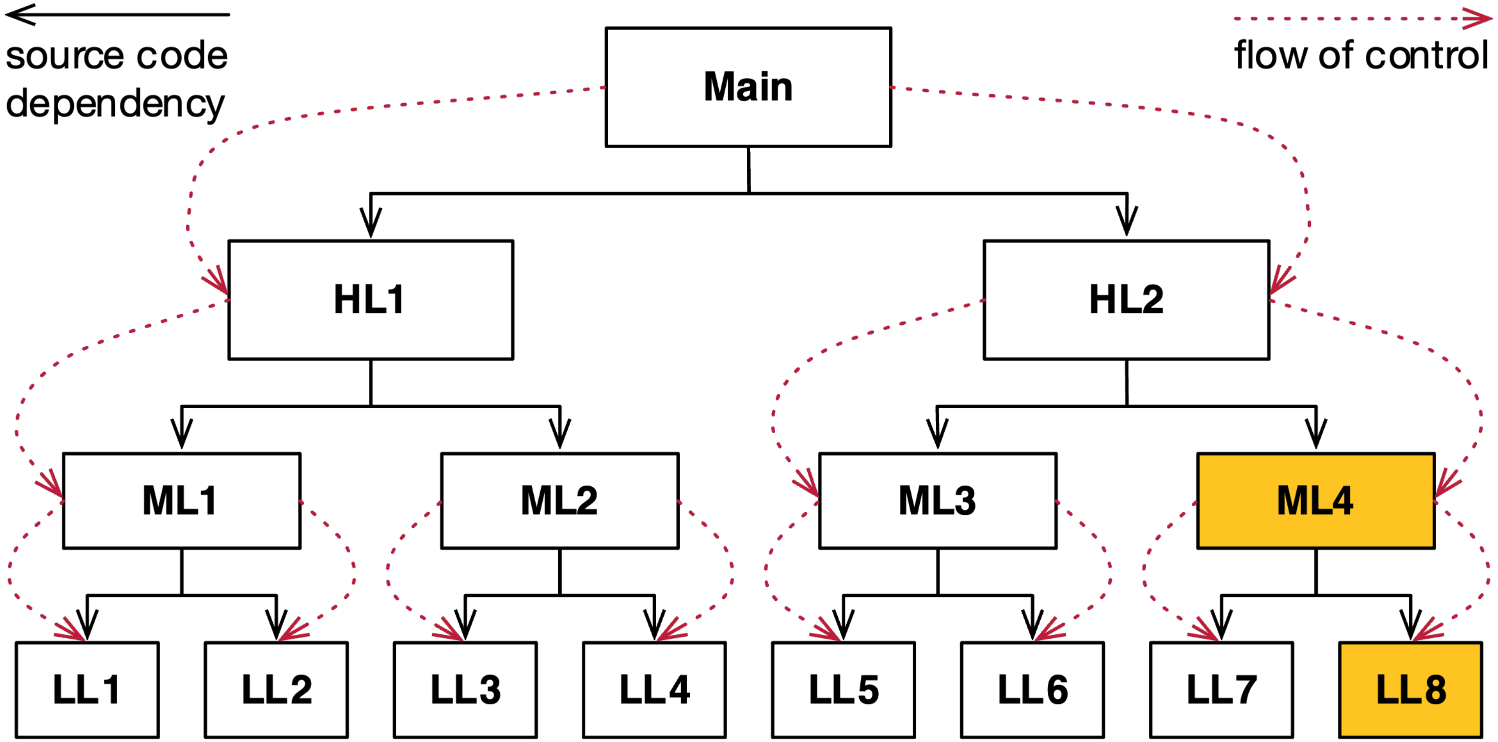
# Dependencies make changes difficult



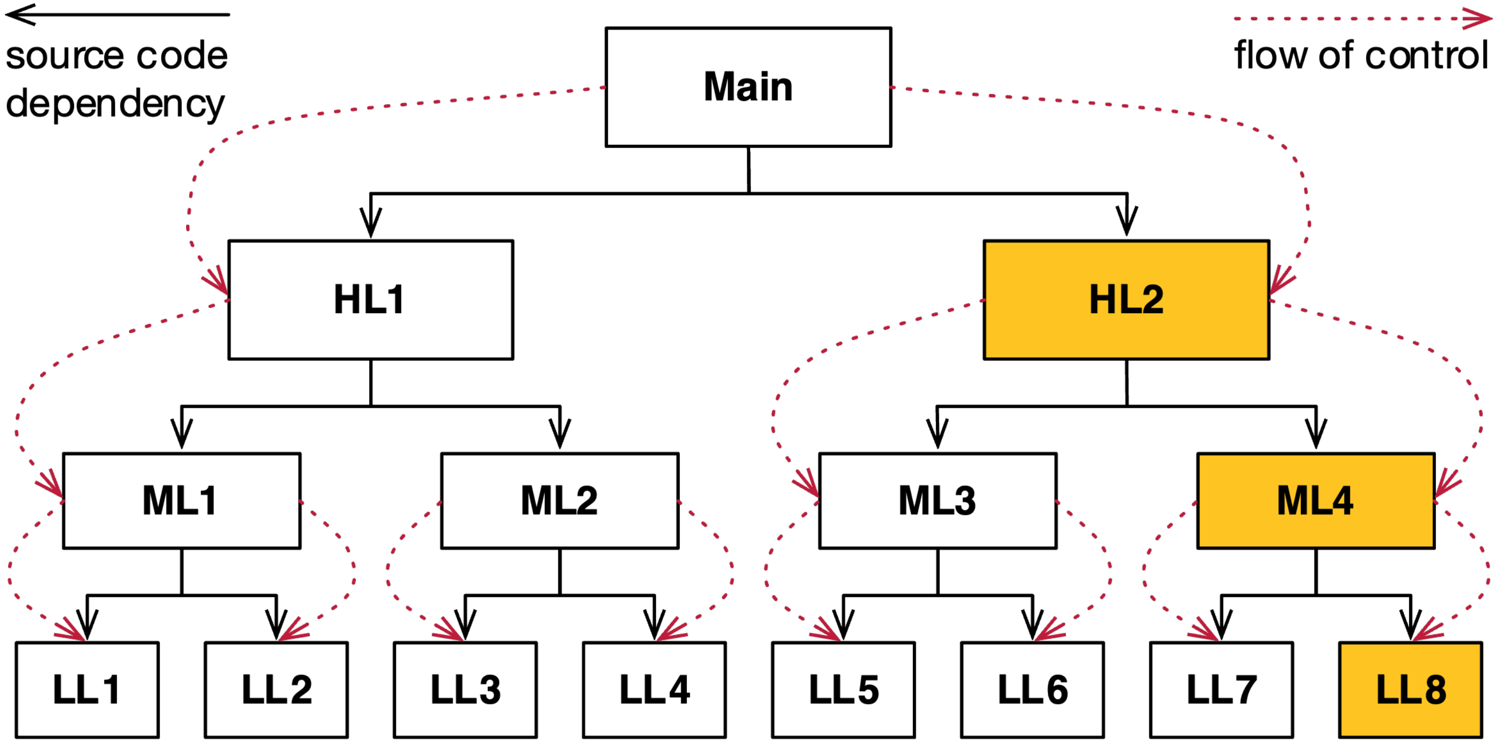
# Dependencies propagate change



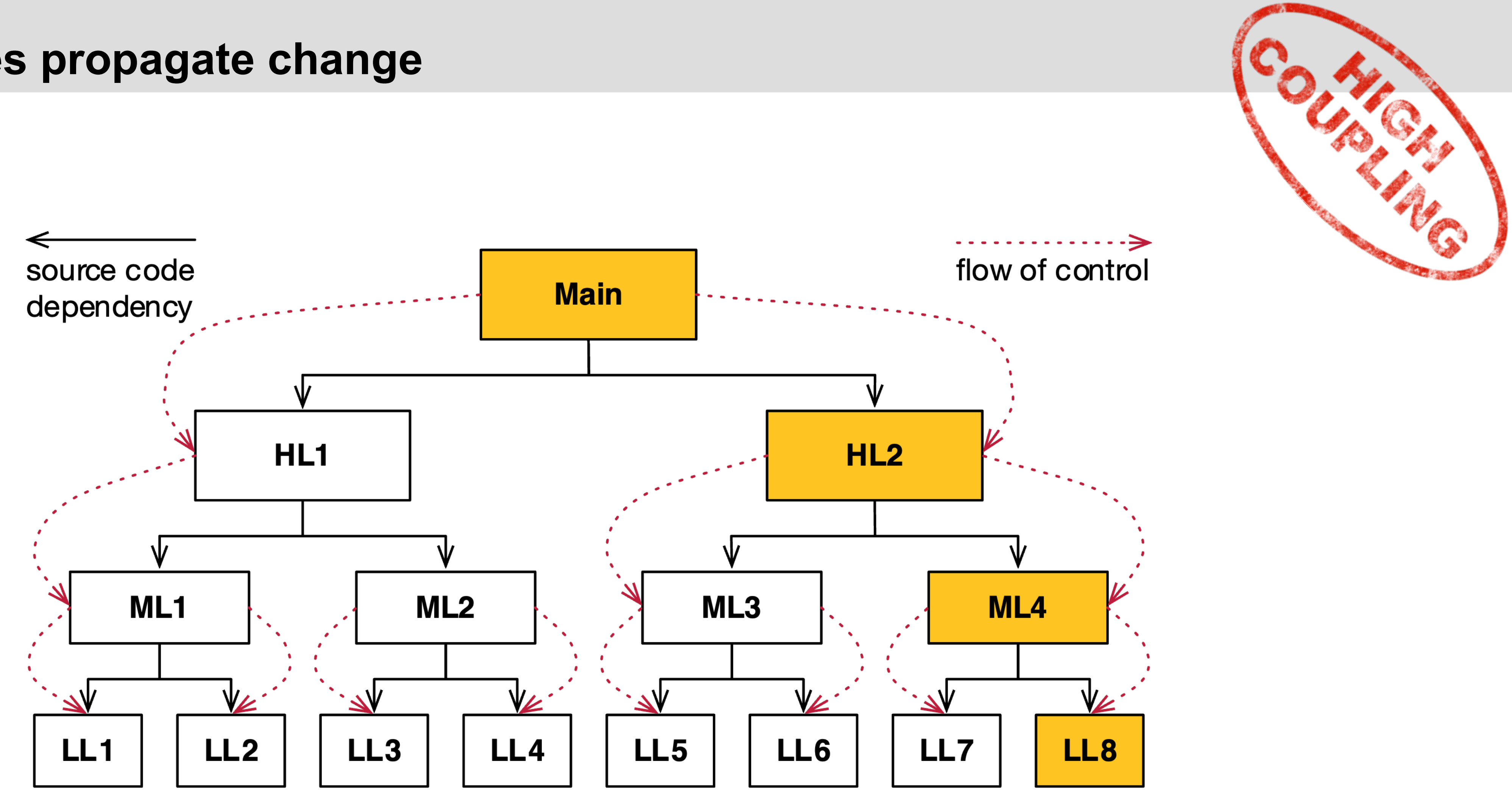
# Dependencies propagate change



# Dependencies propagate change



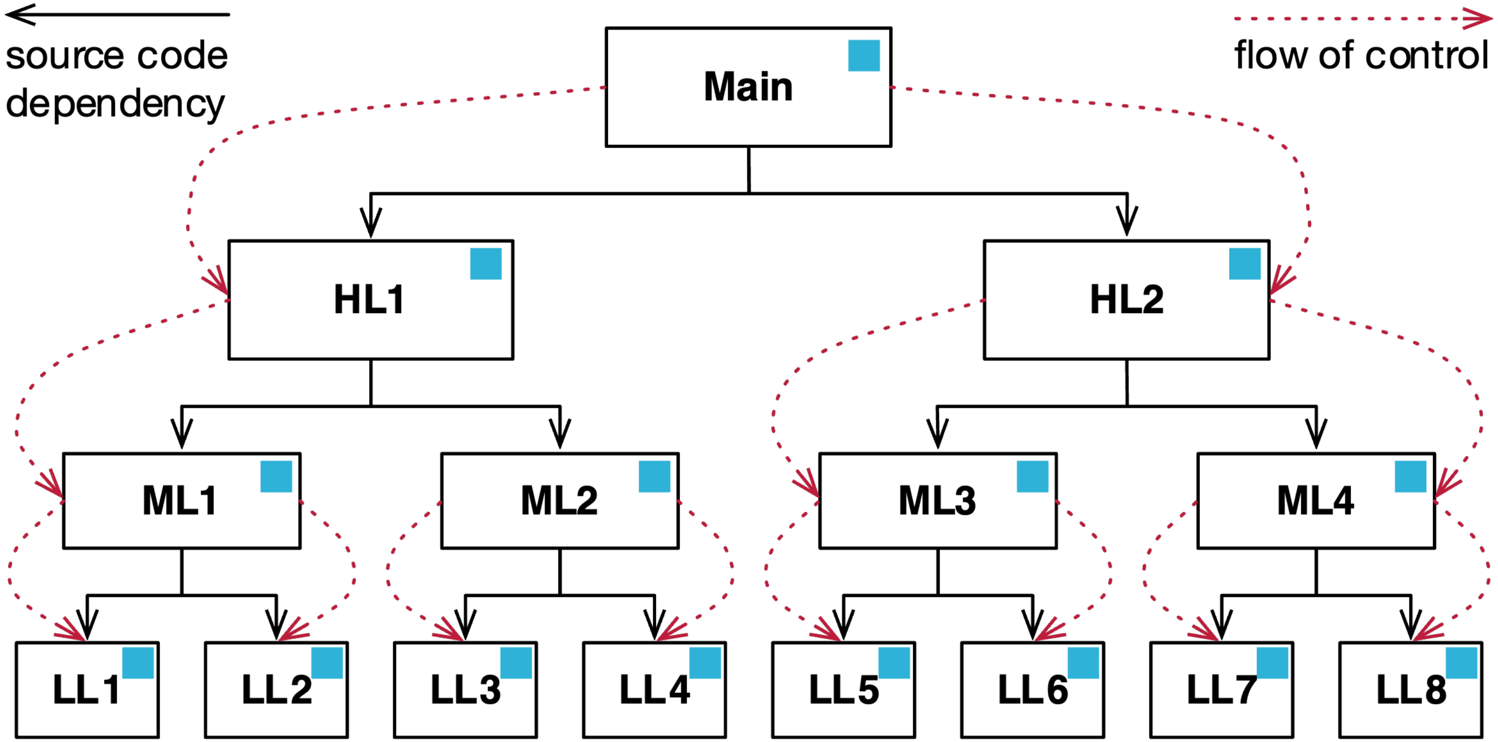
# Dependencies propagate change



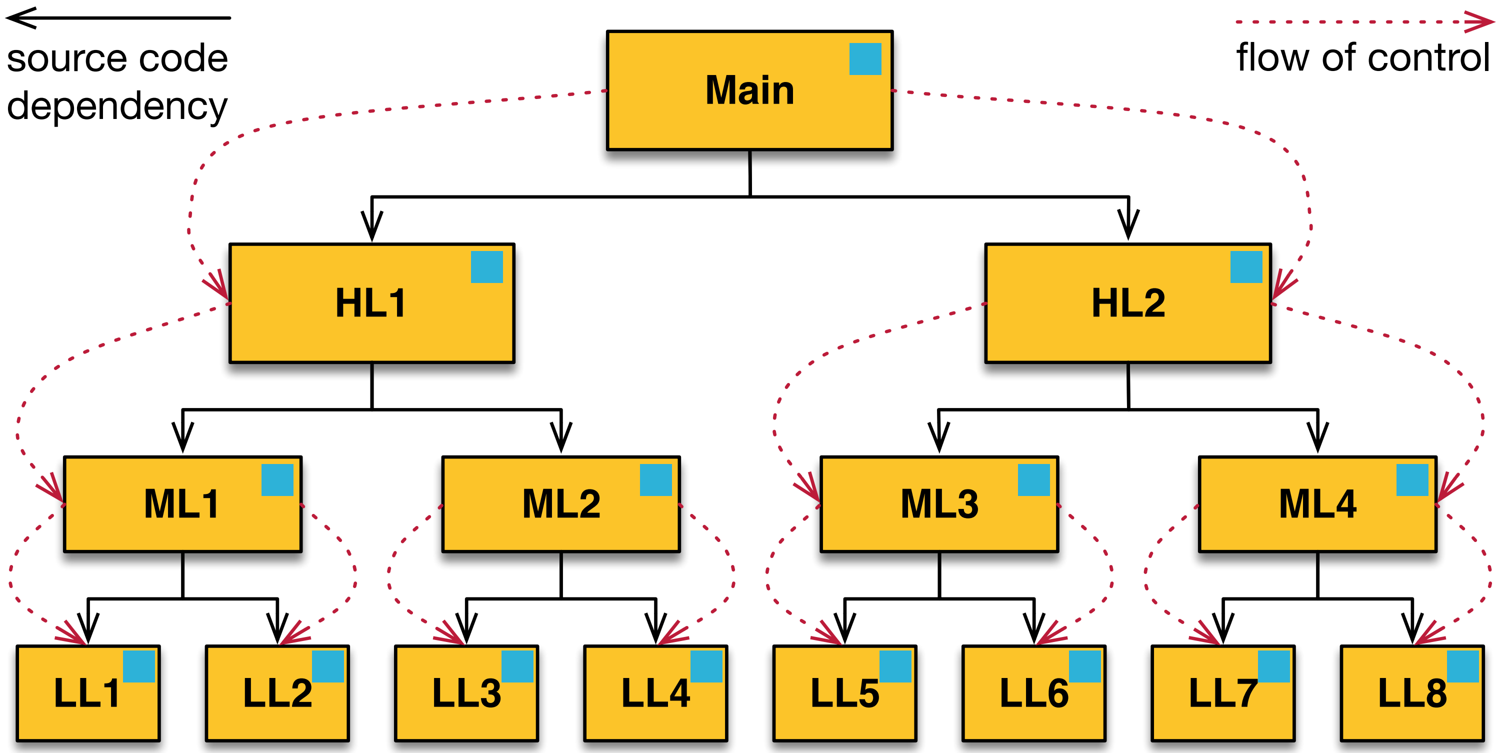
# Take Home Message

"Design for **change** by managing **dependencies**."

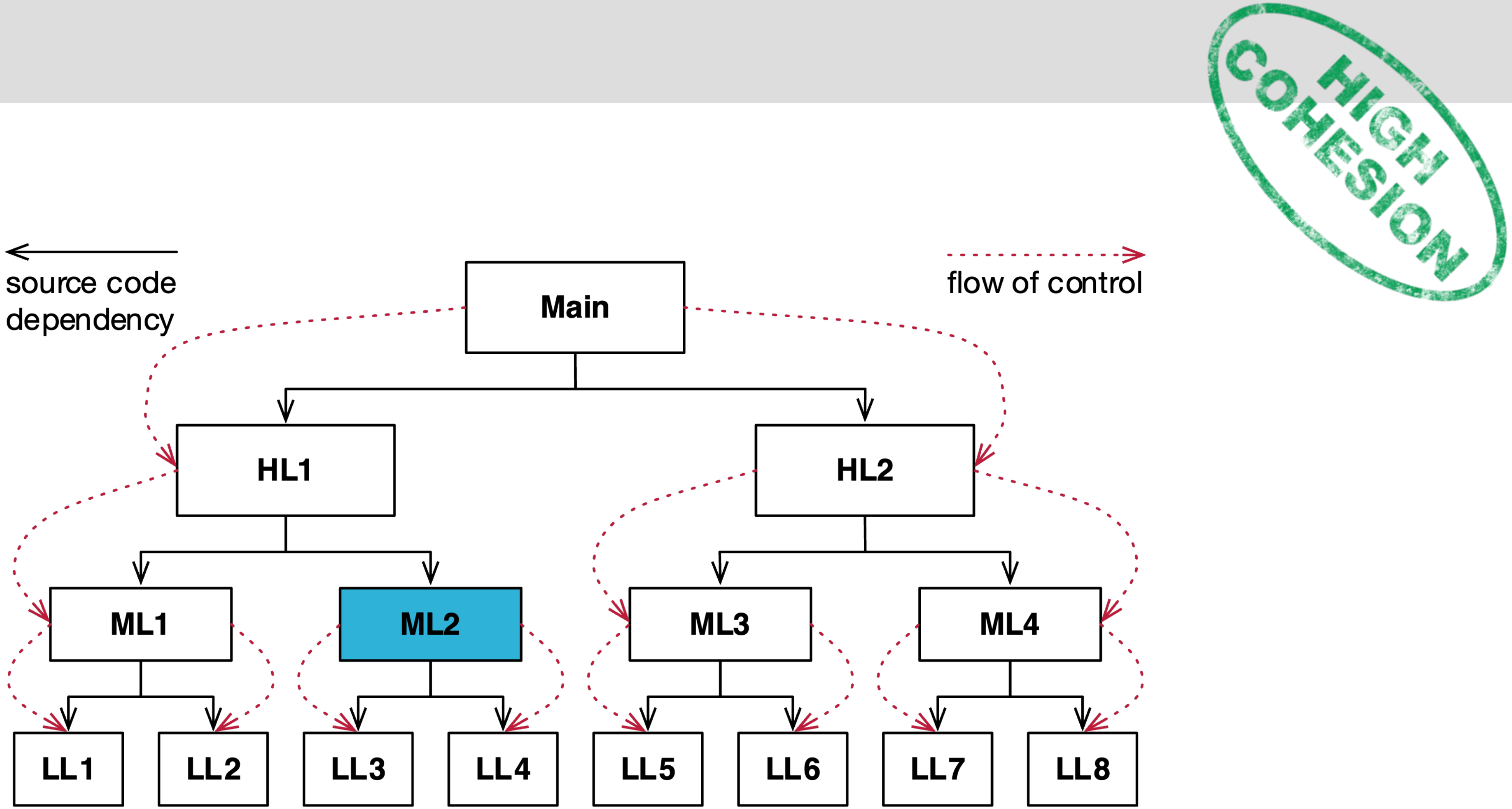
# Low cohesion



# Low cohesion




# High cohesion



# **Metrics** - Yourdon und Constantine, 1979

Use **Coupling** and **Cohesion** as Metrics to evaluate the quality of the design with regard to the impact and the reach of changes.

**Coupling** = Measure of Strength of the Relationship of two or more components.

A high Coupling causes systems, which are complicated to understand, to change or even to fix. Changes in a module often lead to a cascade of changes in other tightly coupled modules.

**Cohesion** = Degree of how well the elements of a module relate/fit together.

Random Cohesion = non-relating Abstractions are grouped together. [=> worst case]

High functional Cohesion = the sum of elements which form a consistent and clearly defined Behavior. Changes regarding a specific Behavior are easily performed, since they are ideally affecting a single element.

The Goal of the Design should therefore simultaneously aim for a

**preferably high cohesion and low coupling** of its components.

[E. Yourdon and L.L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, 1979]

# Software can not be proven to be right

“Testing shows the presence, not the absence of bugs.”

Edsger W. Dijkstra[1]

Software is not a mathematical endeavour in the sense that software can not formally be proven to be correct. In that way at has a lot in common with scientific theories and laws which can only be proven to be incorrect.

[[1]Software Engineering Techniques: Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27--31 October 1969]

## Symptoms of poor design

**Rigidity** – The system is difficult to change

**Fragility** – The system is easy to break

**Immobility** – The system is difficult to reuse

**Viscosity** – It is difficult to do the right thing

**Needless Complexity** – Overdesign (YAGNI - "You Aren't Gonna Need It")

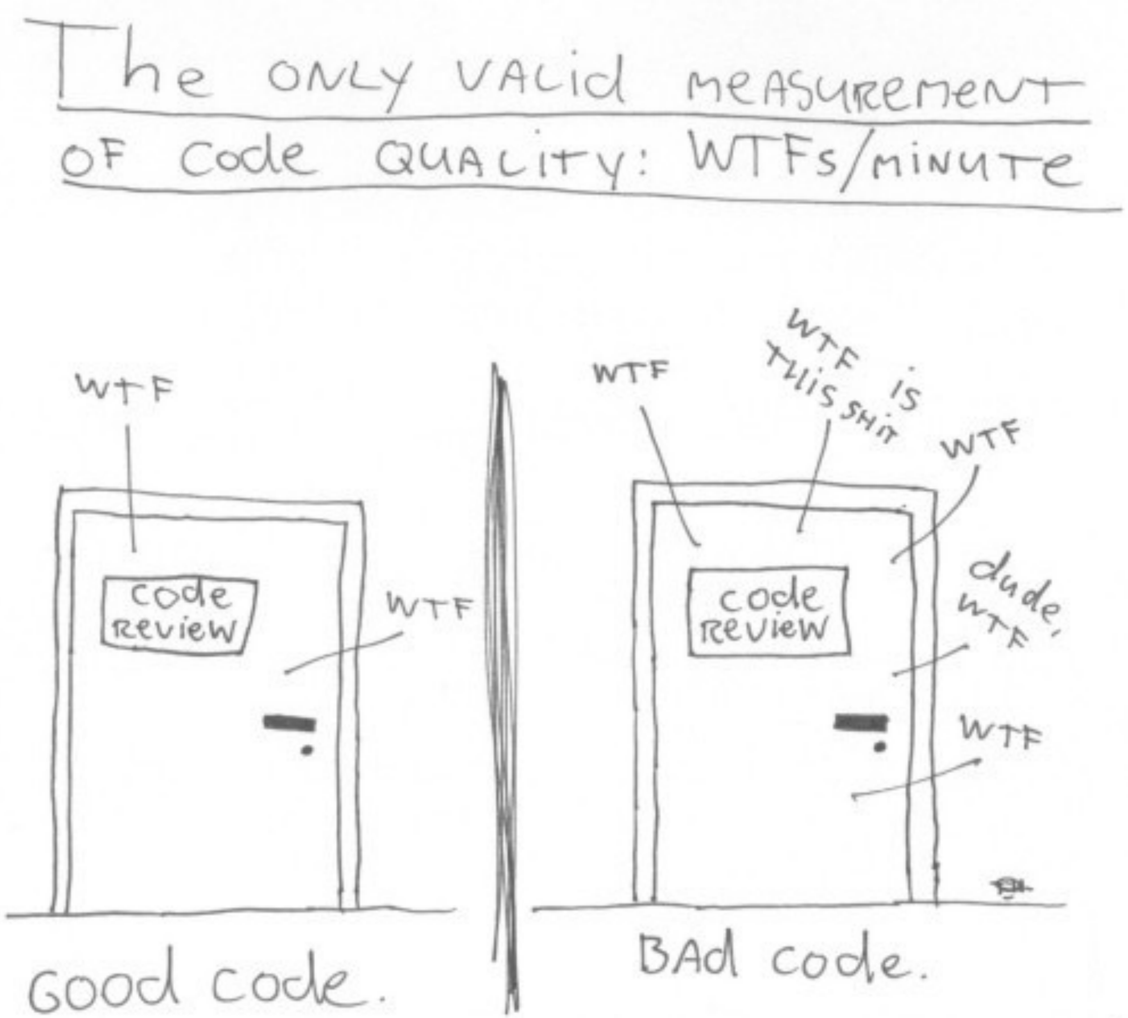
**Needless Repetition** – Copy / Paste Development

**Opacity** – The systems design is hard to understand

These symptoms are similar in nature to code smells, but are at a higher level. They are smells that pervade the overall structure of the software rather than a small section of code.

[Robert C. Martin, Agile Principles, Patterns, and Practices, 2003]

# Another metric for code quality




https://bit.ly/2VWkvBA

# » No Silver Bullet «

Software Development is a **non-deterministic** process – McConnell, 2004

Clearly, there is no magic, **no “silver bullet”** (F. P. Brooks, 1987) that can unfailingly lead the software engineer down the path from requirements to the implementation of a complex software system, which reflects a high-quality design on the other hand.

Software Design and Software Development are **evolutionary processes** which demand for an **incremental and iterative approach**.

There are “no” norms nor standards, which assure the quality of development.

Software Design is a **heuristic process** – Quality is mainly achieved by cautiously applying proven **Principles, Patterns and Practices**.

[Steve McConnell, Code Complete, Second Edition, 2004]

[Frederick P. Brooks, No Silver Bullet: Essence and Accidents of Software Engineering, 1987]

# Complexity

**“There is an inherent complexity in software systems that works against the software quality.”**

– Dijkstra, 1972

**Causes of Complexity** (Booch et al., 2007)**:**

- complexity of the problem domain
- difficulty of managing the development process
- flexibility possible through software
- problems of characterizing the behavior of discrete systems

Problems evolving in scope, result in continuously growing complexity of software systems. Complexity does not increase linearly. The more complex a system gets, the higher its possibility of failure gets. The task of designing high-quality software consists in managing its complexity.

[Edsger W. Dijkstra, The humble programmer, 1972]

[Grady Booch et al., Object-oriented analysis and design with applications, 2007]

# Basic Concepts of Organized Complexity

**"…the maximum number of chunks of information that an individual can handle simultaneously is on the order of seven, plus or minus two."**

– G. A. Miller, 1956

**Decomposition:** "divide et impera" (divide and conquer) – When designing a complex software system, it is essential to decompose it into smaller and smaller parts, each of which we may then refine independently.

**Abstraction:** "We (humans) have developed an exceptionally powerful technique for dealing with complexity. We abstract from it. Unable to master the entirety of a complex object, we choose to *ignore its inessential details*, dealing instead with the generalized, idealized model of the object" (W. Wulf ,1980)

**Hierarchy:** Another way to increase the semantic content of individual chunks of information is by explicitly recognizing the class and object hierarchies within a complex software system. By classifying objects into groups of related abstractions (e.g., kinds of plant cells versus animal cells), we come to explicitly distinguish the common and distinct properties of different objects, which further helps us to master their inherent complexity (A. Goldberg, 1984).

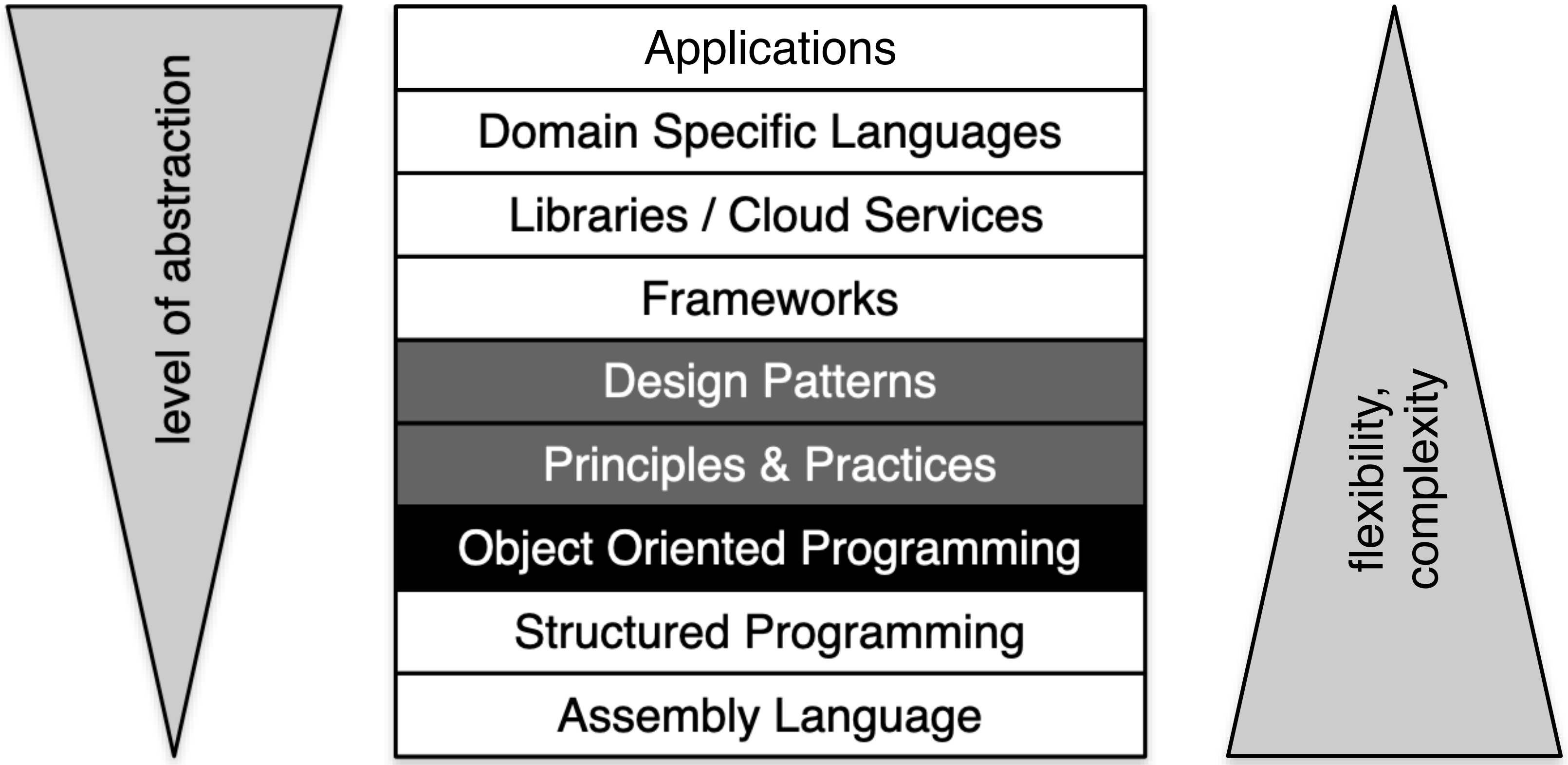
[George A. Miller, The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information, 1956]

# Object-orientation vs. complexity

Object-orientation (OO) is an approved paradigm to **organize the complexity of software**.

The methods of object-oriented analysis (OOA) and design (OOD) provide an engineering approach, which leads from the specific requirements to its software implementation.

The general procedure is based on principles of **modularization (decomposition), encapsulation, abstraction** and **hierarchy**.


level of abstraction
Applications
Domain Specific Languages
Libraries / Cloud Services
Frameworks
Design Patterns
Principles & Practices
Object Oriented Programming
Structured Programming
Assembly Language
flexibility,
complexity

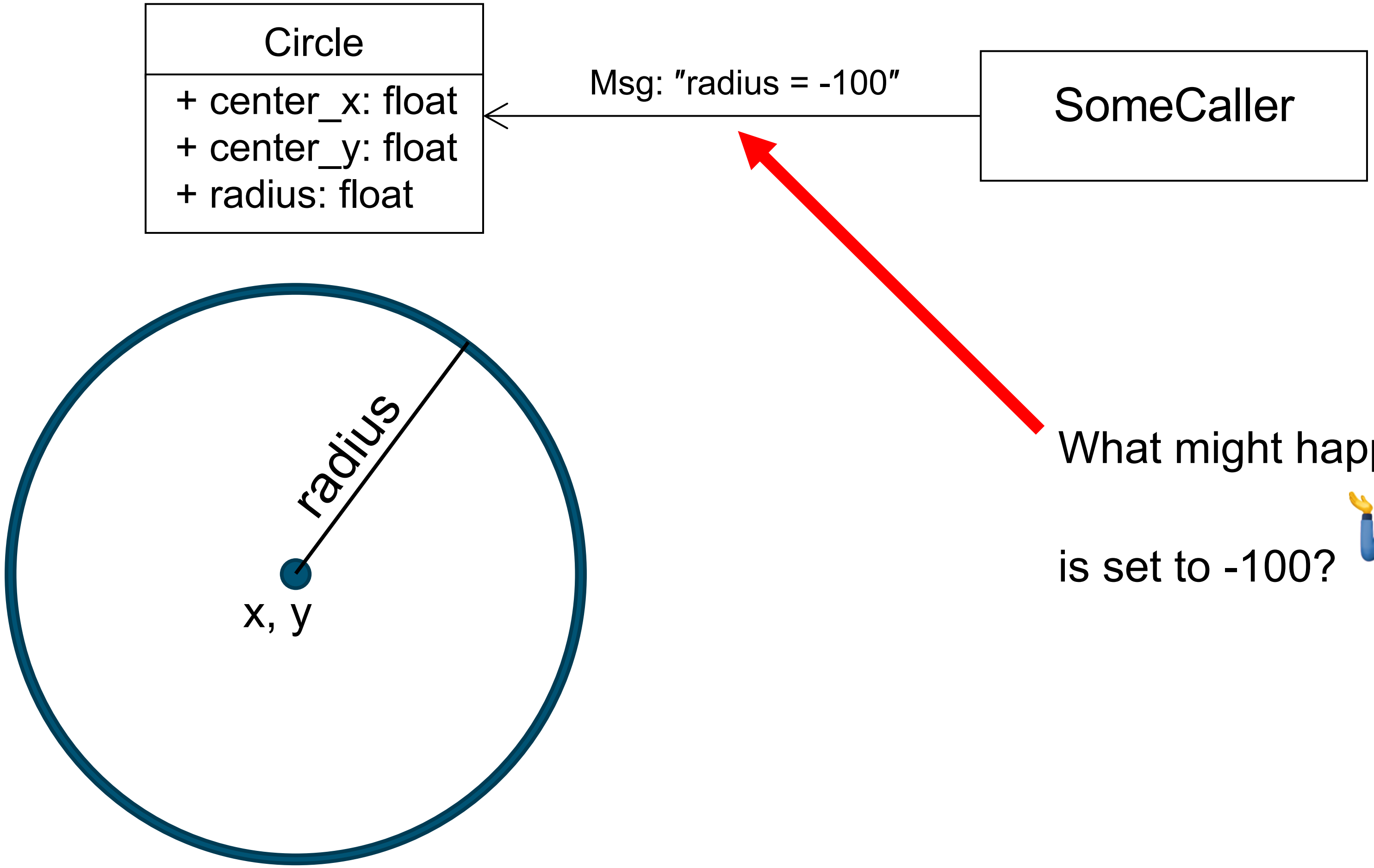
# Managing complexity with Object Oriented Programming

In the 1980s Alan Kay introduced the term „Object Oriented Programming“.

An OO Language should at least support the following three key principles:

- **Encapsulation**
- **Inheritance**
- **Polymorphism**

# Object Oriented Programming - Encapsulation




What might happen when the radius is set to -100? 🤷‍♂️

:(

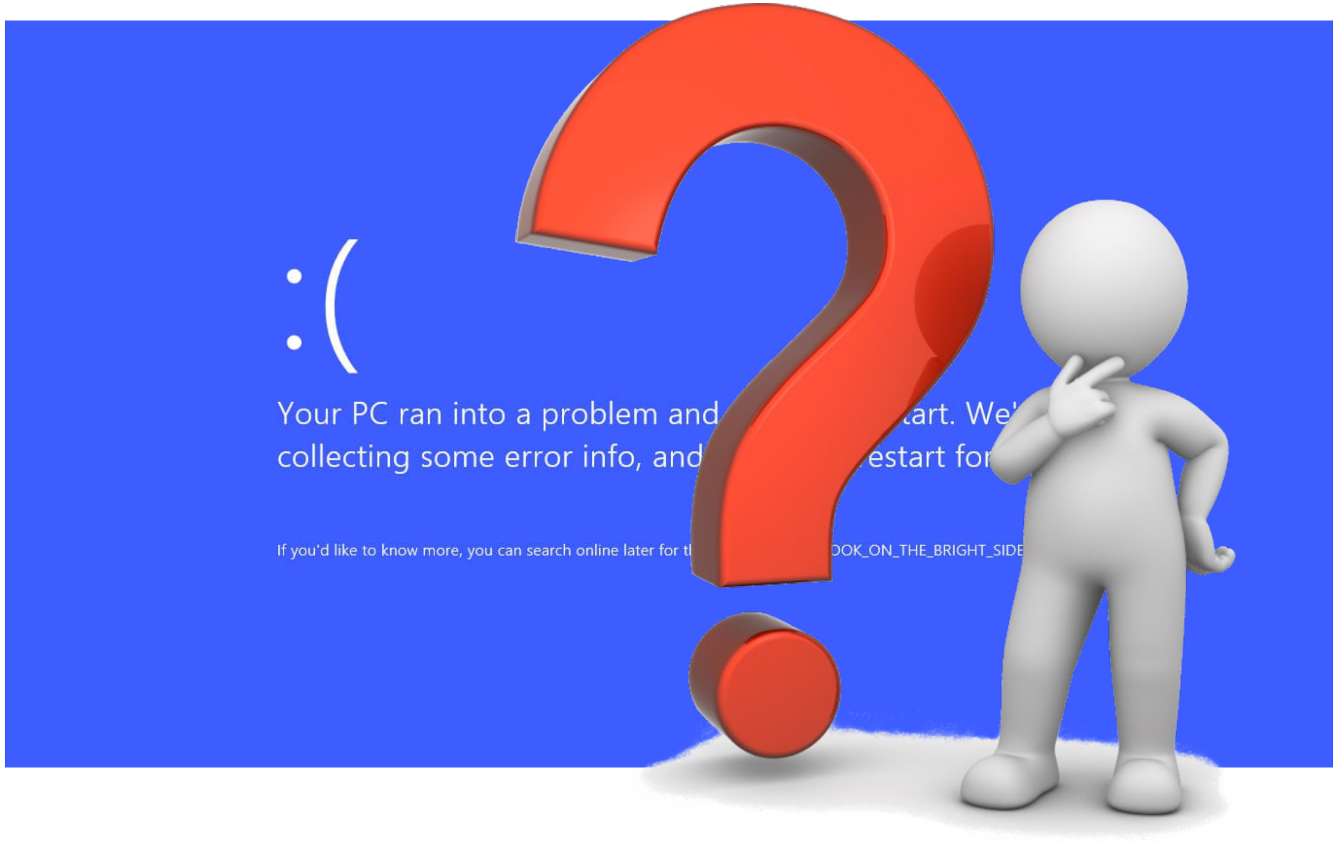
Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

If you'd like to know more, you can search online later for this error: ALWAYS_LOOK_ON_THE_BRIGHT_SIDE_OF_LIFE

# Who‘s to blame?

a) The circle
b) The caller
c) Both
d) None of them

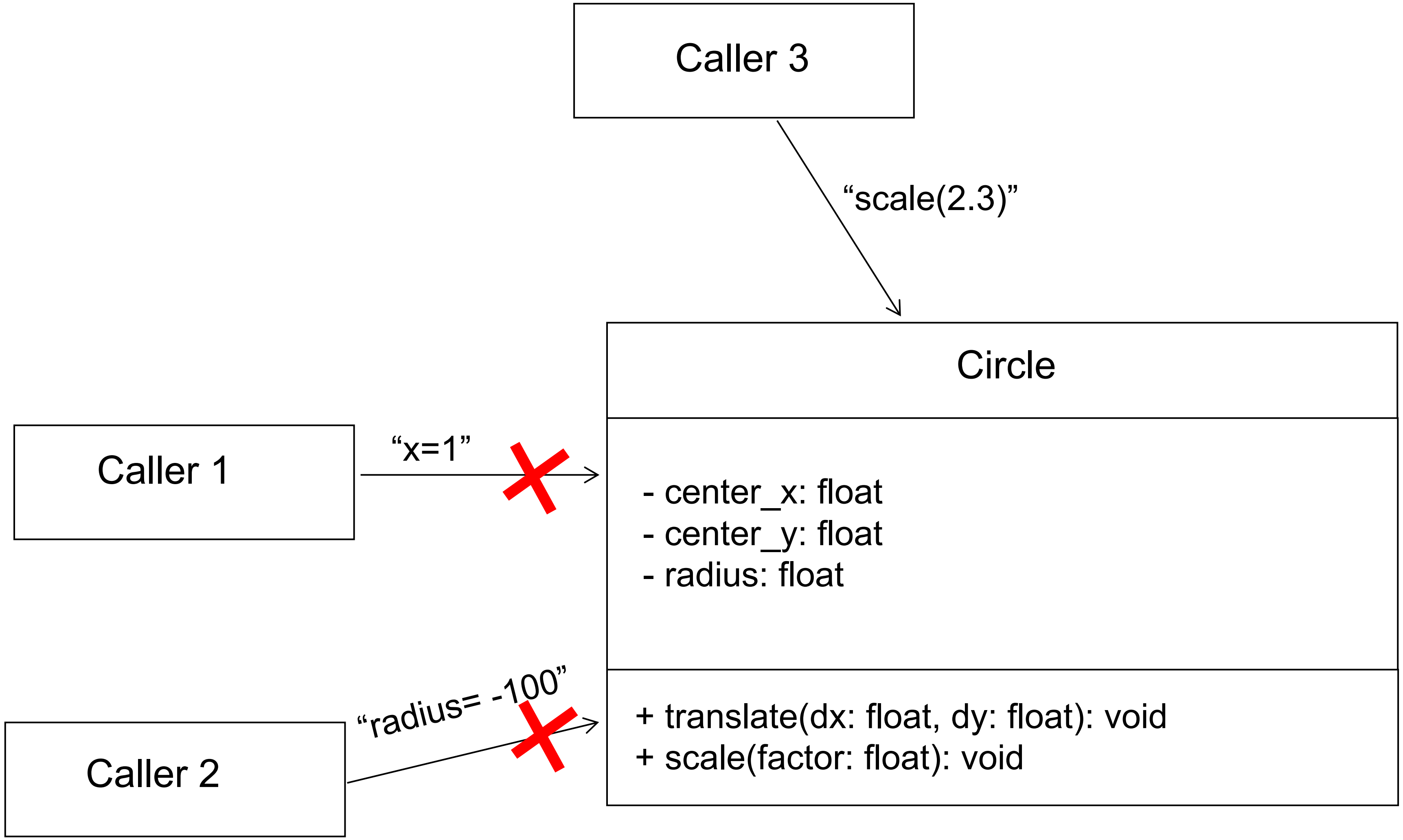
Join at
slido.com
#123 444



# Keep in mind!

An object is responsible to guarantee it’s state is valid!

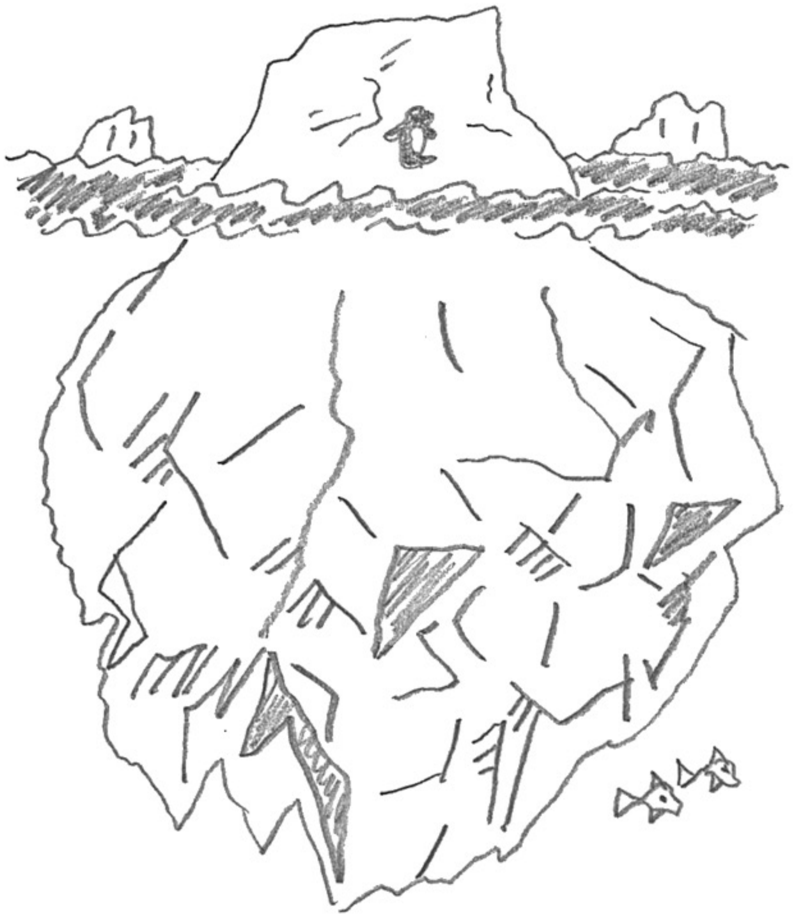
# Object Oriented Programming - Encapsulation



# Information Hiding

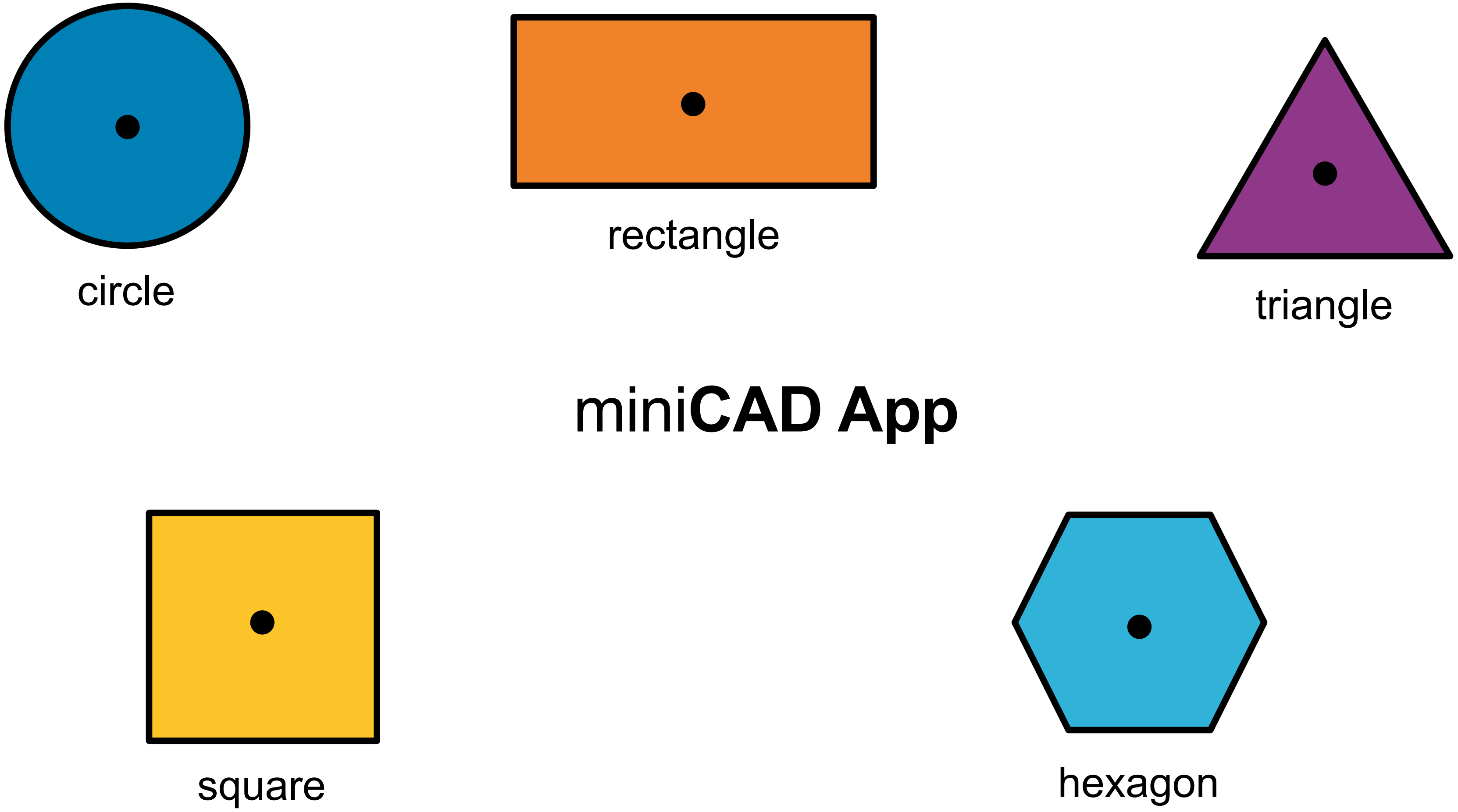
Public API

Details / Implementation

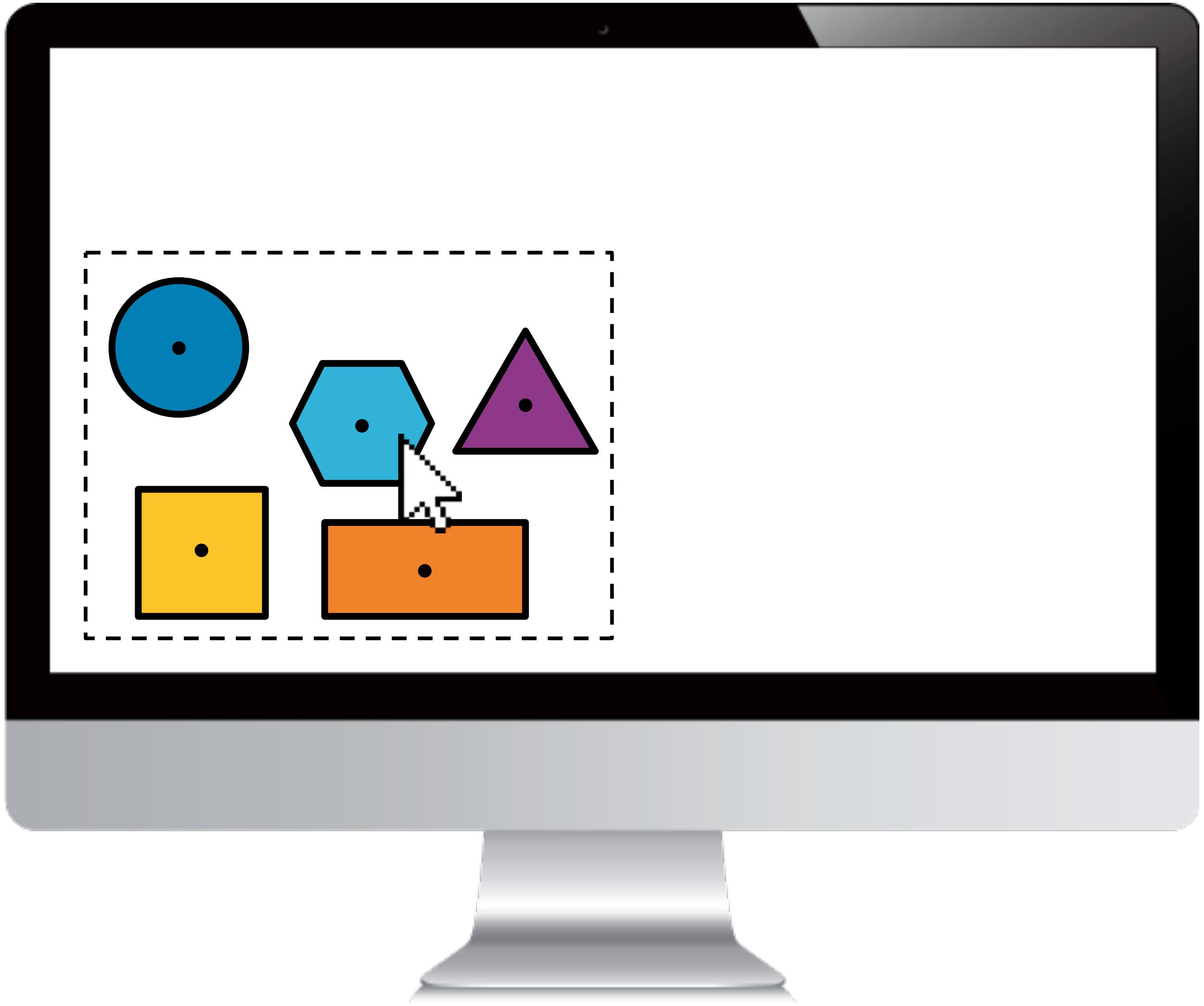


[Steve McConnell, Code Complete, Second Edition, Microsoft Press, 2004]

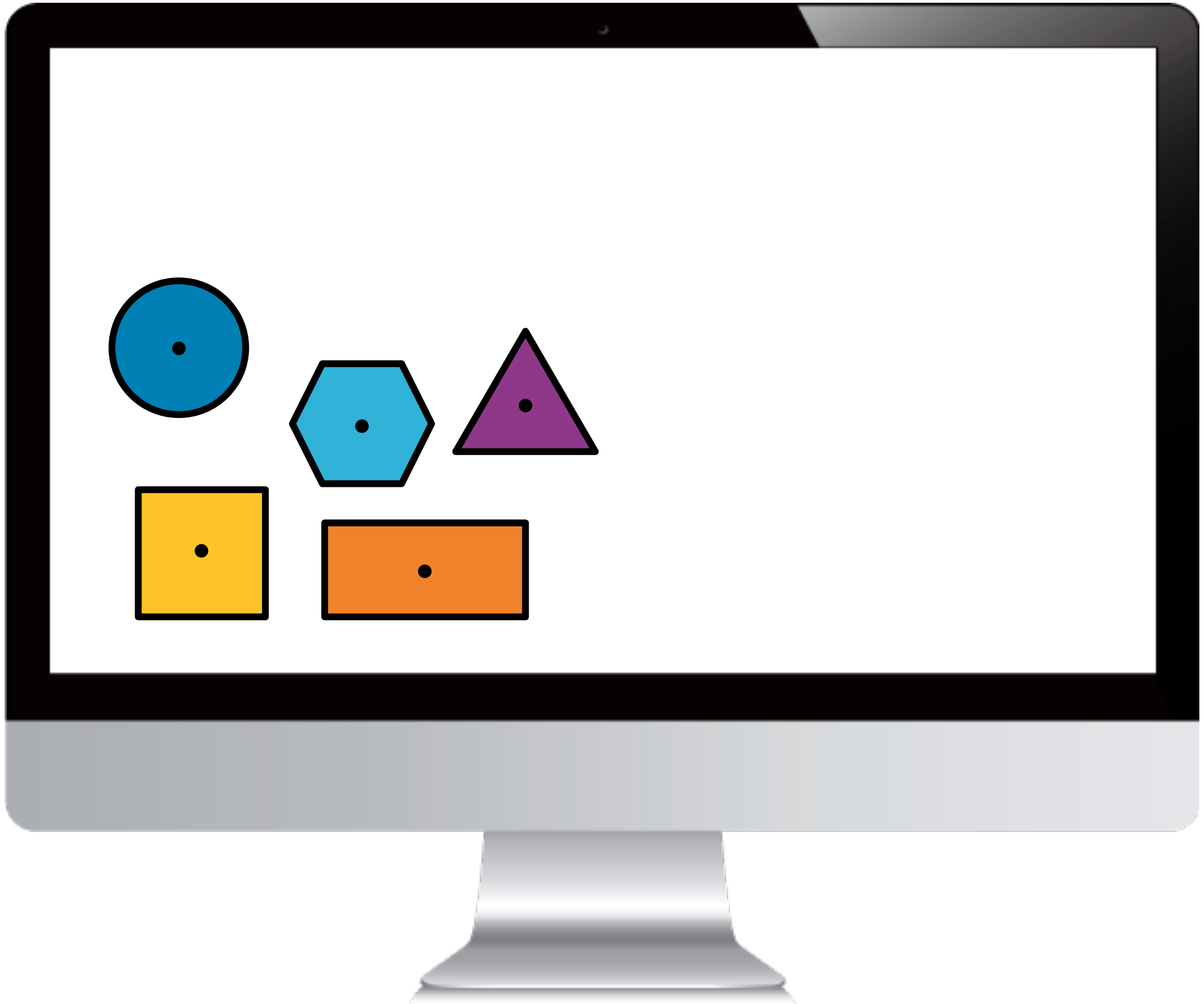
# Object Oriented Programming - Inheritance



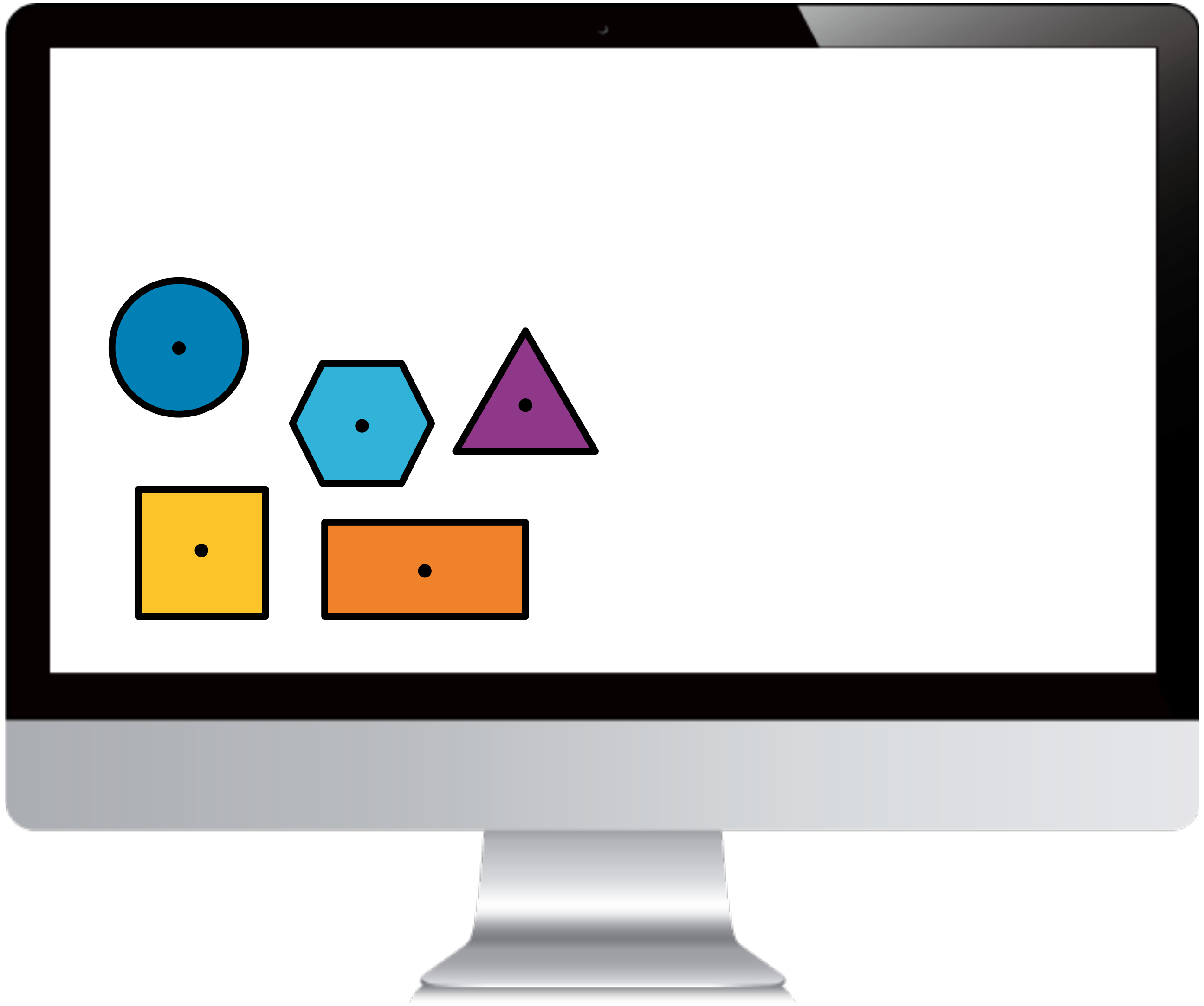

# Move objects on the screen



# What actually happens …



# After you fixed it …



# Object Oriented Programming - Inheritance

Classes can be specializations of other classes. That means classes can be in hierarchical order by inheriting properties and behavior of higher ranked classes.

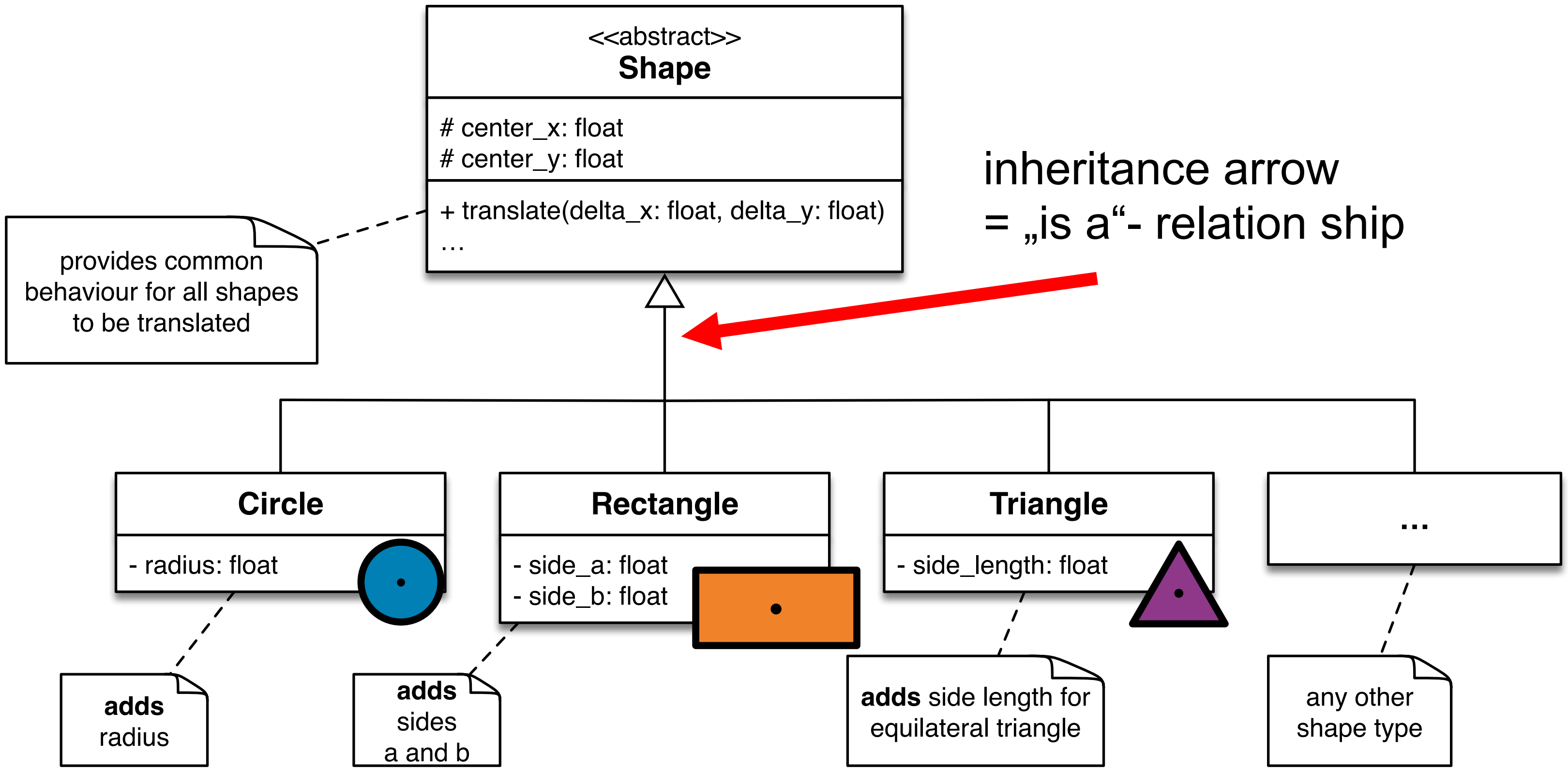
Lower ranked classes can specialize (override) or extend higher classes.

On code level Inheritance may be used to **avoid redundancy** by allowing code reuse.

Supports
**Don’t Repeat Yourself (DRY)**
principle

[A. Hunt and D. Thomas, The Pragmatic Programmer, 1999]

**Inheritance enables Polymorphism!**



# Object Oriented Programming - Inheritance

Classes can be specializations of other classes. That means classes can be in hierarchical order by inheriting properties and behavior of higher ranked classes.

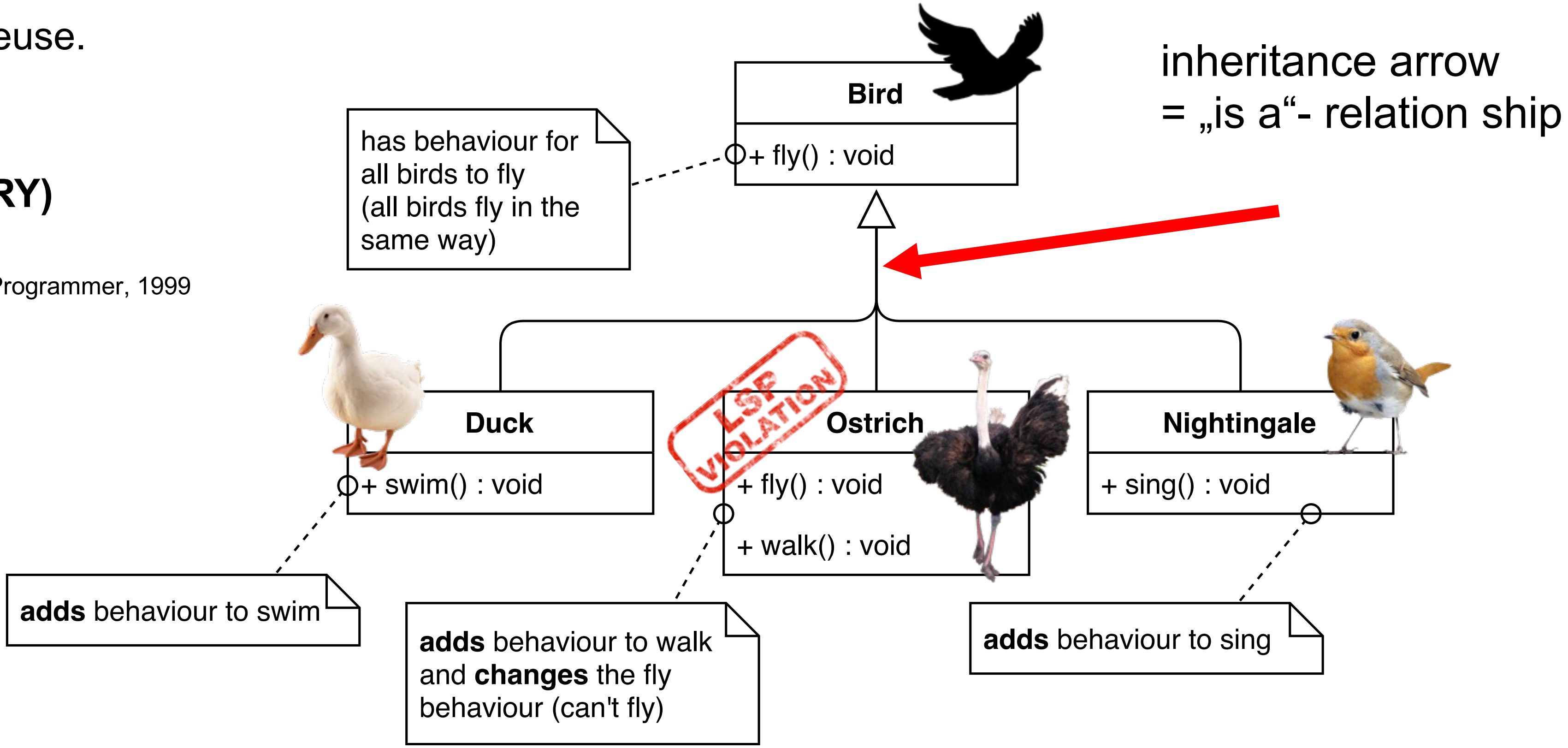
Lower ranked classes can specialize (override) or extend higher classes.

On code level Inheritance may be used to **avoid redundancy** by allowing code reuse.

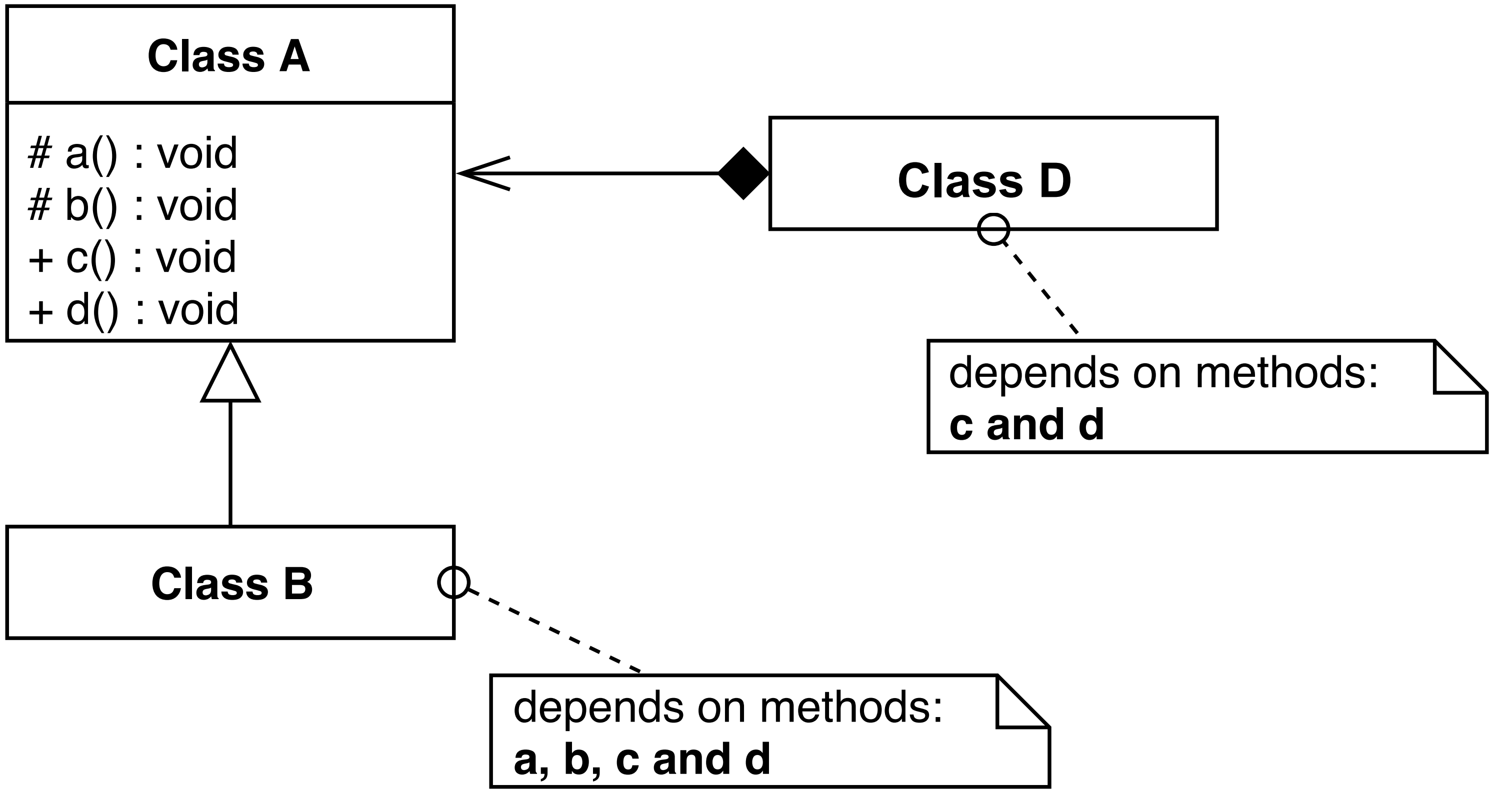
Supports
**don't repeat yourself (DRY)**
principle

[A. Hunt and D. Thomas, The Pragmatic Programmer, 1999

**Inheritance enables Polymorphism!**



# Favor composition over inheritance (FCoI)




[Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, 1995]

# Object Oriented Programming - Polymorphism

- Generally, the ability to have different individuals of a species.
  (…also in Biology, Chemistry)
- In object-oriented programming, polymorphism refers to a programming language's ability of objects to react differently to one and the same message depending on their class.
  (Technically, this is achieved by redefining methods in derived classes - Inheritance)
- One may also speak of the autonomy or independence of objects.

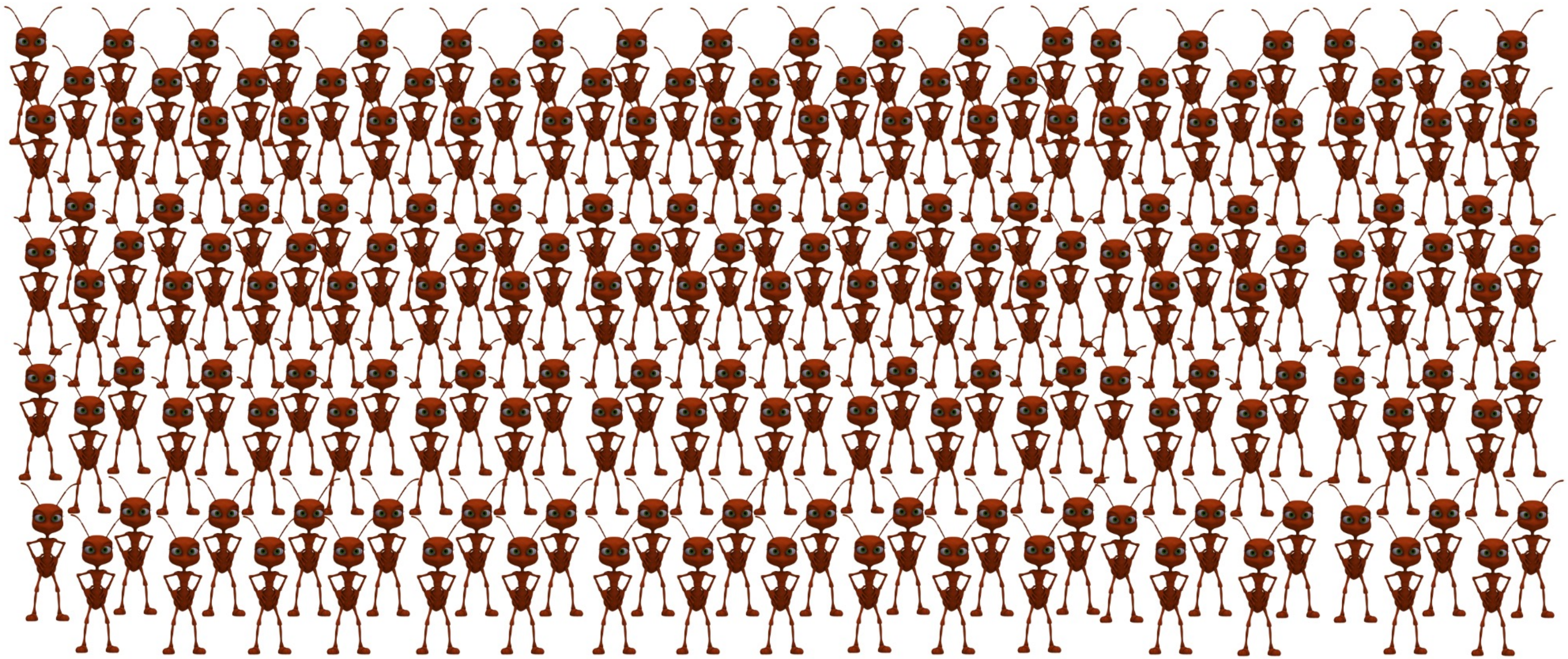
# Have you heard about polymorphism?

a) Poly … what??
b) I have heard of it, but I don't use it.
c) I use it, but I can't explain it.
d) I use it all the time. It's part of my daily work.
e) I'm a polymorphism ninja
f) other

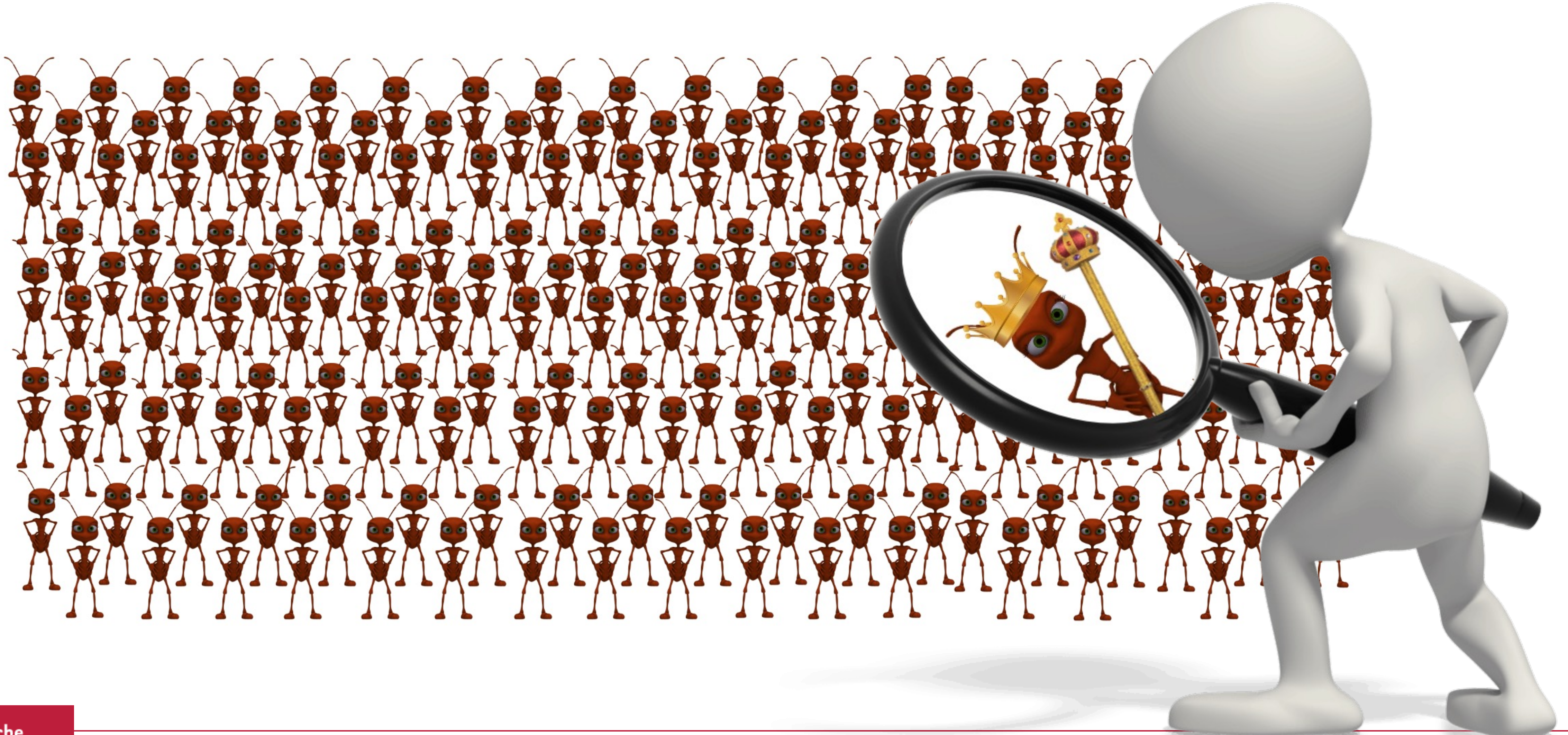
Join at
slido.com
#123 444

# Polymorphism - Ant Hill

# Polymorphism – All are ants

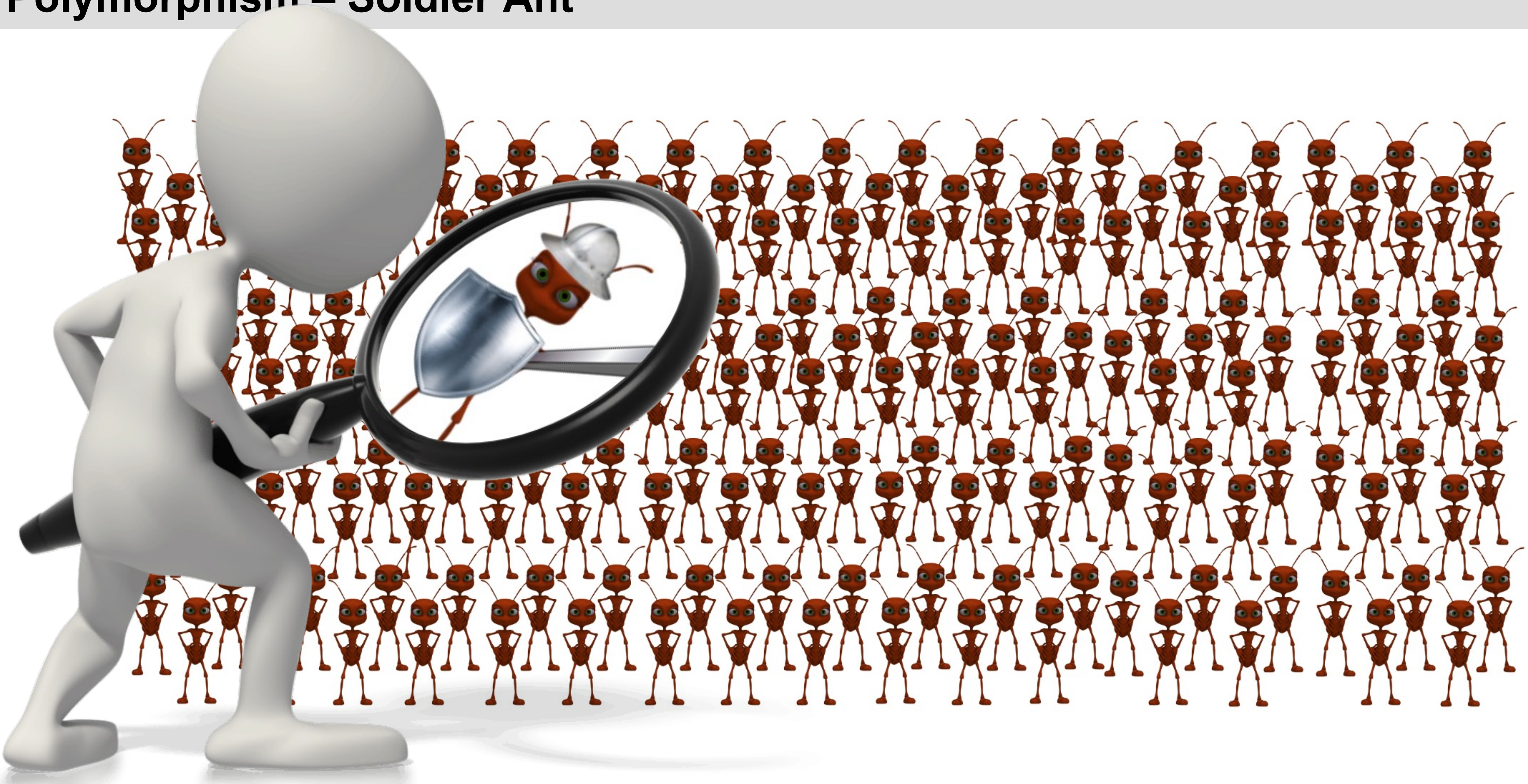


# Polymorphism – Queen is managing the tribe (Core Unit)



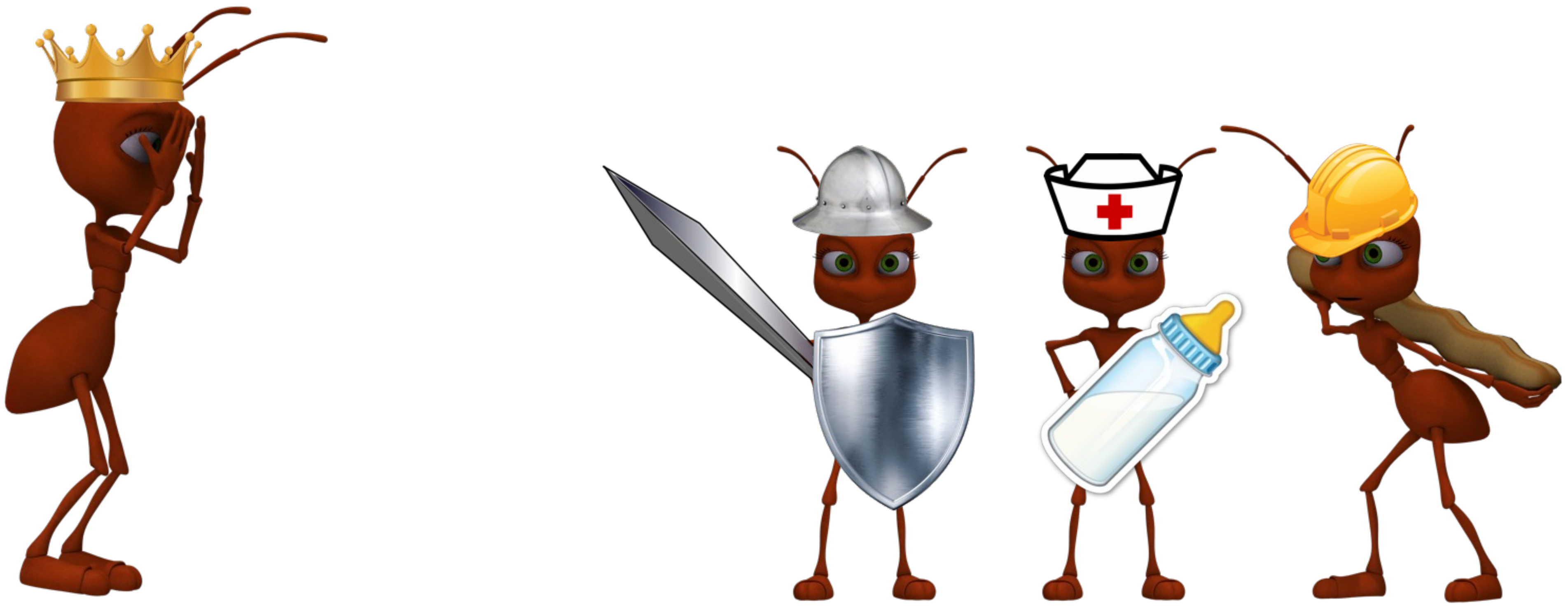
# Polymorphism - Worker Ant

# Polymorphism – Soldier Ant

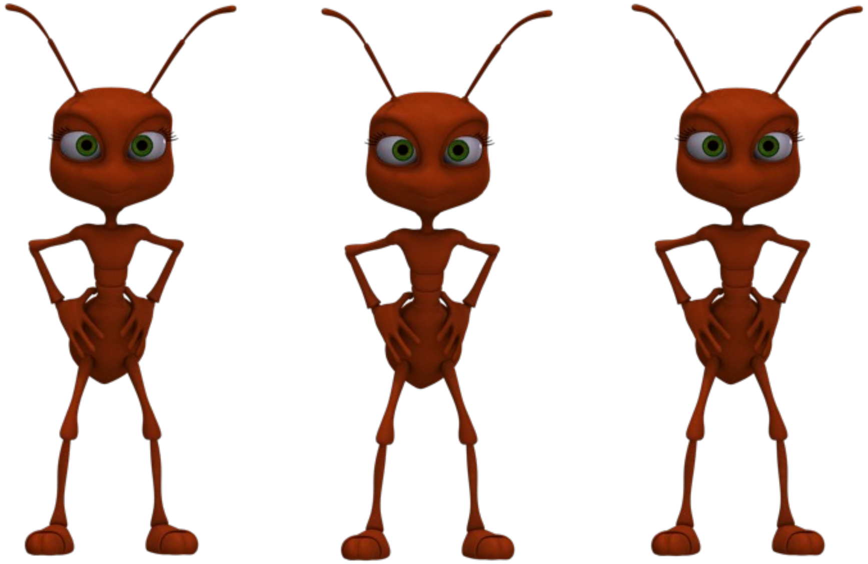


# Polymorphism – Nurse Ant

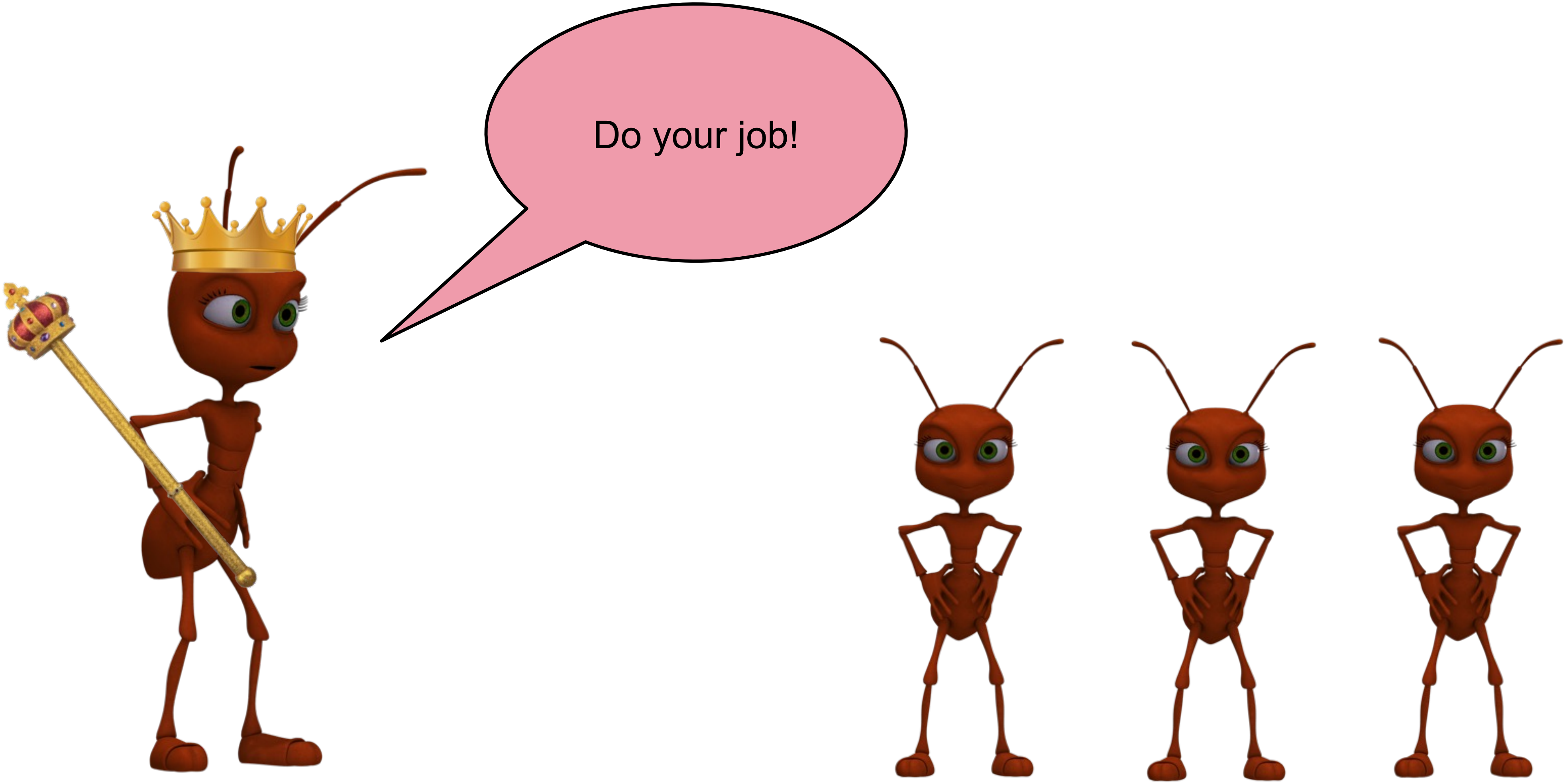
# Polymorphism – Queen doesn‘t know specific ants



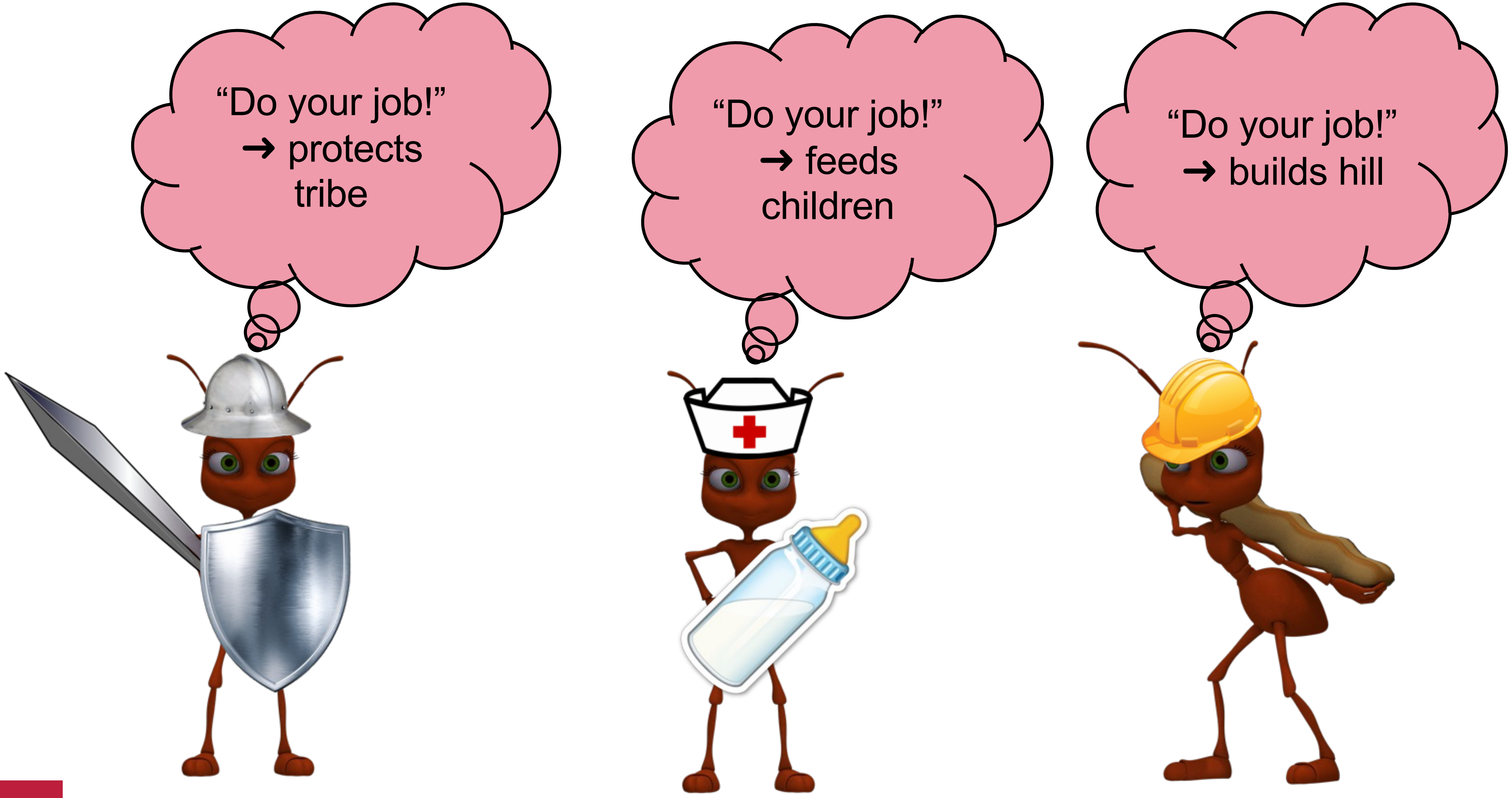
# Polymorphism – Queen only knows ants



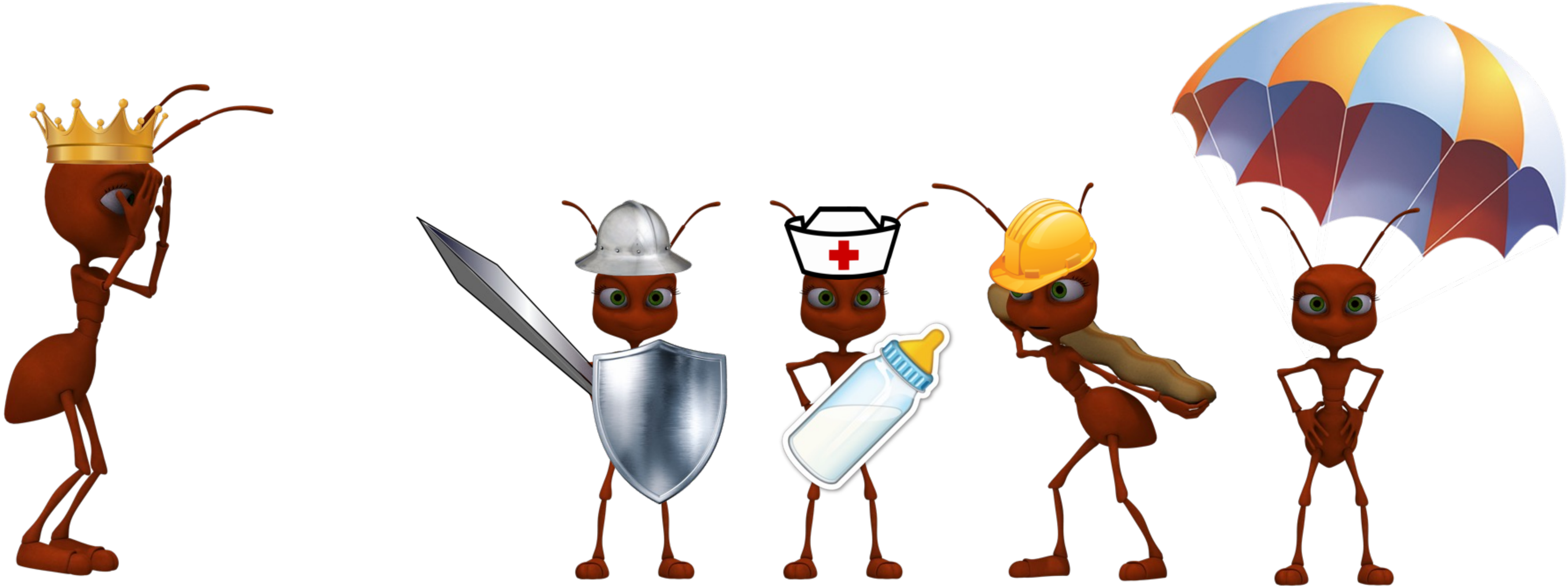
# Queen knows that ants understand the message: „DO YOUR JOB!“



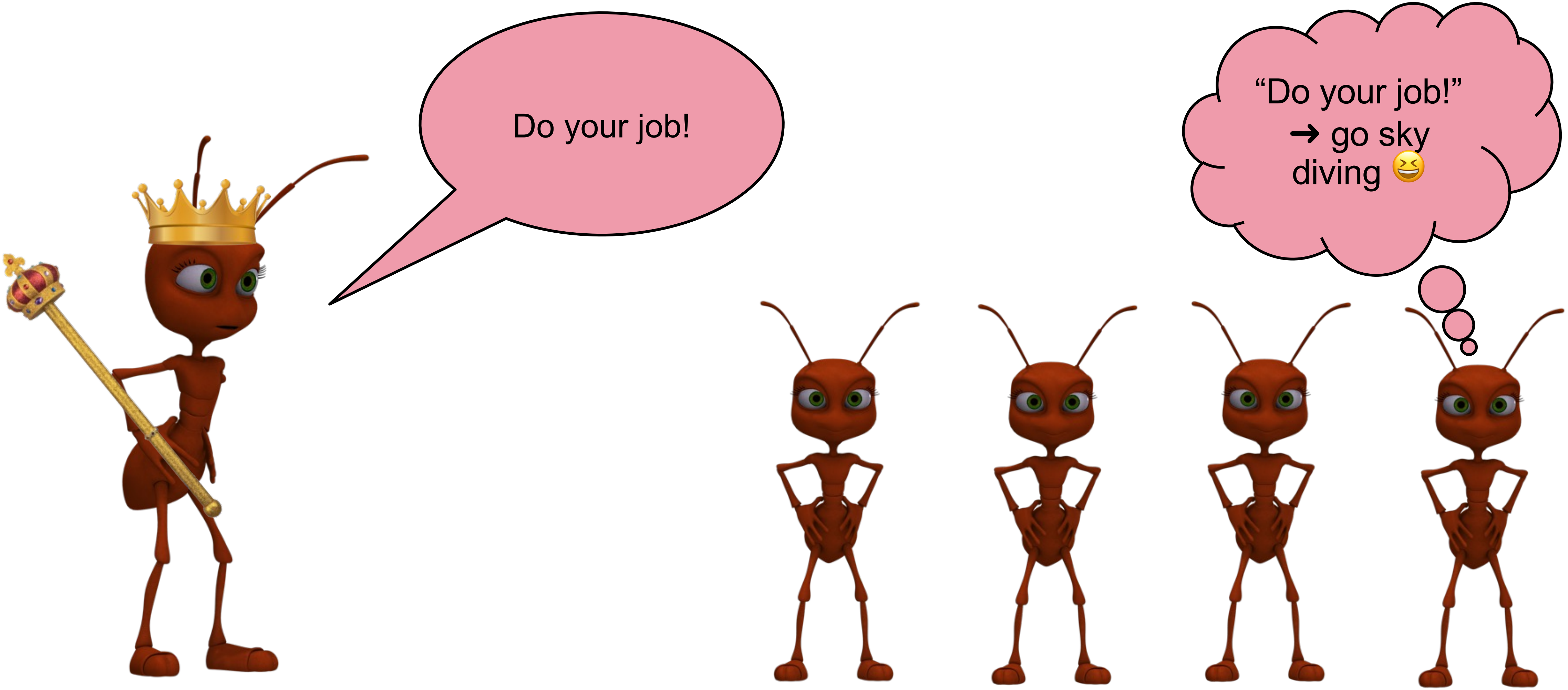
# All ants react differently to the message according to their subtype



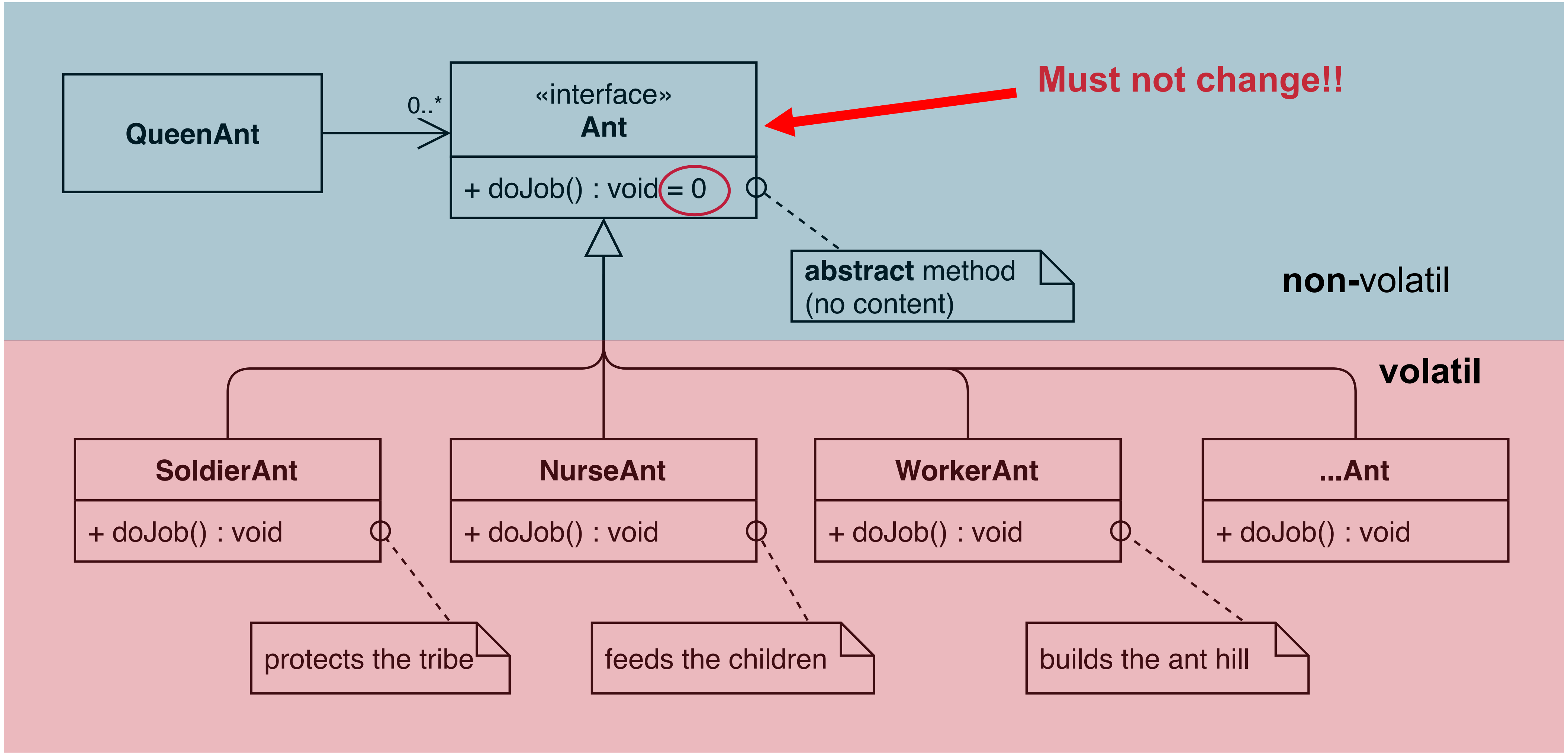
# Polymorphism – Ready for Evolution - SkyDriver Ant



# Polymorphism – Ready for Evolution - Skydiver Ant



# Polymorphism – Queen Ant is not affected by change



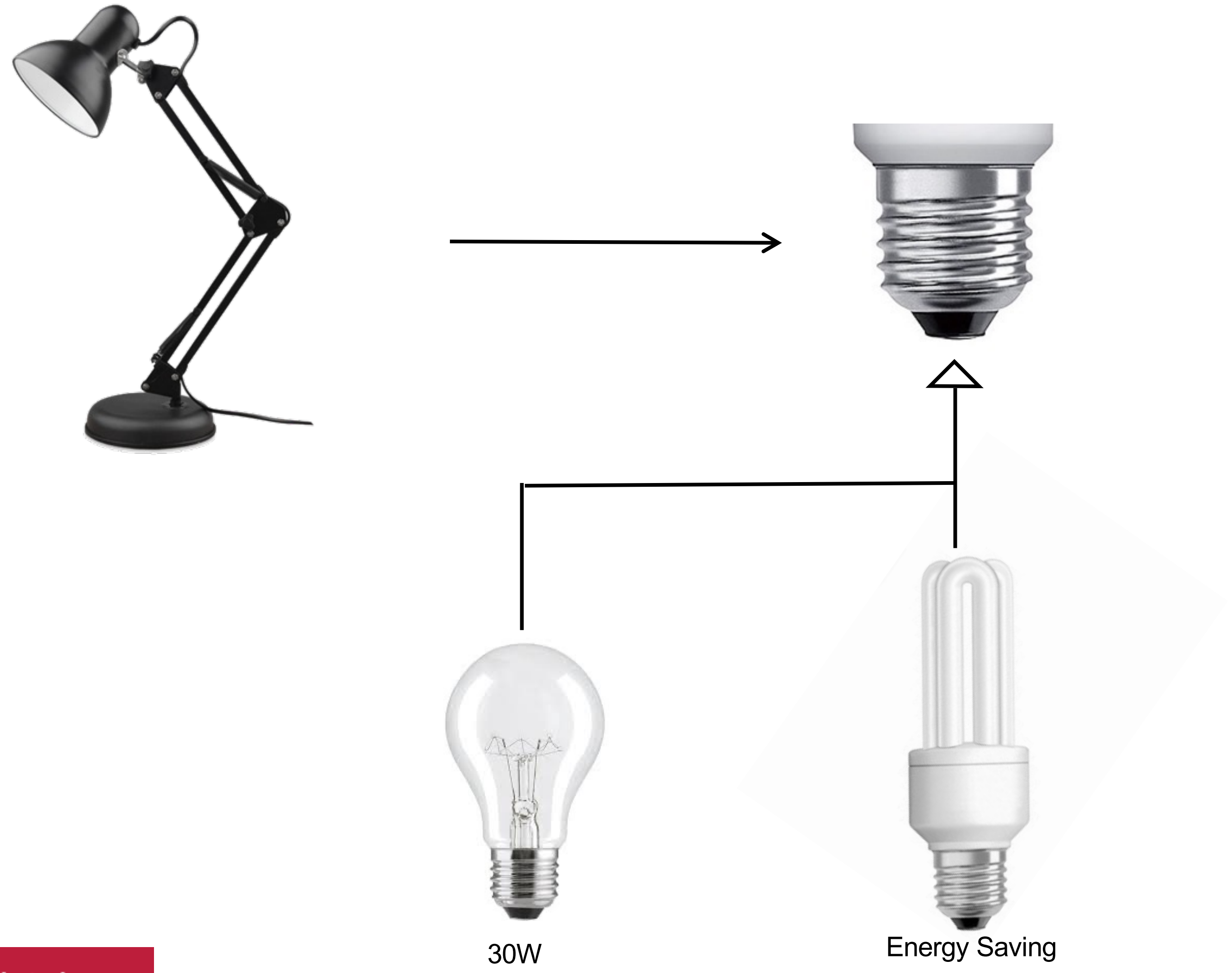
# Object Oriented Programming - Polymorphism

# Object Oriented Programming - Polymorphism

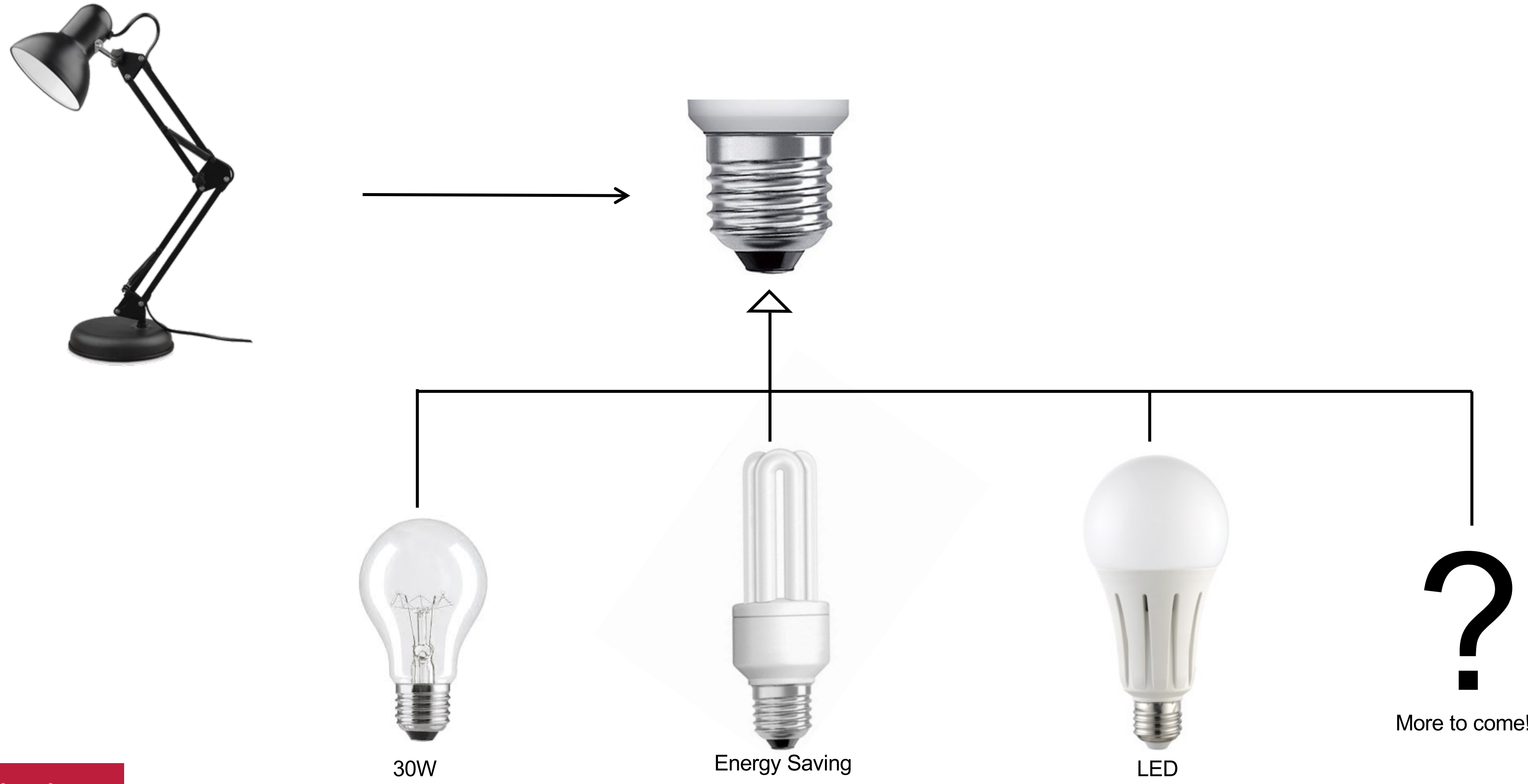
# Object Oriented Programming - Polymorphism

# Object Oriented Programming - Polymorphism

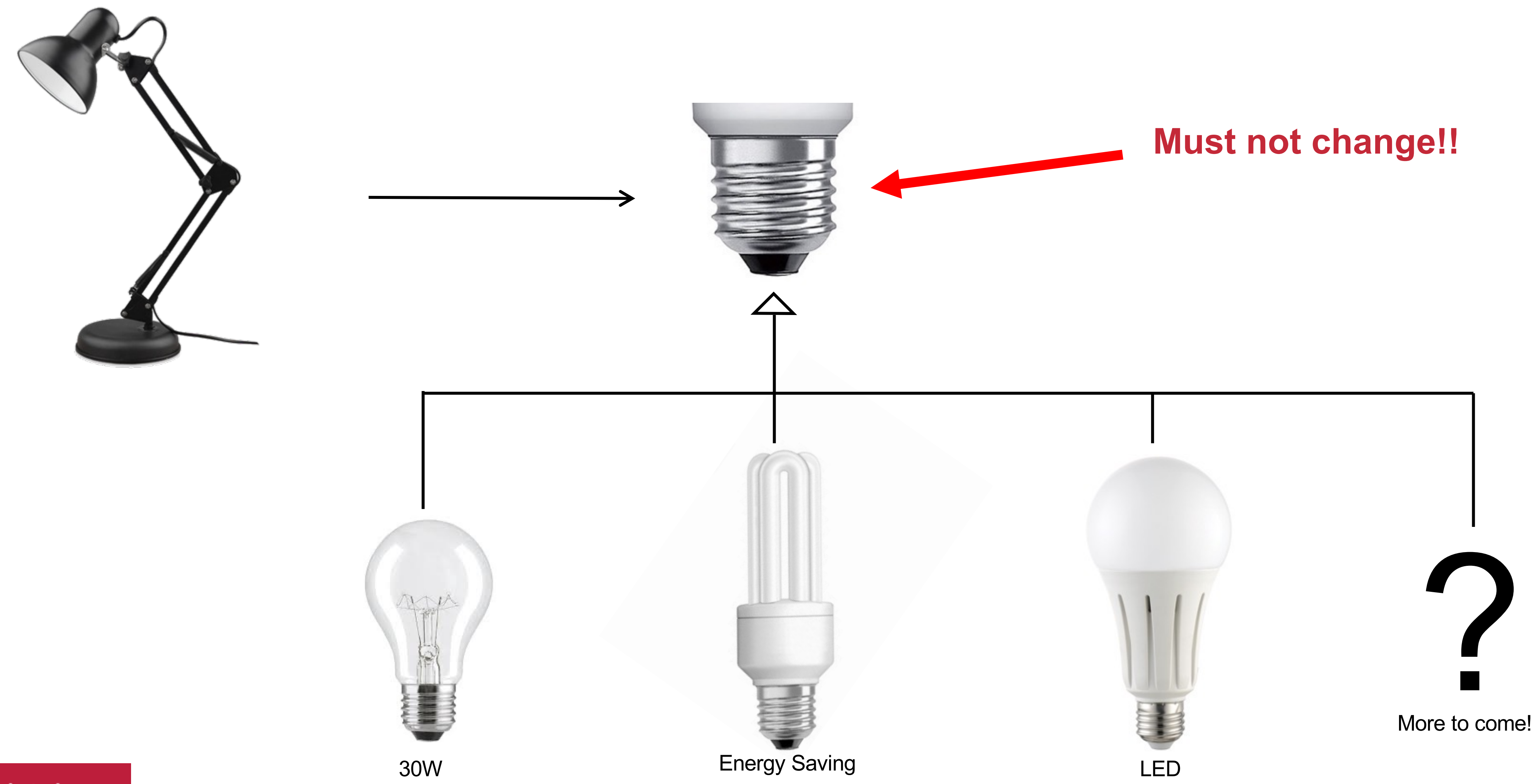
# Object Oriented Programming - Polymorphism



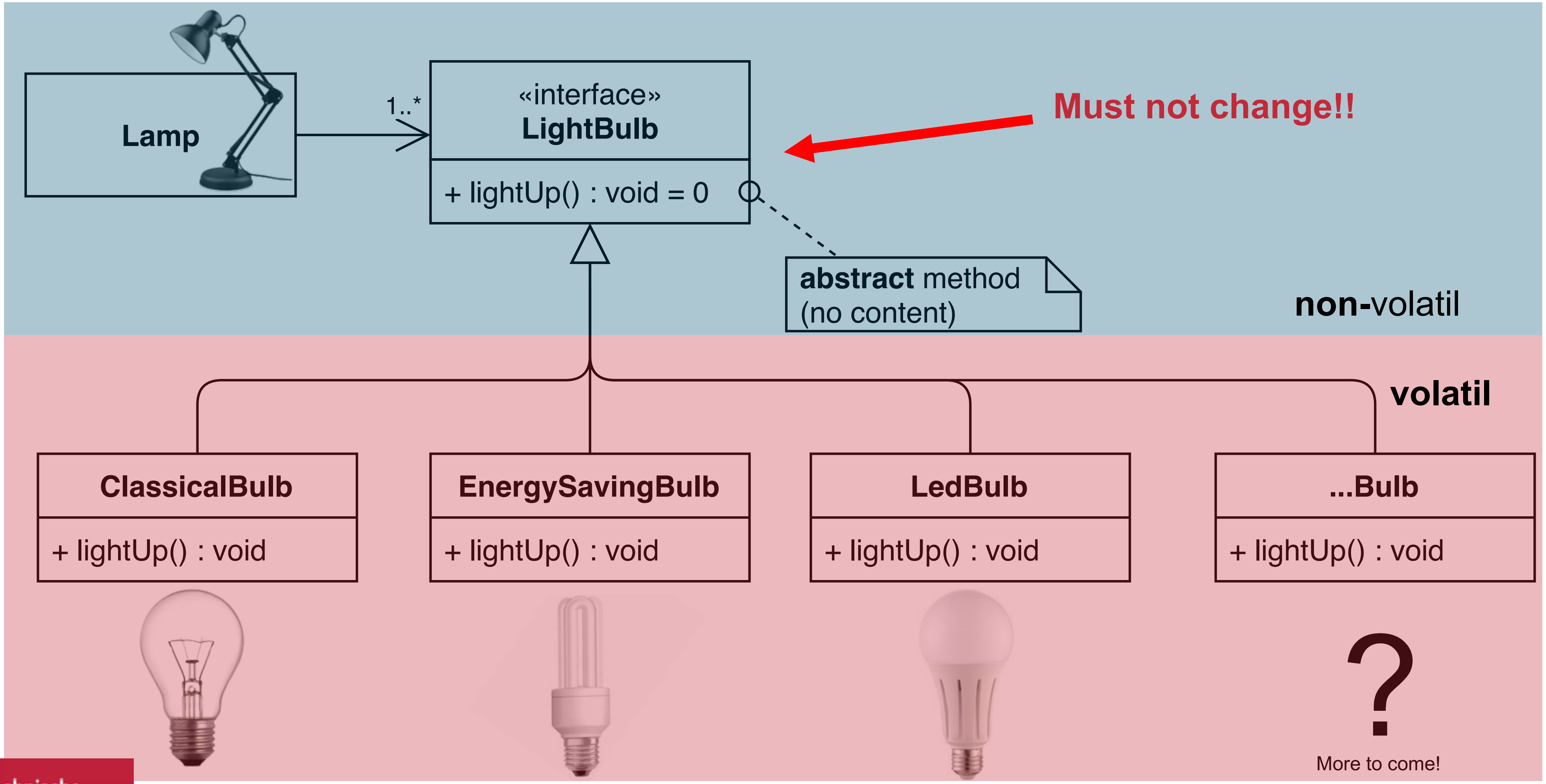
# Object Oriented Programming - Polymorphism

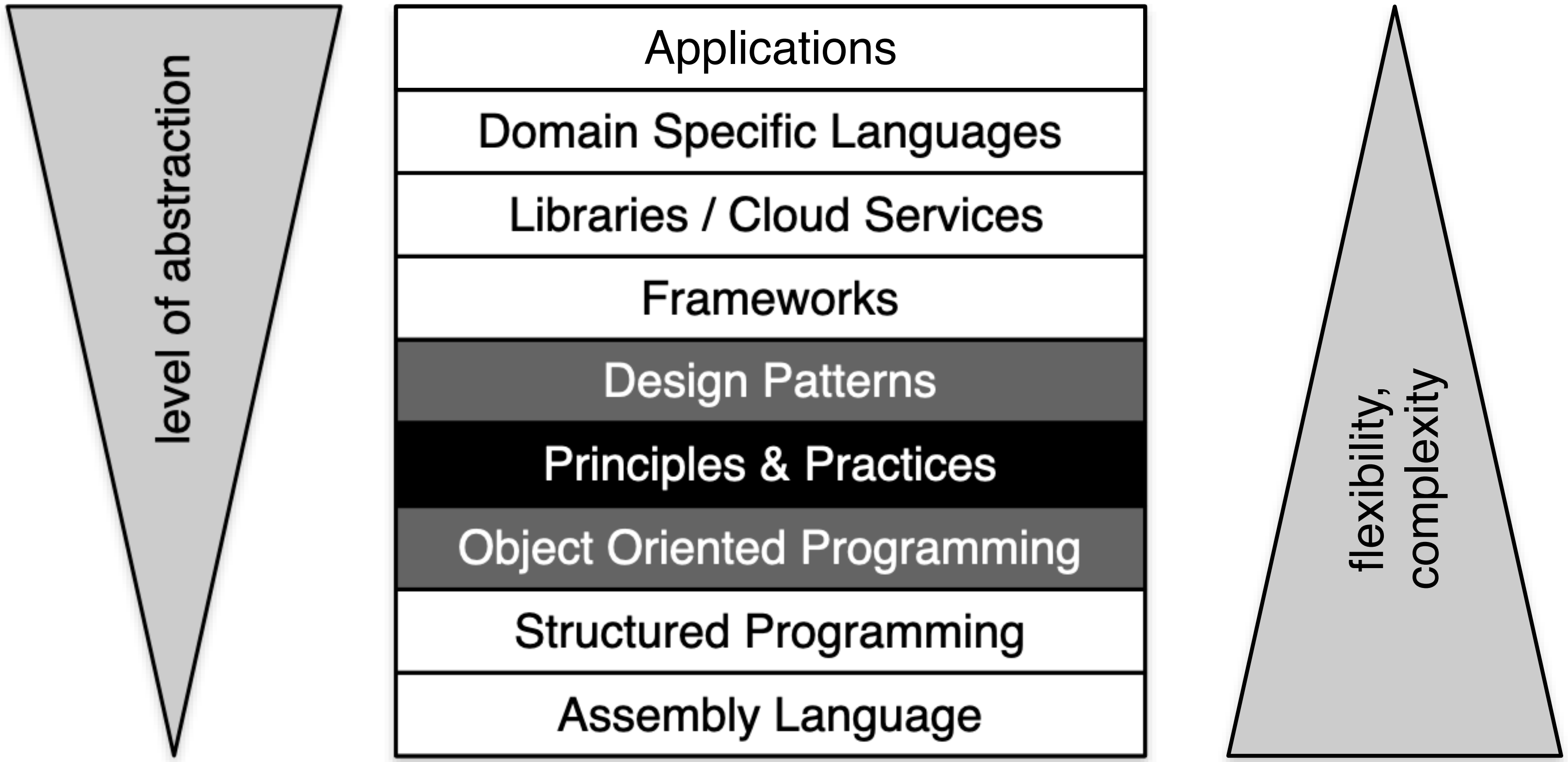


# Object Oriented Programming - Polymorphism



# Object Oriented Programming - Polymorphism




level of abstraction
Applications
Domain Specific Languages
Libraries / Cloud Services
Frameworks
Design Patterns
Principles & Practices
Object Oriented Programming
Structured Programming
Assembly Language
flexibility,
complexity

# Principles of object-oriented Design

# Do you know the SOLID Principles?

a) No, have never heard of them
b) I have heard of them, but I don’t apply them
c) I follow some of them
d) I can explain every single one of them
e) From time to time they even visit me in my dreams

Join at
slido.com
#123 444

# S.O.L.I.D. Principles

- **S**ingle Responsibility Principle (SRP)
- **O**pen-Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

…Guidelines (no laws) for object-oriented design!

[Robert C. Martin, Agile Principles, Patterns, and Practices, 2003]

# Single Responsibility Principle (SRP)
– Robert C. Martin [90]

# Single Responsibility Principle (SRP)

– Robert C. Martin [90]


everything in a class is related to a single purpose

***Single Responsibility Principle –***
*"Each class should only have one reason to change."* – Uncle Bob

Each Responsibility = Reason to change.

**Changes also affect depending classes**

Changes have to be made to depending classes

Possible failing behavior of depending classes (to be tested)

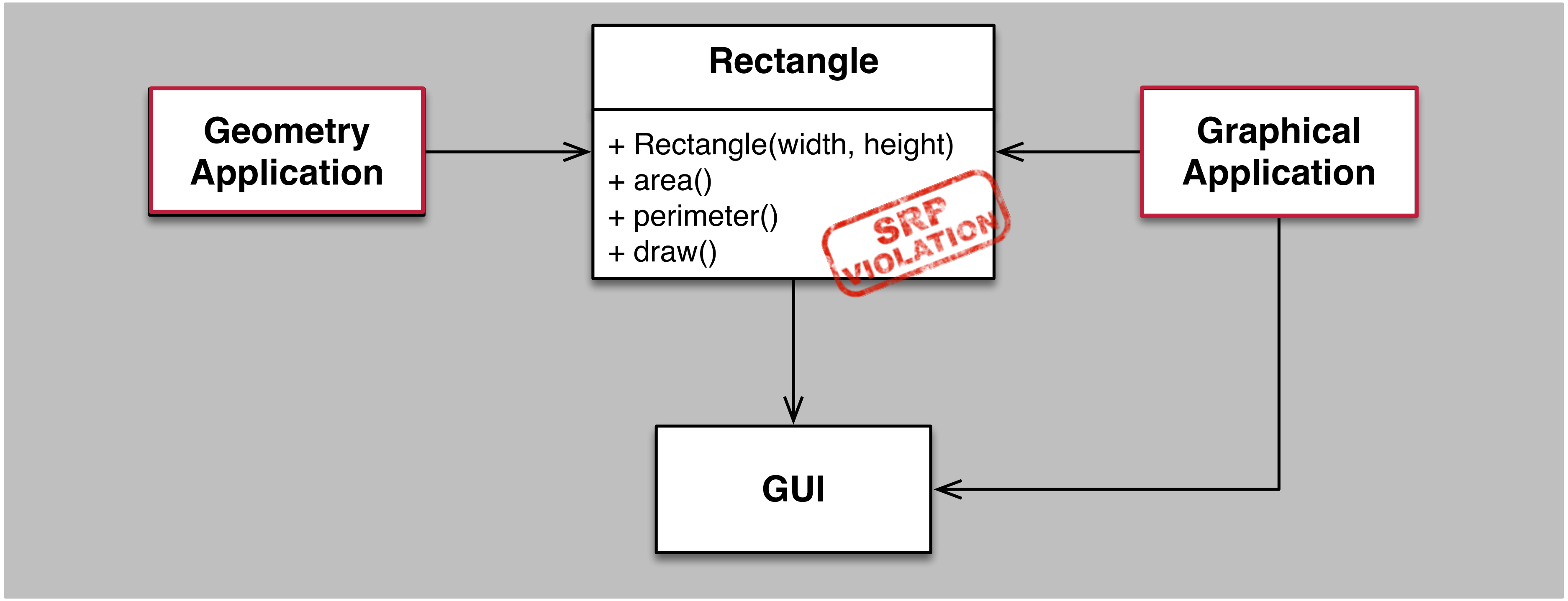
In C++ depending classes have to be recompiled unnecessarily after changes.

Classes that do more than one thing are **difficult to reuse**

[David L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, 1972]
[Tom DeMarco, Structured Analysis and System Specification, 1979]

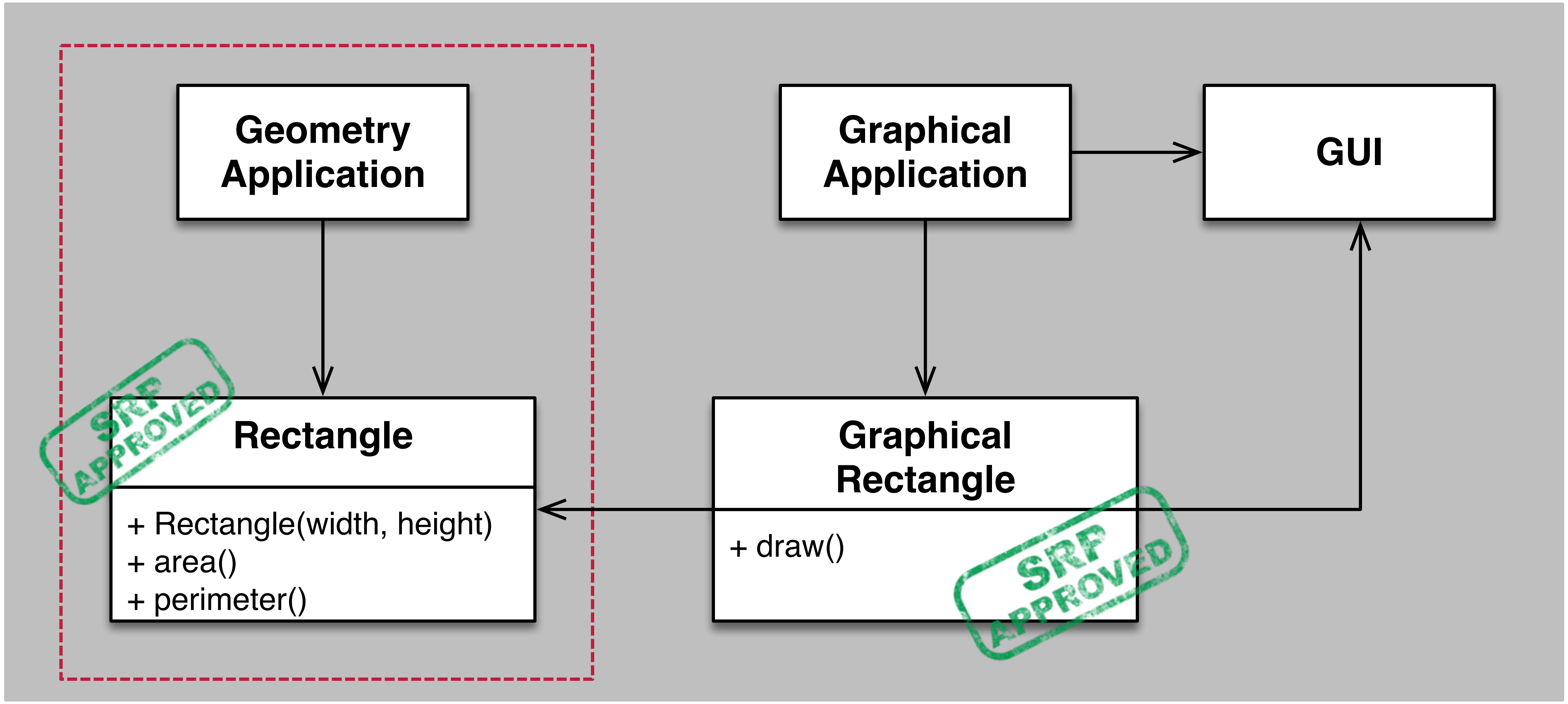
# Single Responsibility Principle (SRP)
– Robert C. Martin [90]

## Example – Violation of SRP:



# Single Responsibility Principle (SRP)
– Robert C. Martin [90]

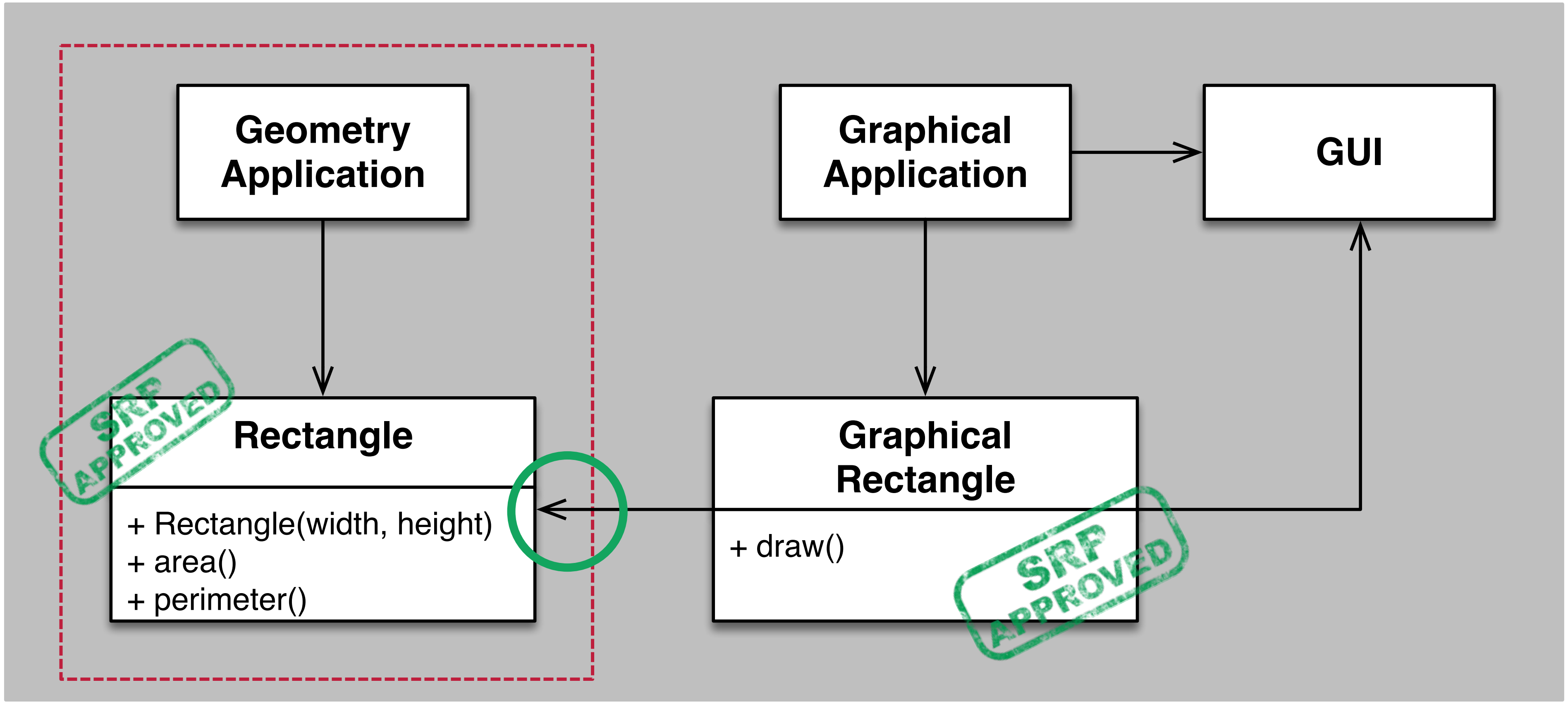
**Solution:**



# Single Responsibility Principle (SRP)

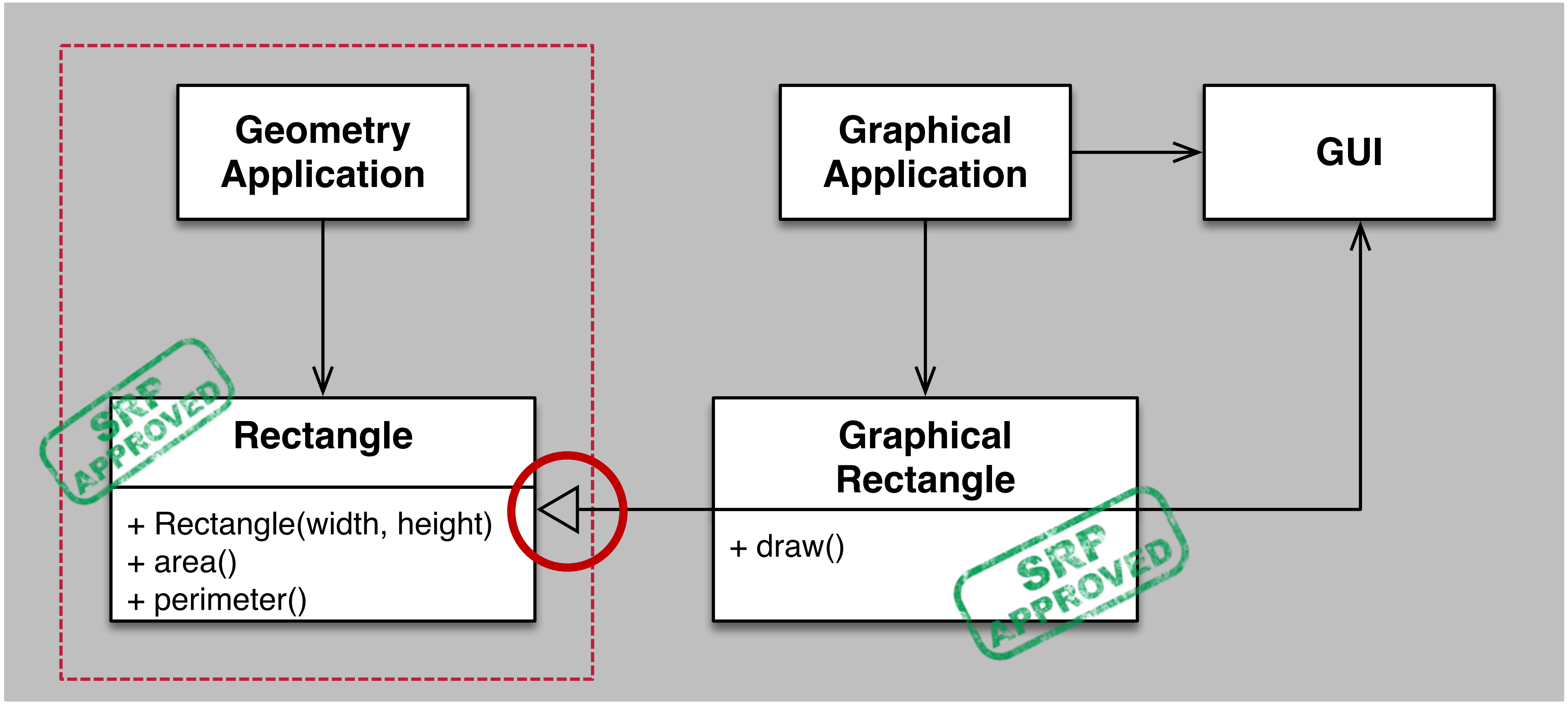
– Robert C. Martin [90]

**Solution:**



# Single Responsibility Principle (SRP)
– Robert C. Martin [90]

**Solution:**



# Open-Closed Principle (OCP)

- Bertrand Meyer, 1988


[Bertrand Meyer, Object-Oriented Software Construction, 1988]

# Open-Closed Principle (OCP)

- Bertrand Meyer, 1988


***Open for Extension, closed for modifications.***

*A module should be open for extensions, but closed for modifications.*

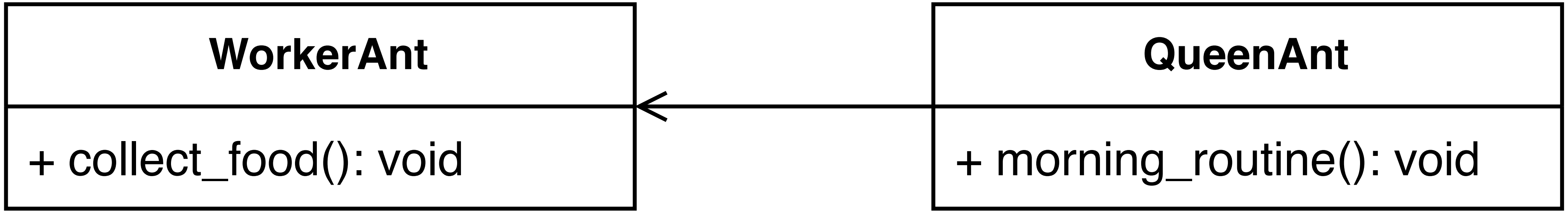
Principle ensures extensions of components **without** changing the source code of the specific component.

Tool: **Polymorphism**

Encapsulation of *volatile* and *non-volatile* Code (in Base Classes)

Abstract Core of the application remains **untouched**.

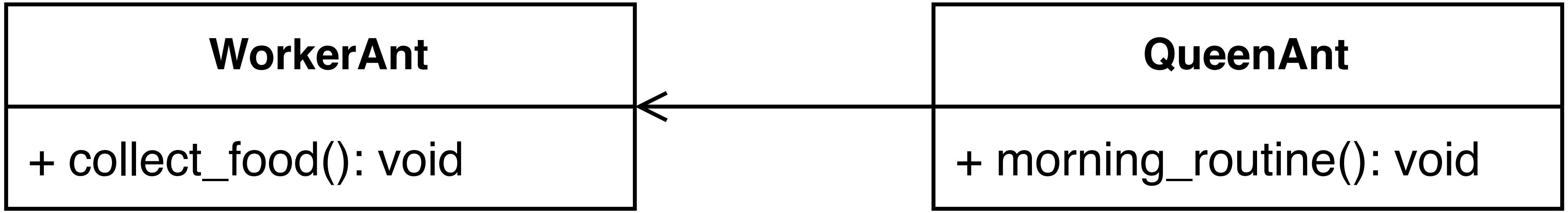
# Open-Closed Principle – Step 1




**Fine, as long nothing changes.**

# Open-Closed Principle – Step 1

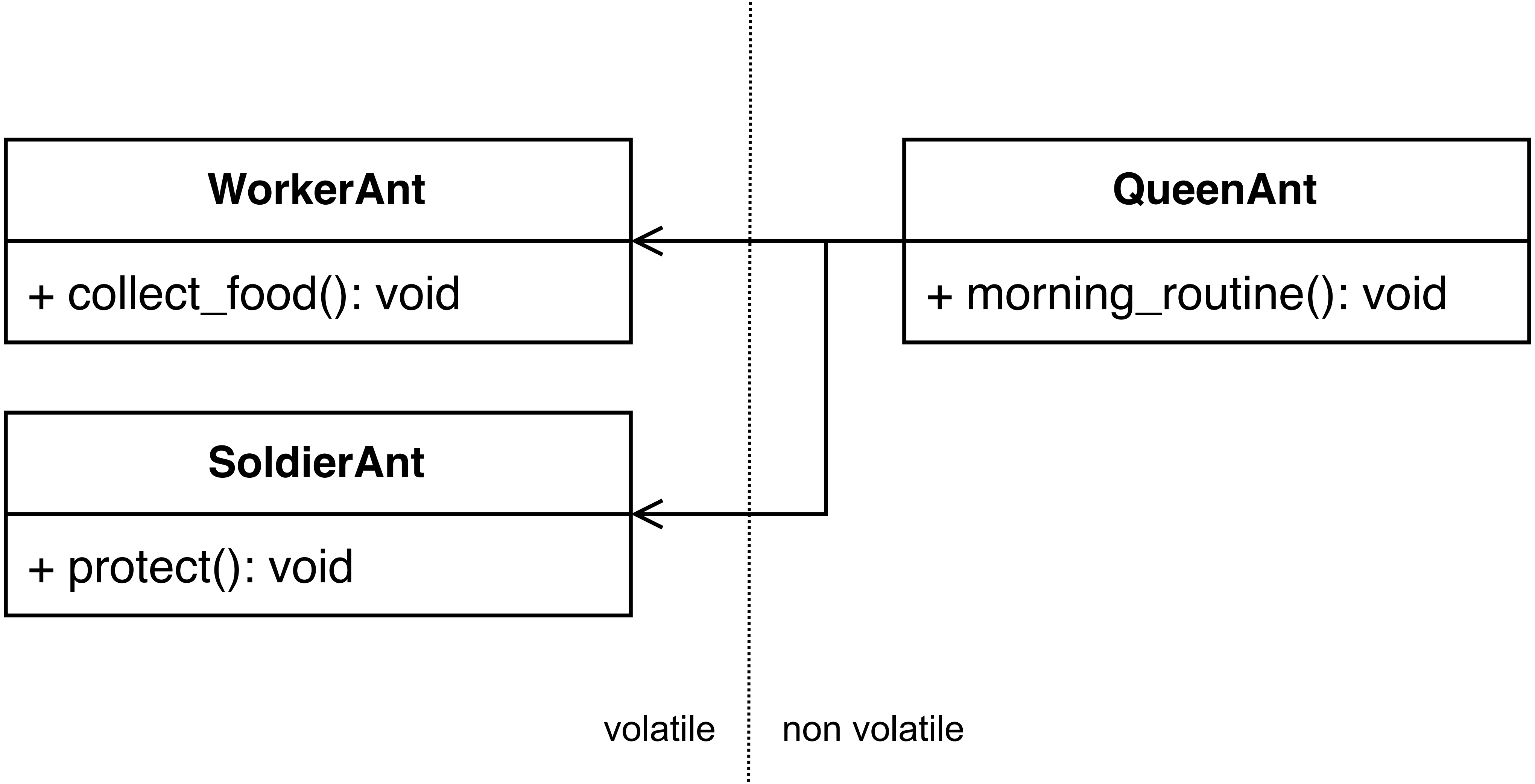
**Fine, as long nothing changes.**




```python
class WorkerAnt():
    def collect_food(self) -> None:
        print("I'm collecting")

worker = WorkerAnt()
queen = QueenAnt([worker])
queen.morning_routine()
```

```python
class QueenAnt():
    def __init__(self, ants) -> None:
        self._ants = ants

    def morning_routine(self) -> None:
        for ant in self._ants:
            ant.collect_food()
```

# Change: Open-Closed Principle – Step 2 – new Ant



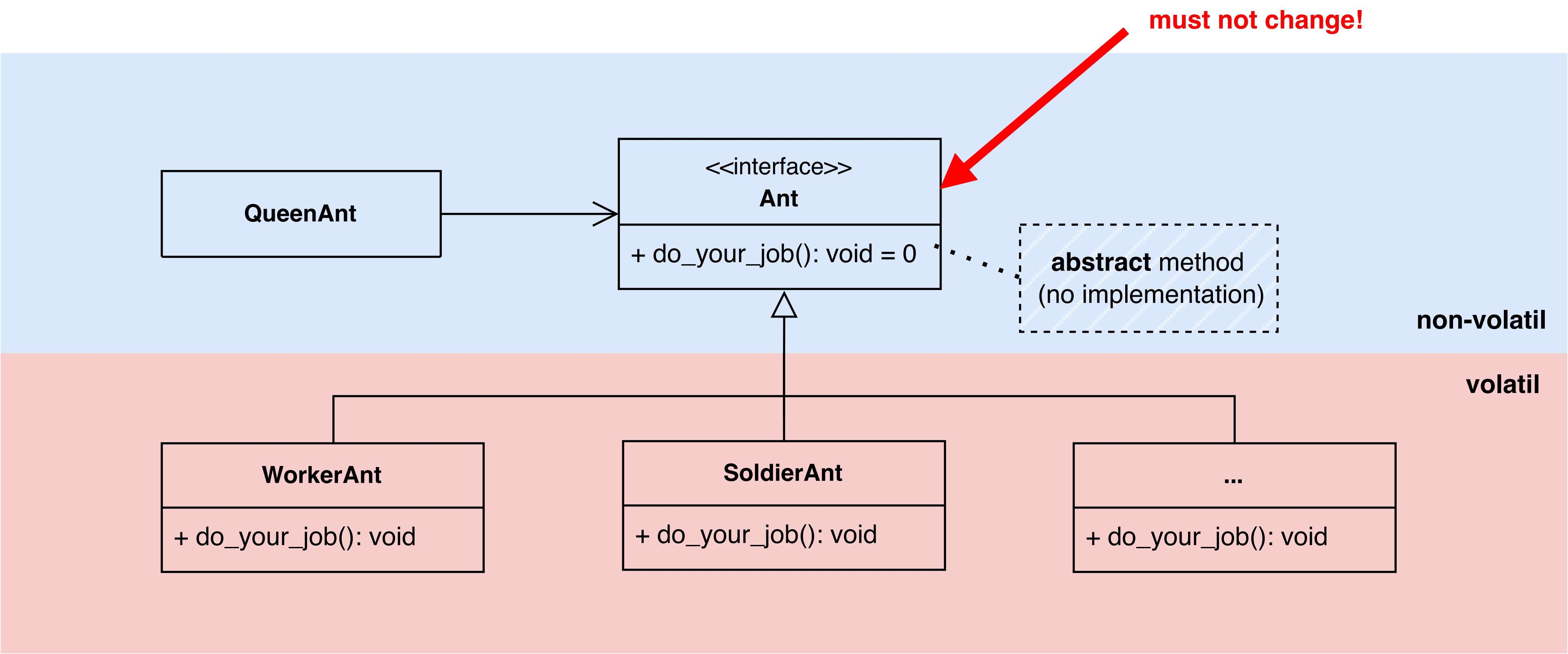
# Open-Closed Principle – Step 2


```python
class WorkerAnt():
    def collect_food(self) -> None:
        print("I'm collecting!")

class SoldierAnt():
    def protect(self) -> None:
        print("I'm protecting!")

worker = WorkerAnt()
soldier = SoldierAnt()
queen = QueenAnt([worker, soldier])
queen.morning_routine()
```

```python
class QueenAnt():
    def __init__(self, ants) -> None:
        self._ants = ants

    def morning_routine(self) -> None:
        for ant in self._ants:
            if isinstance(ant, WorkerAnt):
                ant.collect_food()
            elif isinstance(ant, SoldierAnt):
                ant.protect()
```

volatile | non volatile

# Open-Closed Principle – Solution: Interface (Protocol)



# Open-Closed Principle – Solution: Interface (Protocol)

OCP APPROVED

```python
class WorkerAnt(Ant):
    def do_your_job(self) -> None:
        print("I'm collecting!")

class SoldierAnt(Ant):
    def do_your_job(self) -> None:
        print("I'm protecting!")

worker = WorkerAnt()
soldier = SoldierAnt()
queen = QueenAnt([worker, soldier])
queen.morning_routine()
```

volatile

**Adding a Protocol (Interface):**
Being <u>specific</u> about what messages *ants* needs to understand.

```python
from typing import Protocol

class Ant(Protocol):
    def do_your_job(self) -> None:
        raise NotImplementedError

class QueenAnt():
    def __init__(self, ants: list[Ant]) -> None:
        self._ants = ants

    def morning_routine(self) -> None:
        for ant in self._ants:
            ant.do_your_job()

```

non volatile

# 1. With Protocol and Inheritance

```python
class WorkerAnt(Ant):
    def do_your_job(self) -> None:
        print("I'm collecting!")

class SoldierAnt(Ant):
    def protect(self) -> None:
        print("I'm protecting!")

worker = WorkerAnt()
soldier = SoldierAnt()
queen = QueenAnt([worker, soldier])
queen.morning_routine()
```

**MyPy error!**

*Cannot instantiate abstract class "SoldierAnt" with abstract attribute "do_your_job"*

volatile

```python
from typing import Protocol

class Ant(Protocol):
    def do_your_job(self) -> None:
        raise NotImplementedError

class QueenAnt():
    def __init__(self, ants: list[Ant]) -> None:
        self._ants = ants

    def morning_routine(self) -> None:
        for ant in self._ants:
            ant.do_your_job()
```

non volatile

# 2. With Protocol – Without Inheritance

```python
class WorkerAnt(Ant):
    def do_your_job(self) -> None:
        print("I'm collecting!")

class SoldierAnt():
    def protect(self) -> None:
        print("I'm protecting!")

worker = WorkerAnt()
soldier = SoldierAnt()
queen = QueenAnt([worker, soldier])
queen.morning_routine()
```

**MyPy error!**

*List item 1 has incompatible type "SoldierAnt"; expected "Ant"*

```python
from typing import Protocol

class Ant(Protocol):
    def do_your_job(self) -> None:
        raise NotImplementedError

class QueenAnt():
    def __init__(self, ants: list[Ant]) -> None:
        self._ants = ants

    def morning_routine(self) -> None:
        for ant in self._ants:
            ant.do_your_job()
```

volatile | non volatile

# 3. Without Protocol

```python
class WorkerAnt():
    def do_your_job(self) -> None:
        print("I'm collecting!")

class SoldierAnt():
    def protect(self) -> None:
        print("I'm protecting!")


worker = WorkerAnt()
soldier = SoldierAnt()
queen = QueenAnt([worker, soldier])
queen.morning_routine()
```

```python
class QueenAnt():
    def __init__(self, ants) -> None:
        self._ants = ants

    def morning_routine(self) -> None:
        for ant in self._ants:
            ant.do_your_job()
```

**If not: runtime error!**

*AttributeError: 'SoldierAnt' object has no attribute ‘do_your_job()’*

volatile | non volatile

# Protocol

Protocol types allows developers to define and enforce a set of methods that classes must implement in order to satisfy a particular interface (like a contract).

**Advantages using Protocols:**

- Reusable Interfaces
- Static Duck Typing ("Type Checking")
- Compatibility

# Liskov Substitution Principle (LSP)

- Barbara Liskov, 1987


[Barbara Liskov, Data Abstraction and Hierarchy, 1987]

# Liskov Substitution Principle (LSP)

- Barbara Liskov, 1987

***Liskov Substitution Principle***

*Type `T'` is a Subtype of `T`, if objects of Type `T` can be exchanged by objects of Type `T'` without any limitations.*

Tight relationship to Inheritance and `virtual/abstract` methods

Simple Example:

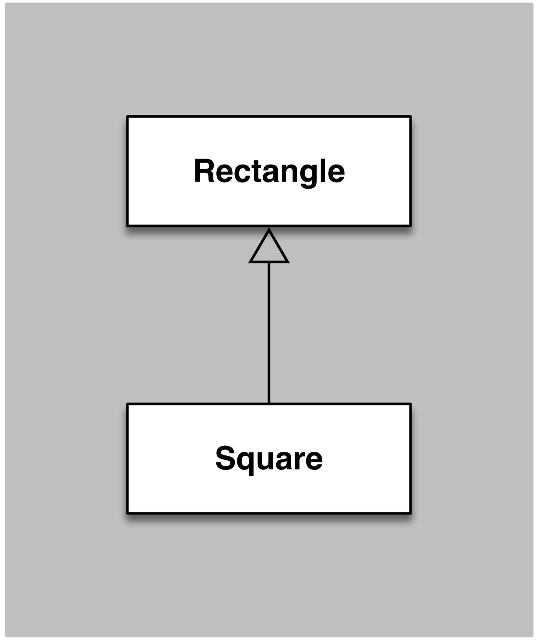
A driver of a BMW shouldn‘t be surprised, if he tries to drive a VW.

(Since both are cars and should work in kind of the same way, when trying to interact with similar functions.)

# Liskov Substitution Principle (LSP)

- Barbara Liskov, 1987

## Example – Violation of LSP




- Situation:
  Class `Rectangle` is existing.
  Class `Square` is needed.
- Inheritance <u>seems</u> right:
  A `Square` **IS** a `Rectangle`

# What do you think – is a square a rectangle?

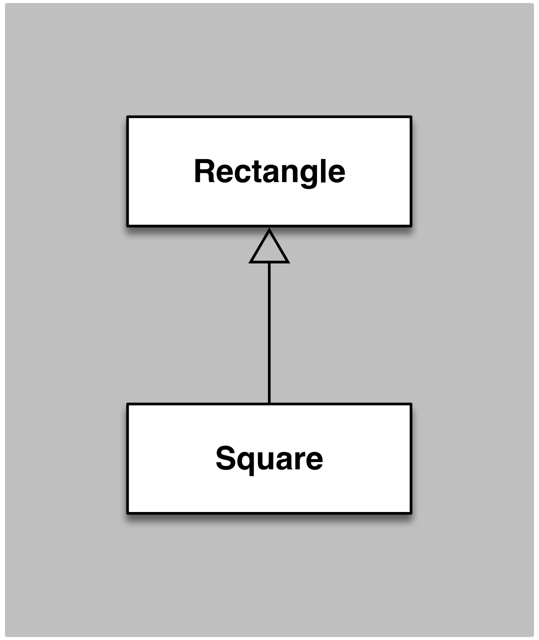

# Liskov Substitution Principle (LSP)

- Barbara Liskov, 1987

```
class Rectangle:
    def __init__(self, height=1.0, width=1.0):
        self._height: float = height
        self._width: float = width

    def set_height(self, height: float):
        self._height = height

    def set_width(self, width: float):
        self._width = width

    def area(self) -> float:
        return self._height * self._width
```

**First doubts**

- properties of `Rectangle`: 2 (height & width) required properties of `Square`: 1
- ok, memory is cheap...



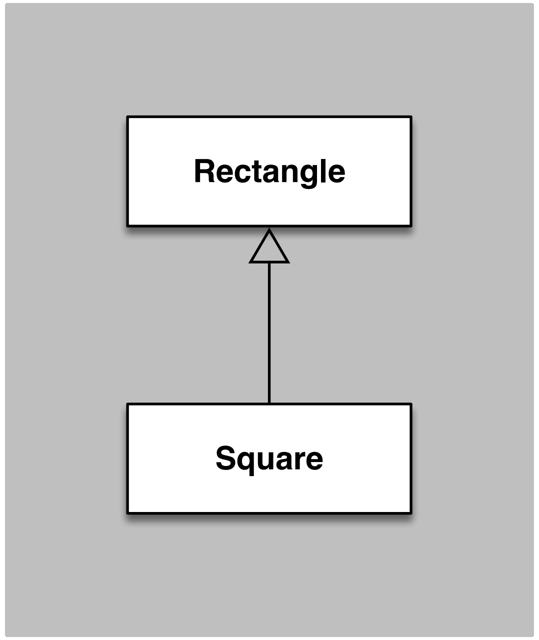
# Liskov Substitution Principle (LSP)

- Barbara Liskov, 1987

```
class Square(Rectangle):
    def __init__(self, width=1.0):
        self._height = width
        self._width = width

    def set_height(self, height: float):
        self._height = height
        self._width = height

    def set_width(self, width: float):
        self._width = width
        self._height = width
```

**Further doubts**

- properties: height & width
- names aren‘t suitable
- height and width have to be consistent



# Liskov Substitution Principle (LSP)

- Barbara Liskov, 1987

```
def foo(r: Rectangle):
  r.set_width(5)
  r.set_height(4)
  if r.area() != 20:
    print("Bad area!")
```

**Unpreventable Error:**

legitimate assumption for `Rectangle`:

If height is changed, the width stays untouched!

But wrong polymorphic behavior of `Square`!

# Liskov Substitution Principle (LSP)

- Barbara Liskov, 1987

**Violation of the Open Closed Principle (OCP)**

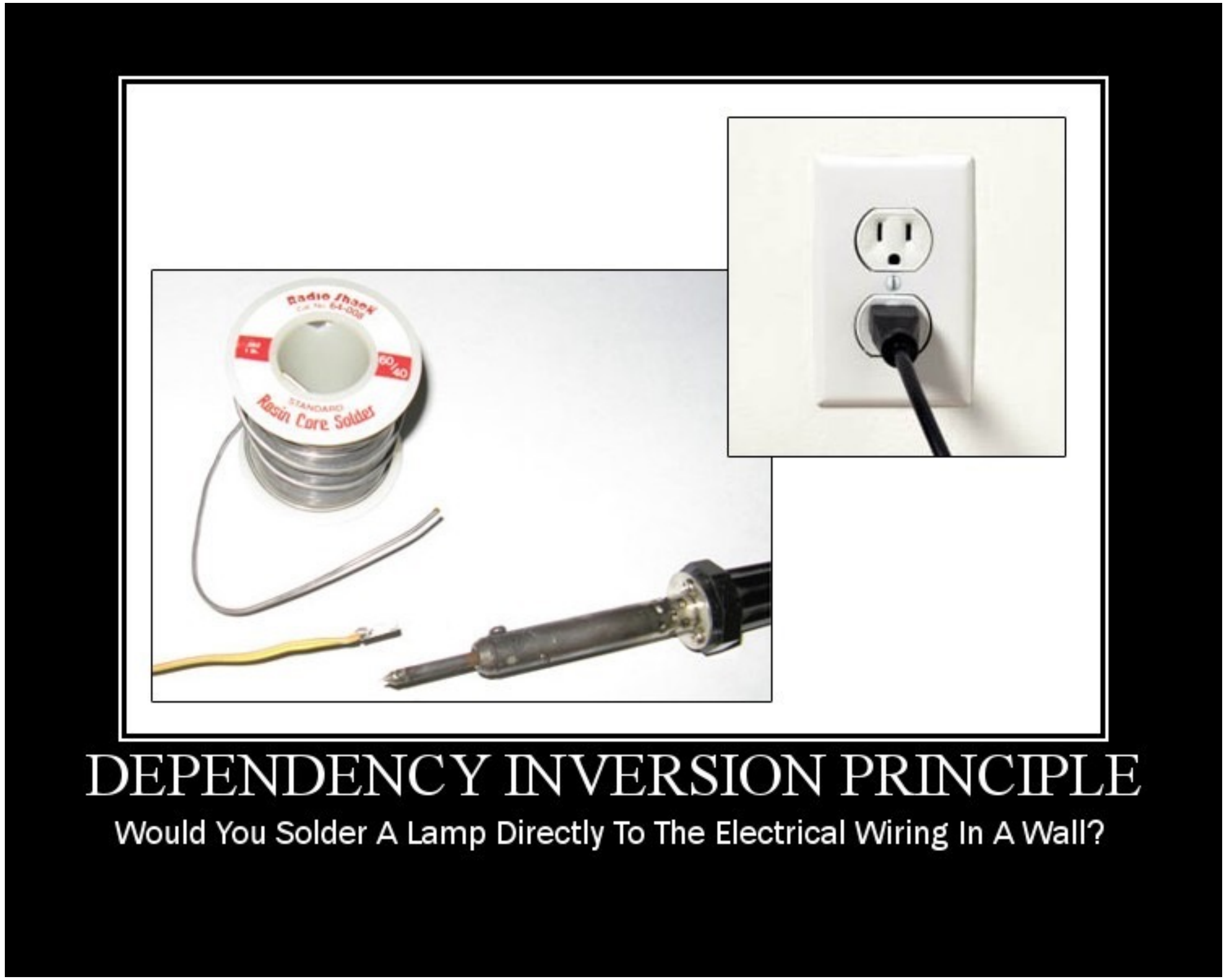
Using the `Square` class introduces a dependency to the subtype of `Rectangle`!

```
def foo(r: Rectangle):
  r.set_width(5)
  r.set_height(4)
  if r.area() != 20 and not isinstance(r, Square):
    print("Bad area!")
```


**Conclusion:**

Although a `Square` is a `Rectangle` in a geometric sense, this isn‘t true in the sense of software (polymorphism)!

# Dependency Inversion Principle (DIP)



# Dependency Inversion Principle (DIP)

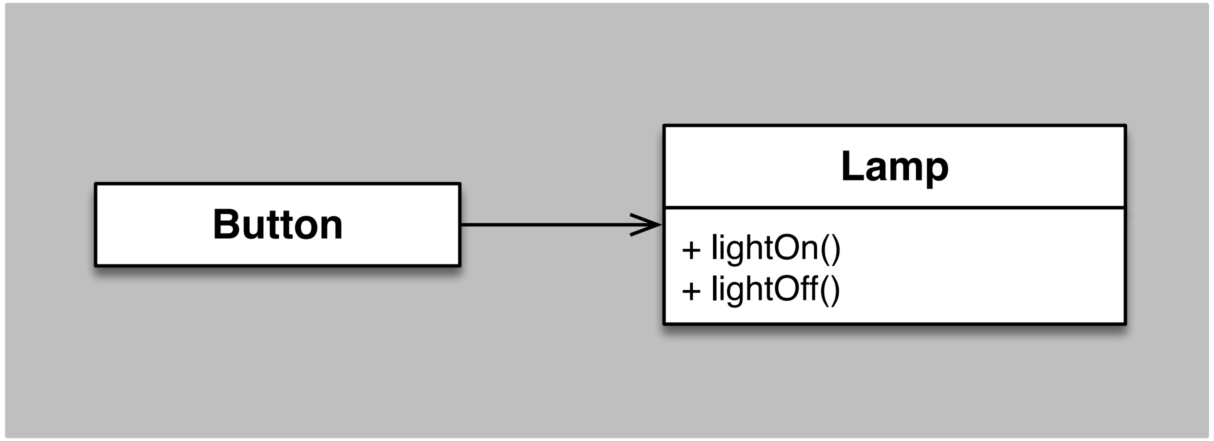
**Dependency Inversion Principle**

a) *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
b) *Abstractions should not depend upon details. Details should depend upon abstractions.*

But: If a concrete class is not going to change very often, and no other similar derivatives are going to be created, it does very little harm to depend on it.

# Dependency Inversion Principle (DIP)

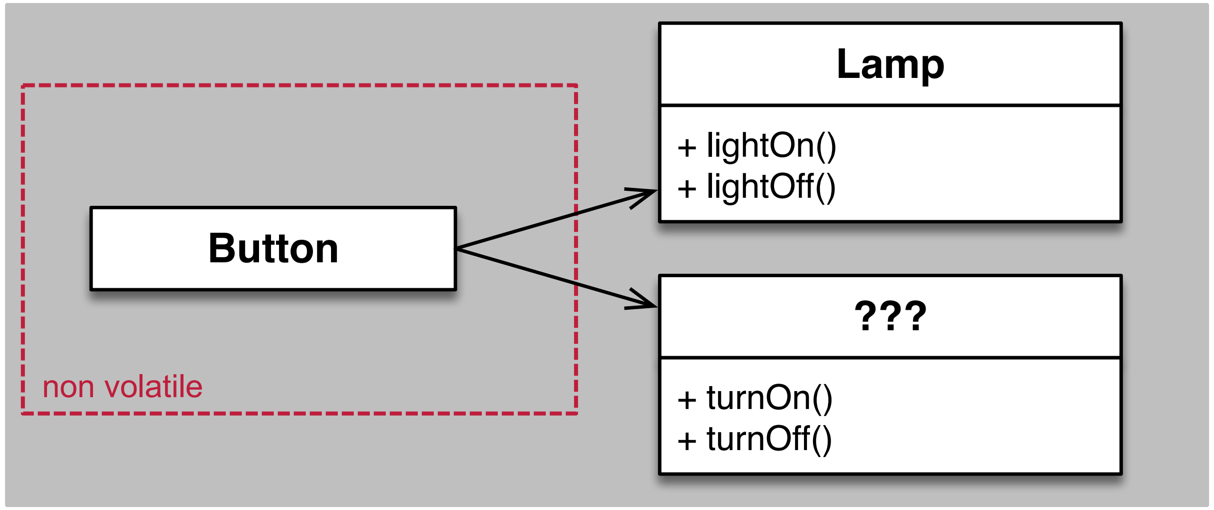
## Example – Violation of DIP




*Do you see a problem here?*

# Dependency Inversion Principle (DIP)

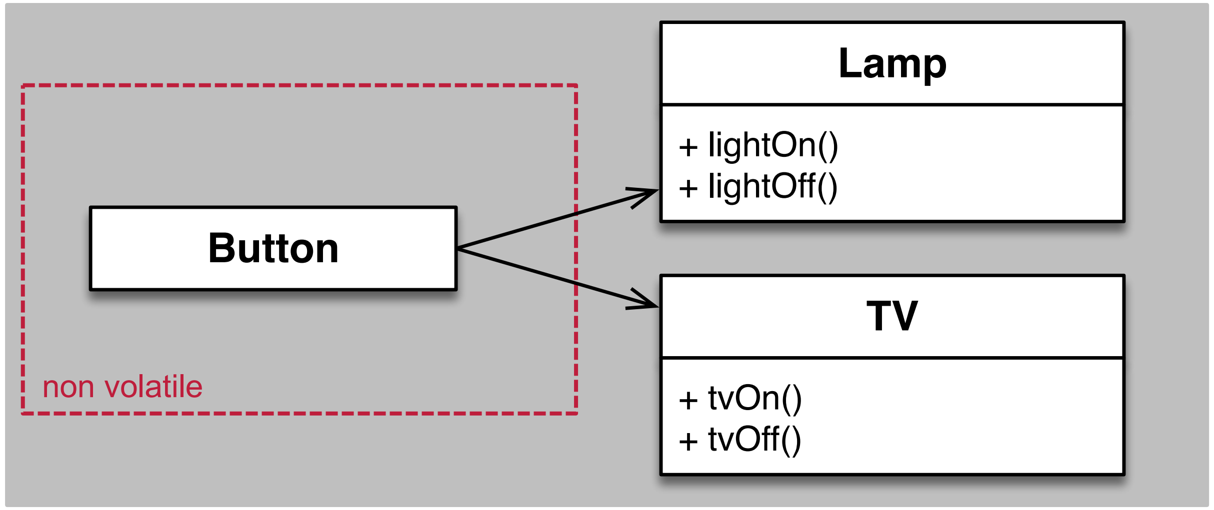
## Example – Violation of DIP




*Do you see a problem here?*

# Dependency Inversion Principle (DIP)

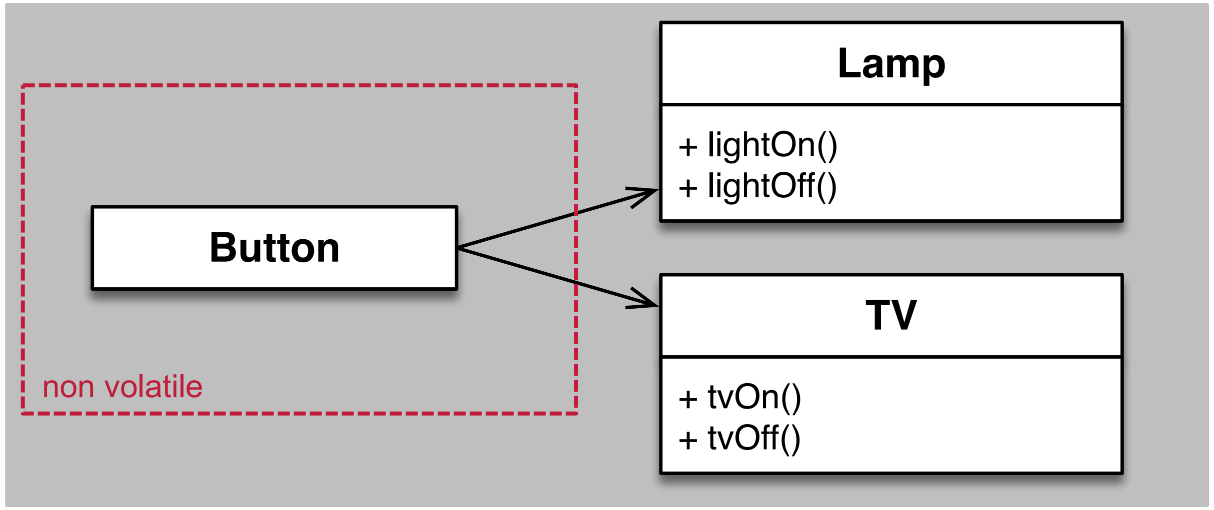
## Example – Violation of DIP




*Do you see a problem here?*

# Dependency Inversion Principle (DIP)


## Example – Violation of DIP




*Button* is directly dependent on *Lamp.*

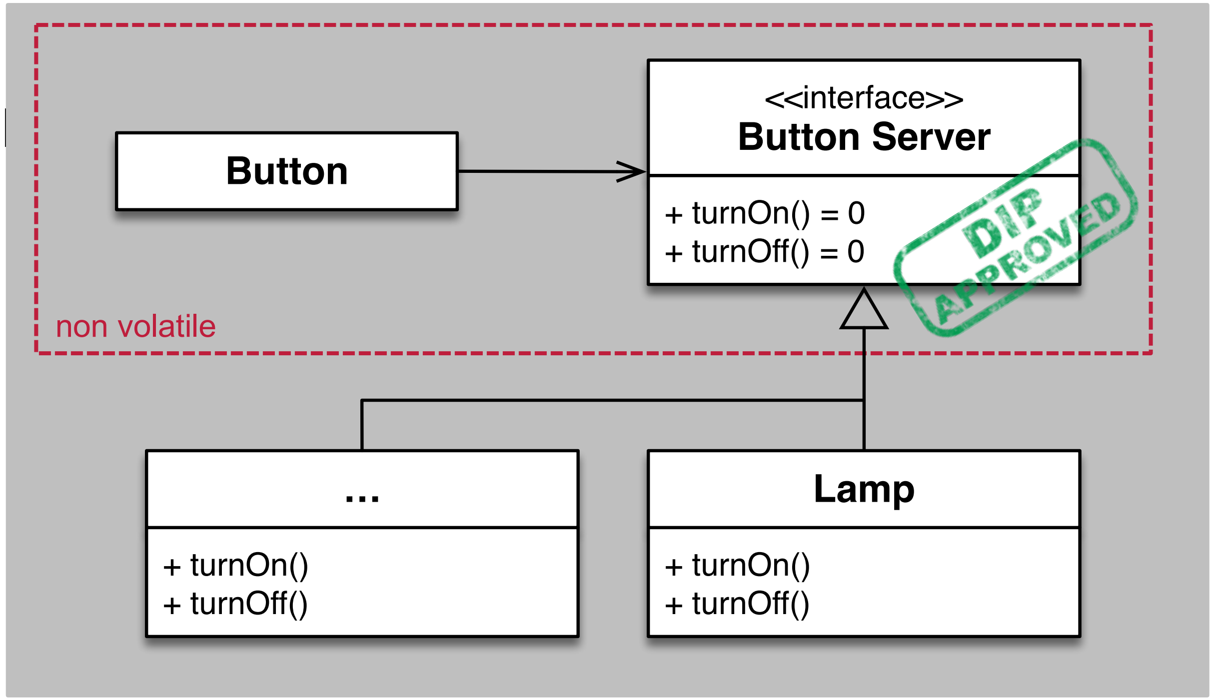
Changes in *Lamp* have a direct impact on *Button.*

*Button* can only control objects of *Lamp.*

*Button* can‘t be reused.

# Dependency Inversion Principle (DIP)

## So



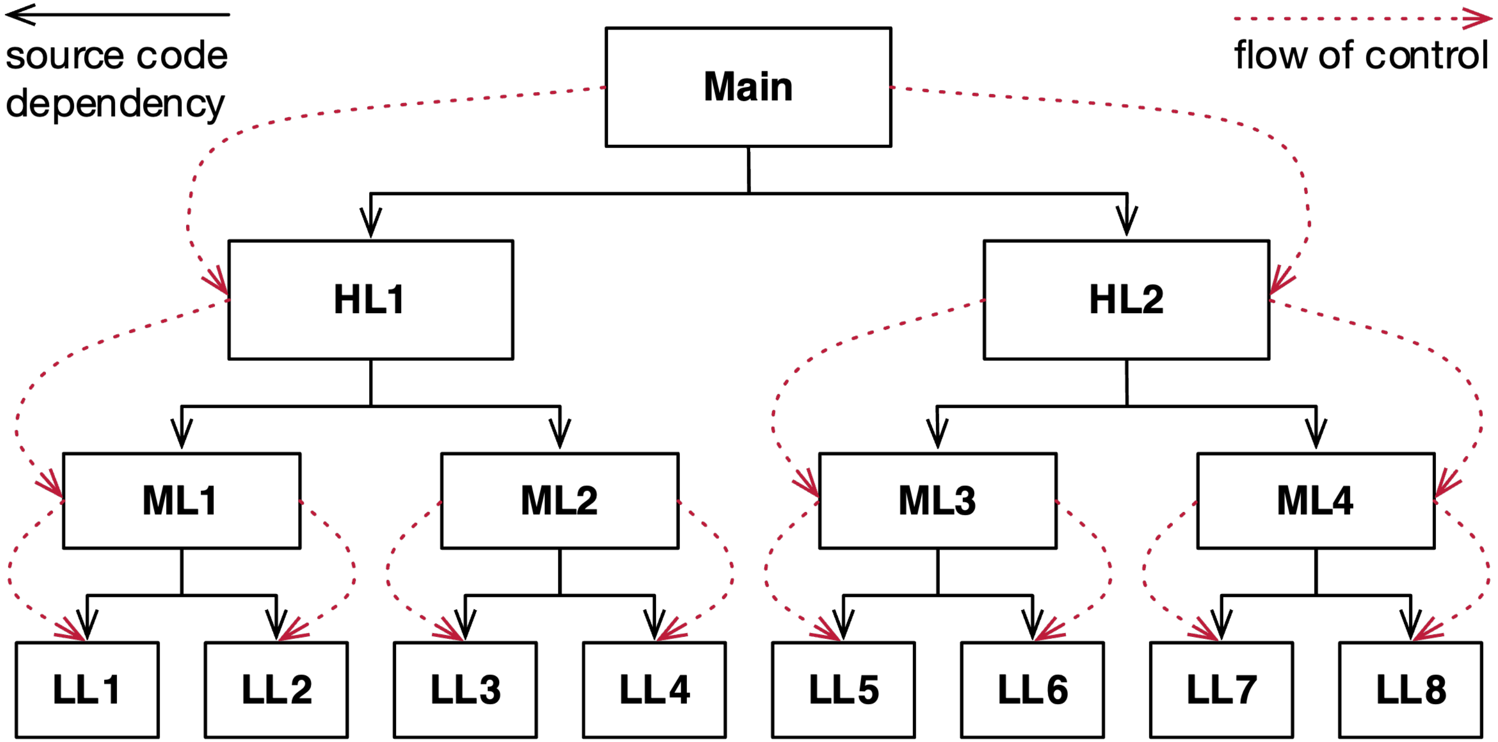

`Button` is now depending on an interface called `ButtonServer`.

Changes in `Lamp` won‘t influence the `Button`.

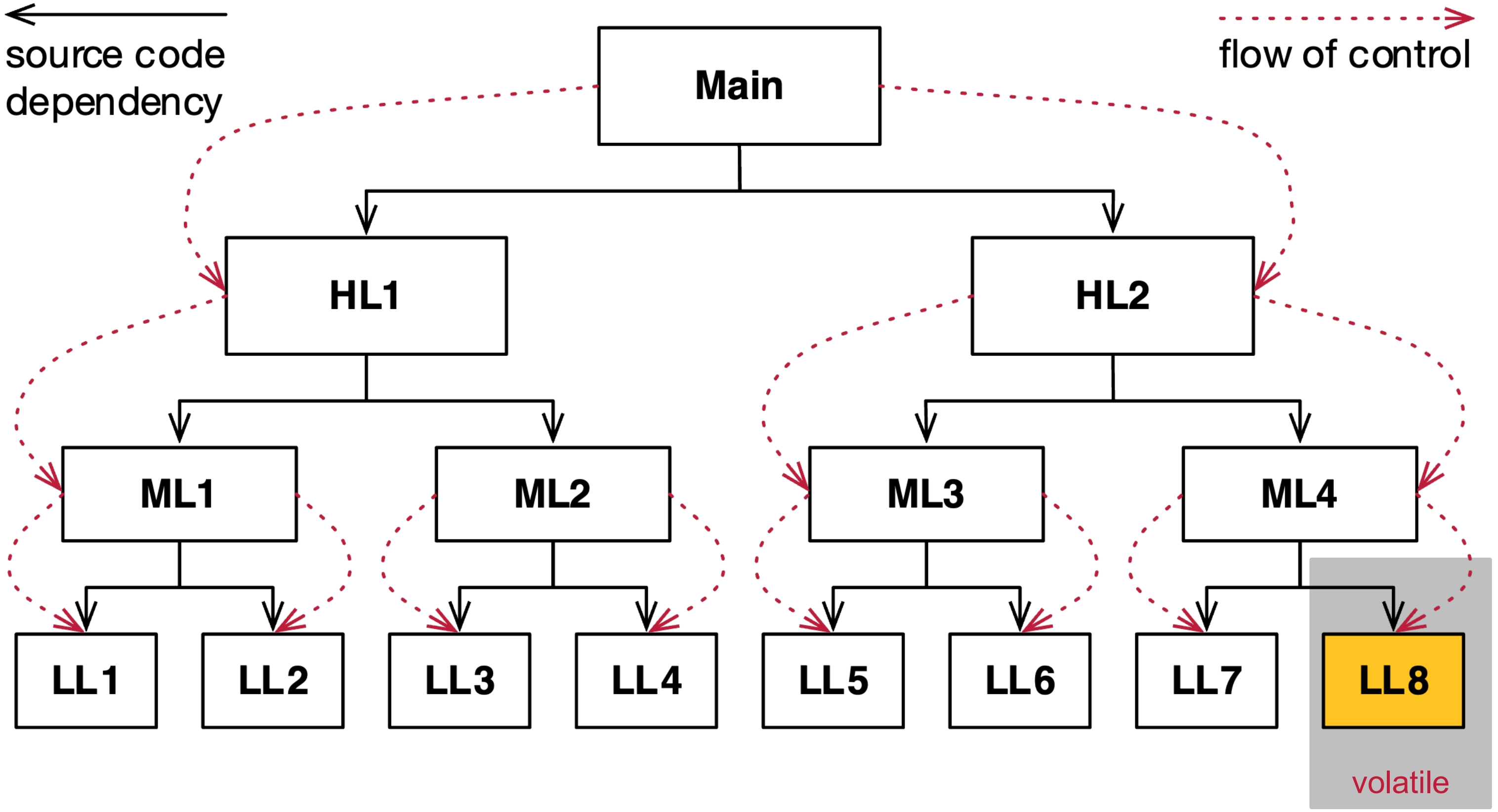
`Button` can be reused.

`Button` can control any object which implements the interface of `ButtonServer`.

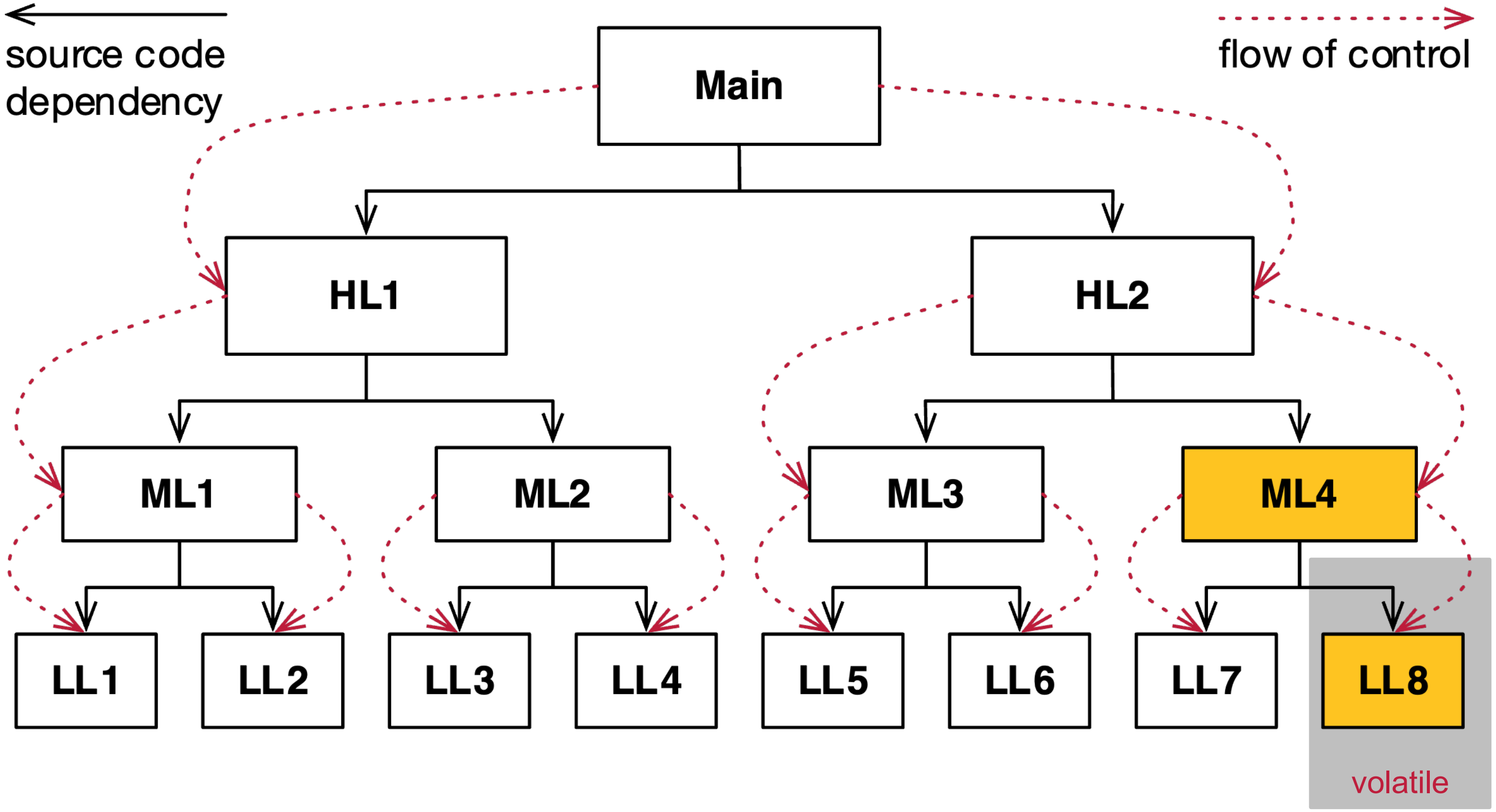
# Dependencies make changes difficult



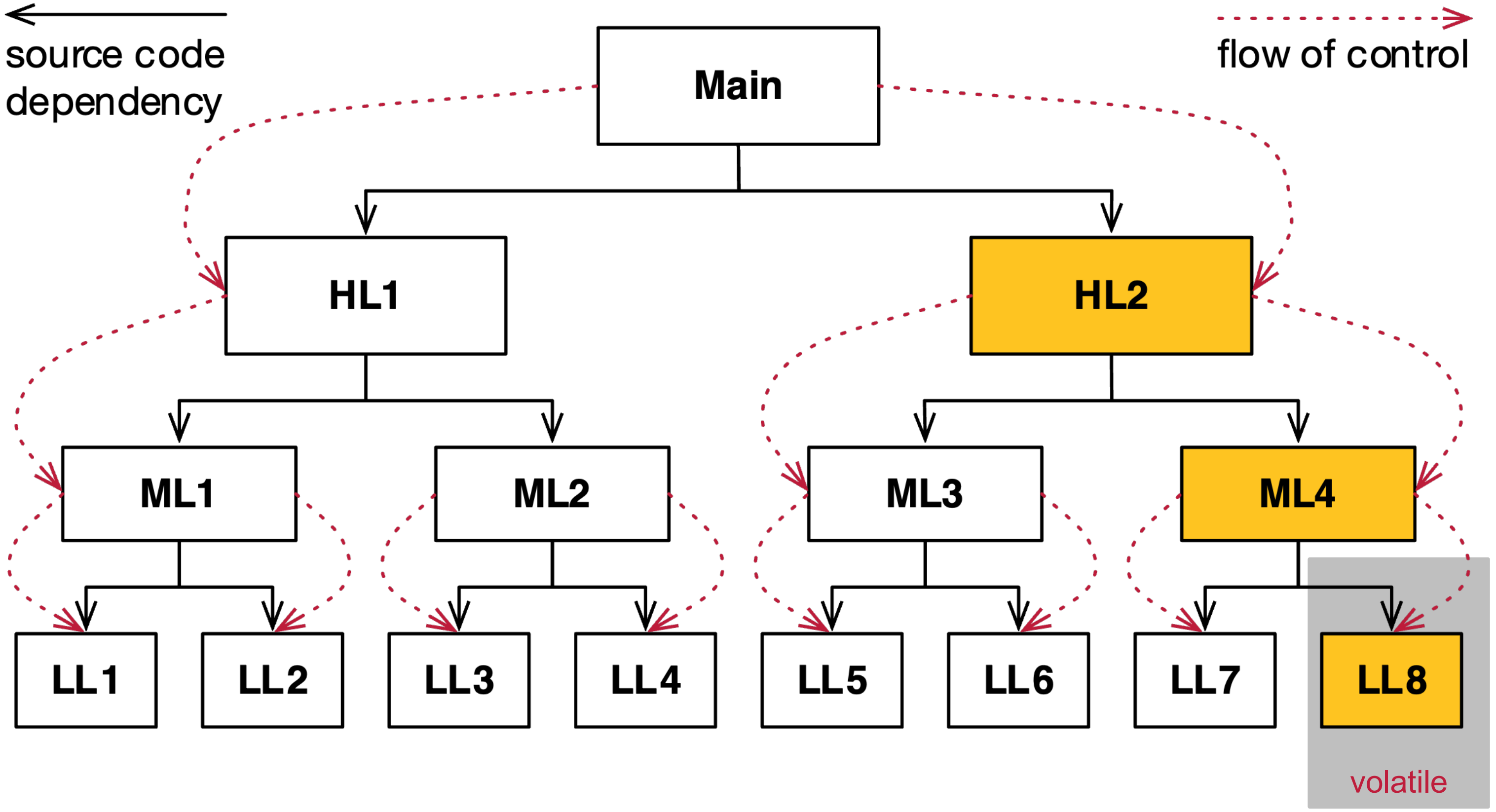
# Dependencies propagate change



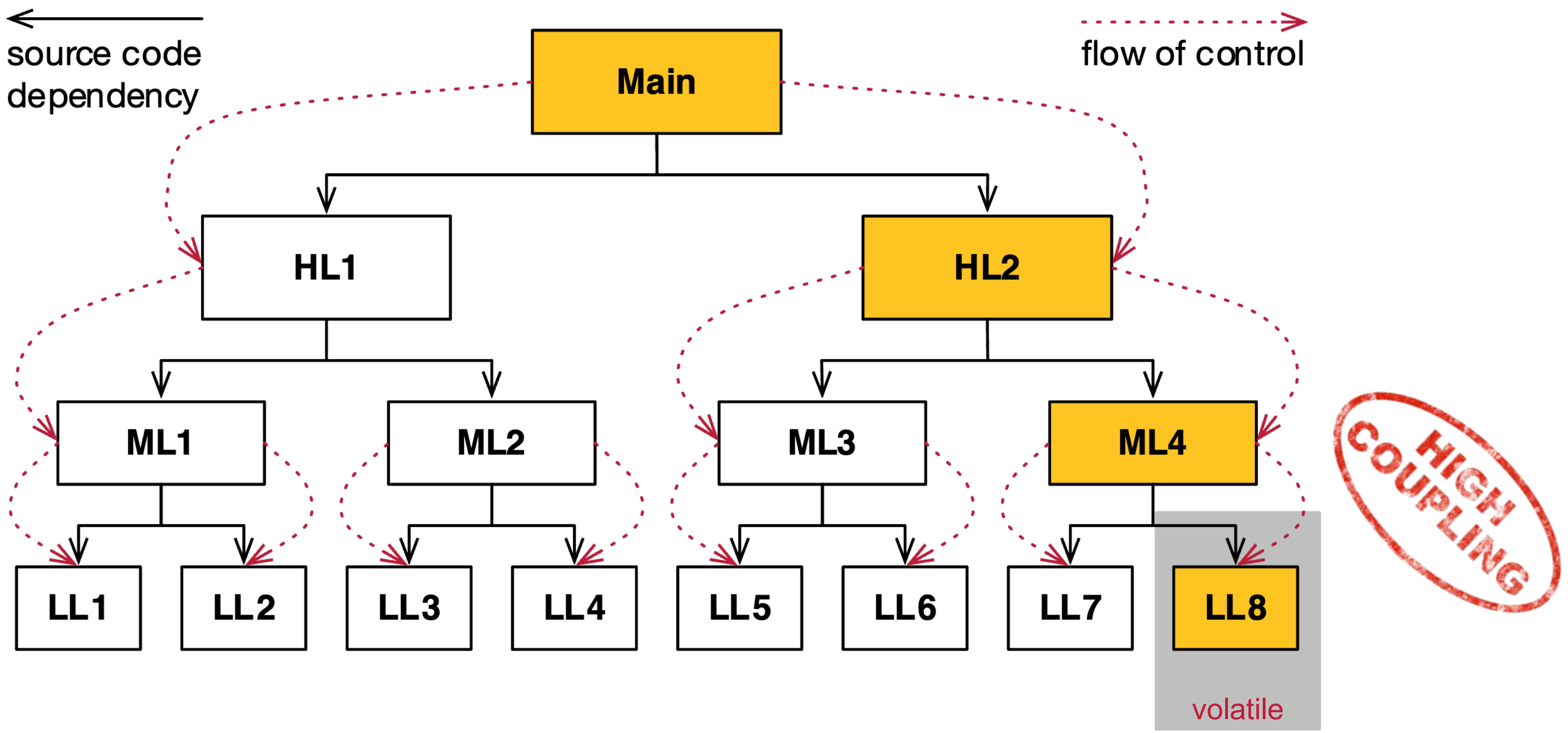
# Dependencies propagate change



# Dependencies propagate change

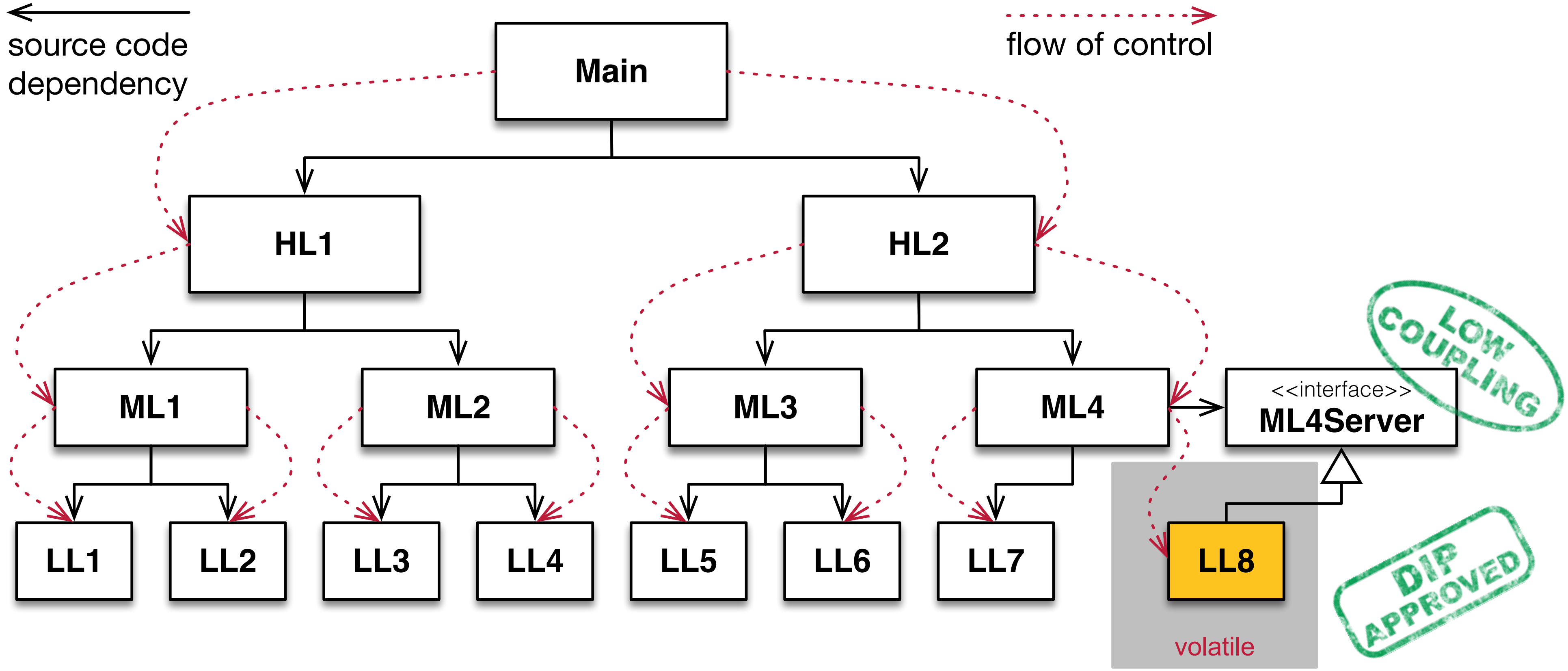


# Dependencies propagate change



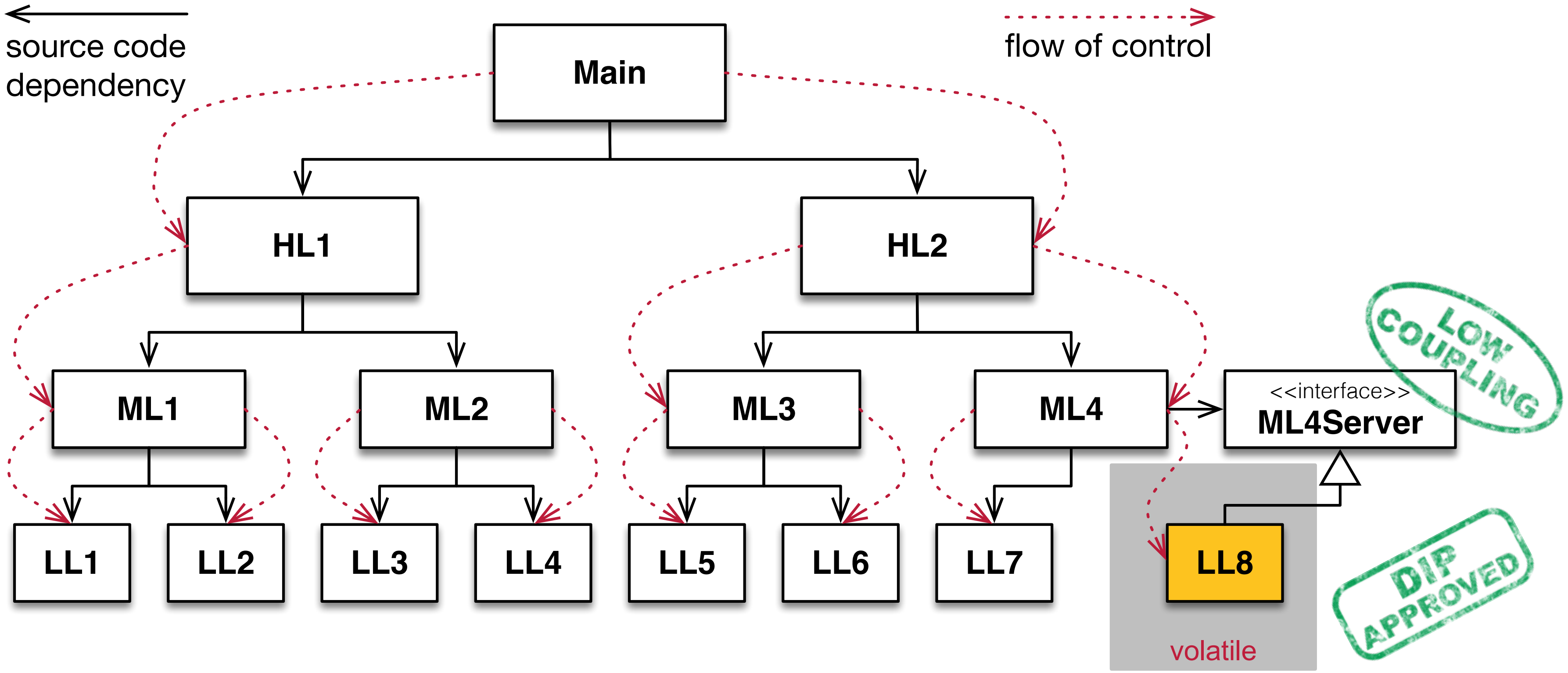

`ML4` is directly dependent on `LL8`
Changes in `LL8` have a direct impact on `ML4`
`ML4` can only work with objects of `LL8`
`ML4` can‘t be reused without `LL8`

# Inverted direction of dependency



# Dependency Inversion Principle (DIP)

## Solution



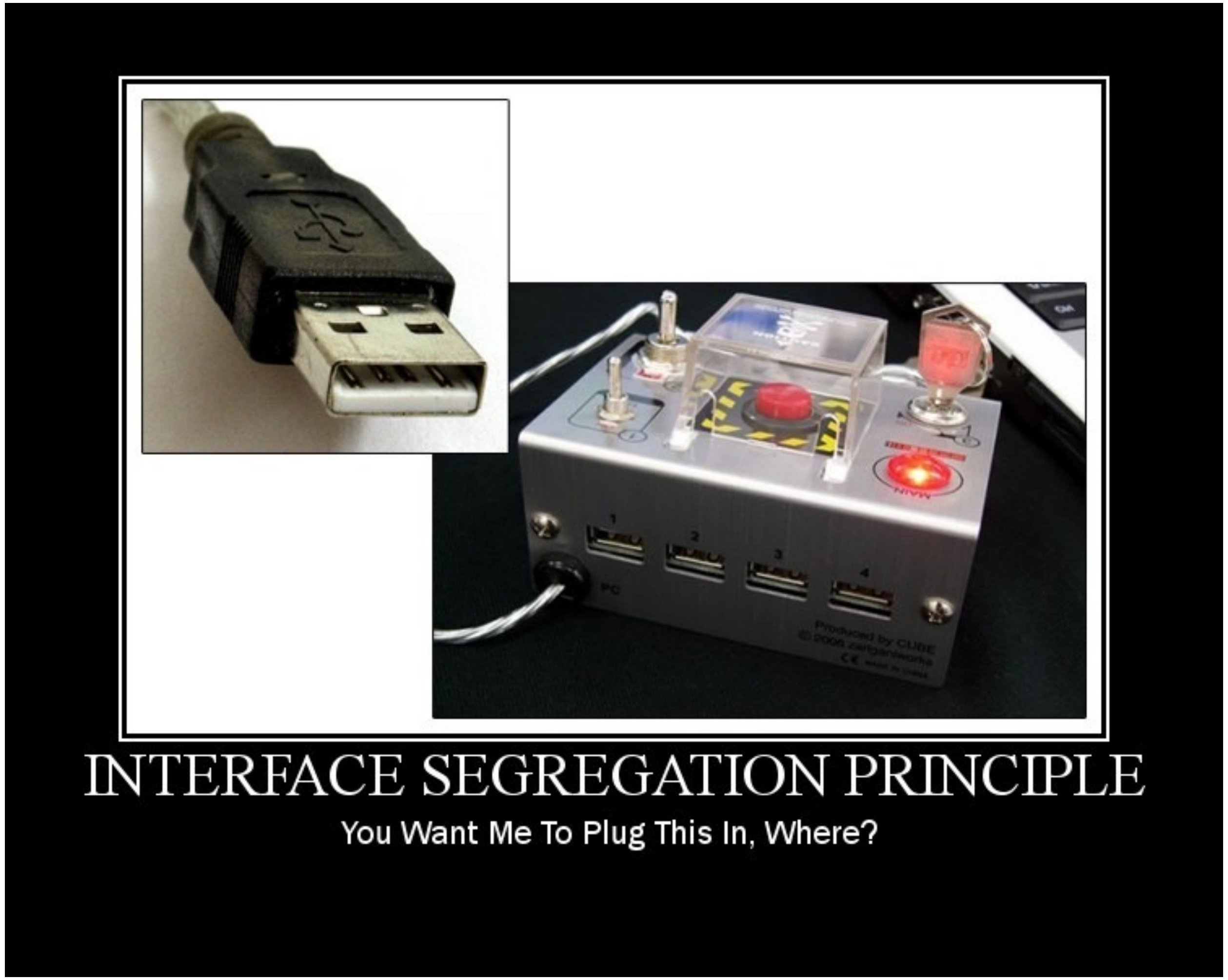

`ML4` is now depending on an interface called `ML4Server`.

Changes in `LL8` won't influence the `ML4`.

`ML4` can be reused.

`ML4` can control any object which implements the interface of `ML4Server`.

# Interface Segregation Principle (ISP)



# Interface Segregation Principle (ISP)

***Interface Segregation Principle***

Covers the drawbacks of broadly defined interfaces

Classes with non-coherent interfaces

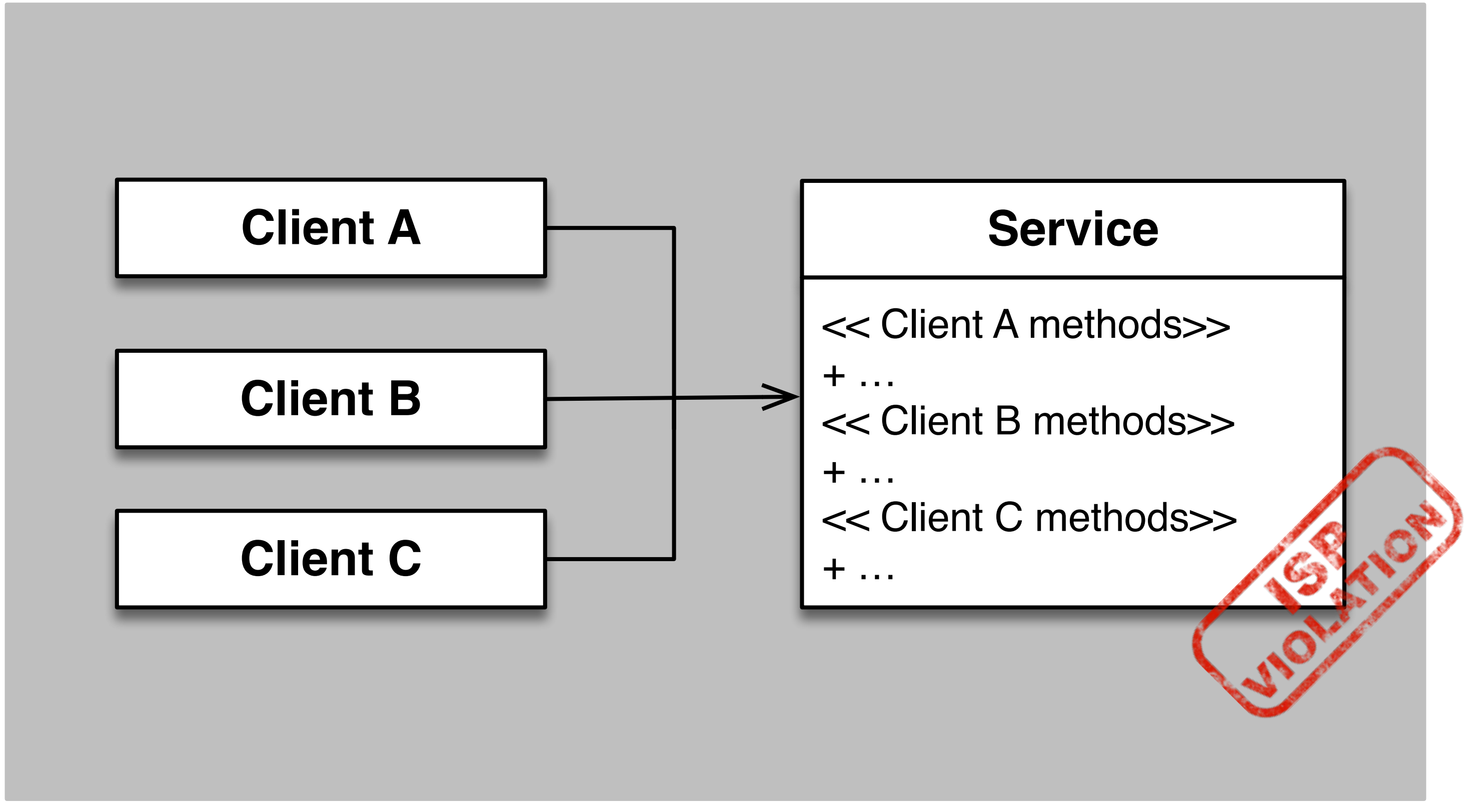
Problem:

Dependency of the calling object to methods, which it doesn‘t need to know because it isn‘t using them. Changes to an interface are concerning **every** object knowing that interface.

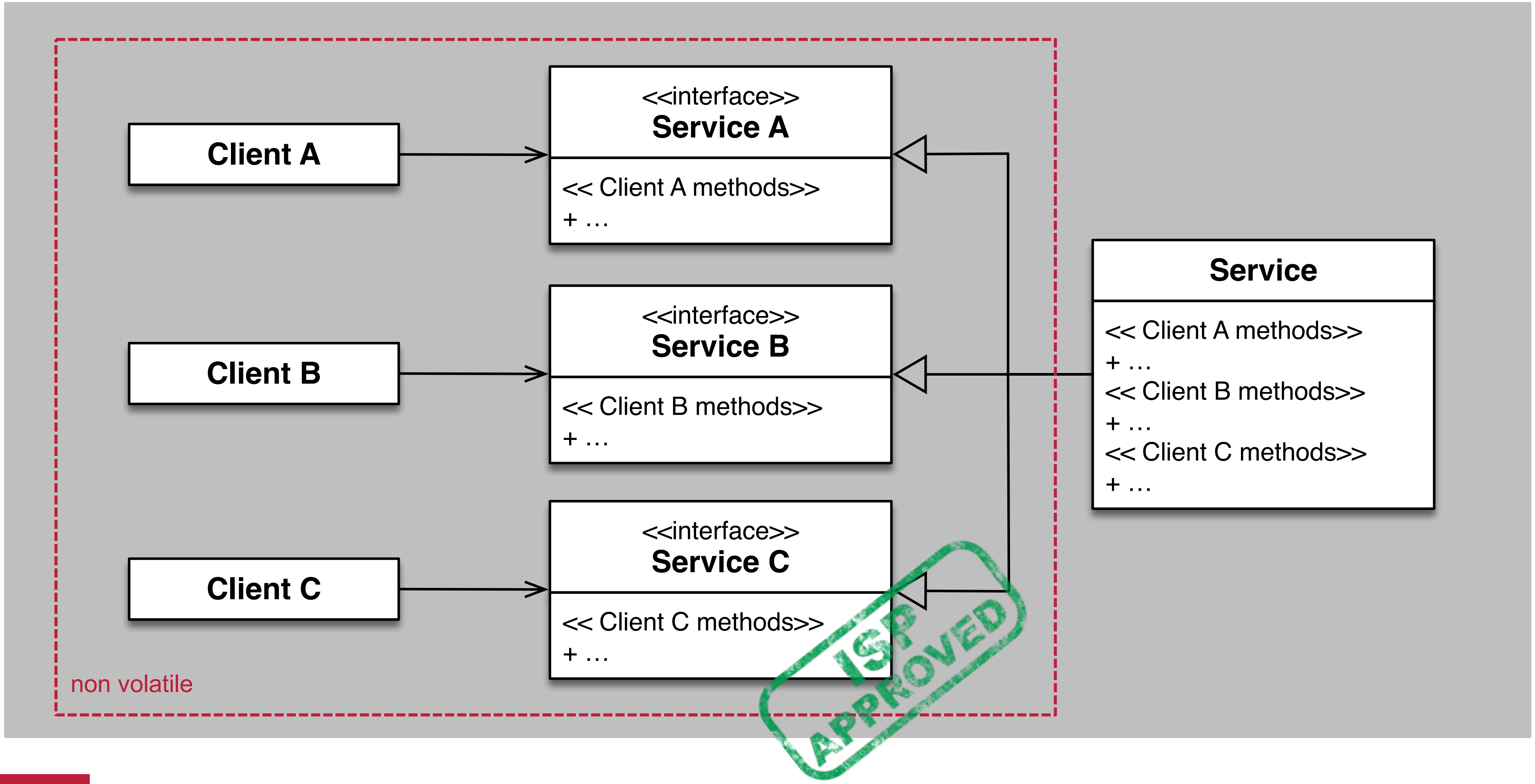
Solution:
(Coherent) Segregation of methods to multiple interfaces for the specific calling objects.

Dynamic languages aren‘t affected.

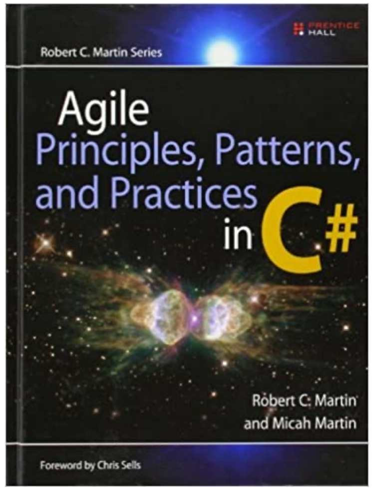
# Interface Segregation Principle (ISP) – Example 1

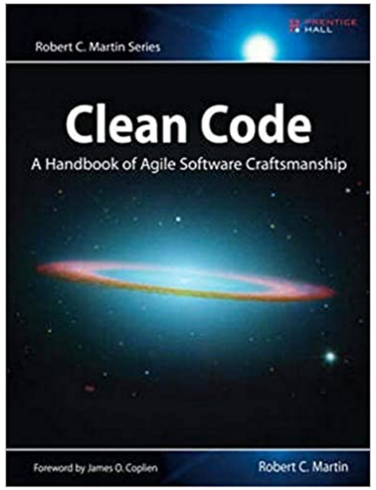


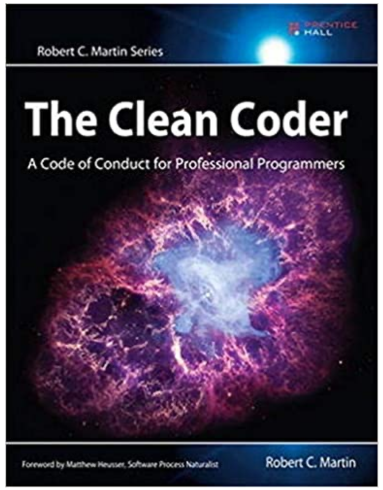
# Interface Segregation Principle (ISP) – Example 1

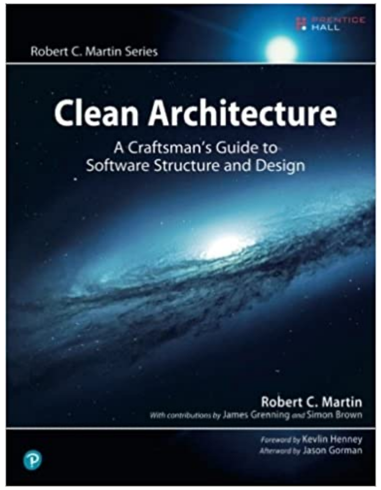


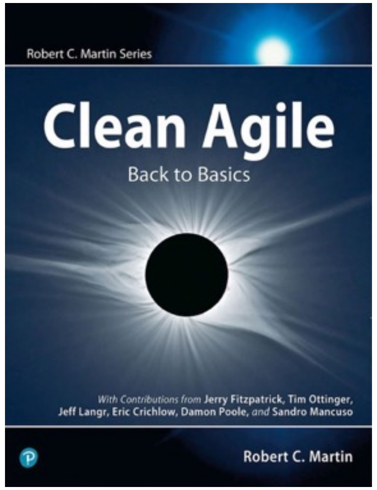
# Books

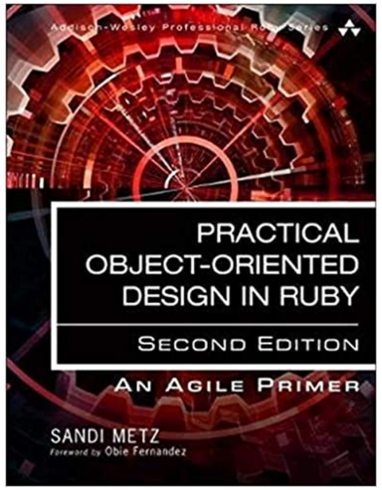

Robert C. Martin Series
Agile
Principles, Patterns, and Practices in C#
Robert C. Martin
and Micah Martin
Foreword by Chris Sells

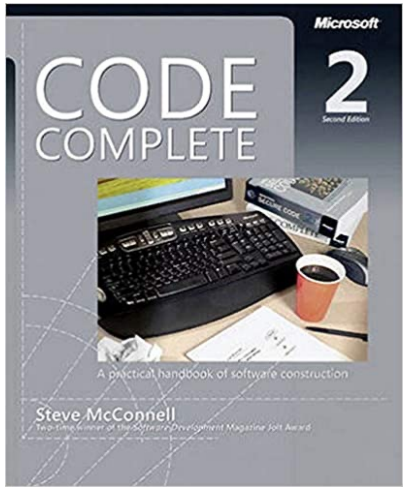

Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

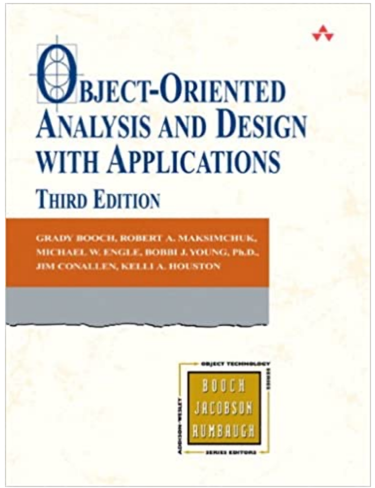

Robert C. Martin Series
The Clean Coder
A Code of Conduct for Professional Programmers
Robert C. Martin

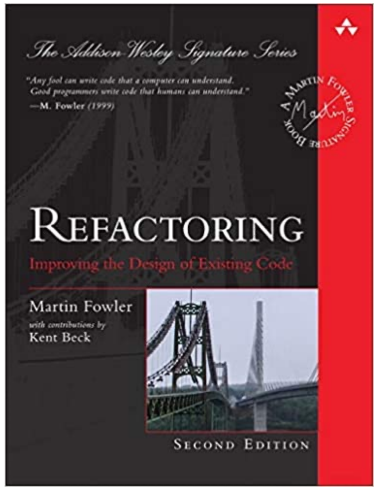

Robert C. Martin Series
Clean Architecture
A Craftsman's Guide to Software Structure and Design
Robert C. Martin

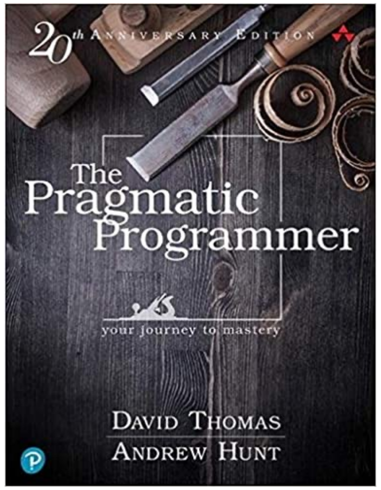

Robert C. Martin Series
Clean Agile
Back to Basics
Robert C. Martin


PRACTICAL OBJECT-ORIENTED DESIGN IN RUBY
SECOND EDITION
AN AGILE PRIMER
SANDI METZ


Microsoft
CODE COMPLETE
2
A practical handbook of software construction
Steve McConnell


OBJECT-ORIENTED ANALYSIS AND DESIGN WITH APPLICATIONS
THIRD EDITION


The Addison Wesley Signature Series
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
SECOND EDITION


20th ANNIVERSARY EDITION
The Pragmatic Programmer
your journey to mastery
DAVID THOMAS
ANDREW HUNT

# References

McCall,J.: Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisiton Manager, Bd. 1-3. General Electric, November 1977.

Meyer, B.: Object-Oriented Software Construction, Prentice Hall PTR, 1988.

McConnell, S.: Code Complete, Second Edition. Microsoft Press, Redmond, WA, USA, 2004.

Dijkstra,E.W.:The humble programmer. Commun. ACM, 15(10):859–866, 1972.

Booch, G., Maksimchuk, R.A., Engle, M.W., Young, B.J., Connallen, J. und Houston, K.A.: Object-oriented analysis and design with applications. Addison Wesley, 3. Aufl., 2007.

Yourdon, E. und Constantine, L. L.: Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Yourdon Press computing series. Prentice-Hall, Inc, Upper Saddle River, NJ, USA, 1979.

Ingalls,D.H.H.:Design Principles Behind Smalltalk. Byte,6(8):286–298,1981.

Brooks, Jr., F.P.: No Silver Bullet - Essence and Accidents of Software Engineering. Computer, 20(4):10–19, 1987.

Jack W. Reeves:, What is software design?. C++ Journal, 1992, http://www.developerdotstar.com/mag/articles/reeves_design.html

# Get in touch!

| | |
|---|---|
| website online | https://suresoft.dev |
| chat 19 user | https://matrix.to/#/#suresoft-general:matrix.org |
| Zenodo Community | https://zenodo.org/communities/suresoft/ |
| Mailinglist | https://lists.tu-braunschweig.de/sympa/info/musen-rse |
| | https://git.rz.tu-bs.de/suresoft |

# Acknowledgment

*The SURESOFT project is funded by the German Research Foundation (DFG) as part of the “e-Research Technologies” funding programme under grants: EG 404/1-1, JA 2329/7-1, KA 3171/12-1, KU 2333/17-1, LA 1403/12-1, LI 2970/1-1 and STU 530/6-1.*


Thanks!

The SURESOFT project is funded by the German Research Foundation (DFG) as part of the “e-Research Technologies” funding programme under grants: EG 404/1-1, JA 2329/7-1, KA 3171/12-1, KU 2333/17-1, LA 1403/12-1, LI 2970/1-1 and STU 530/6-1.