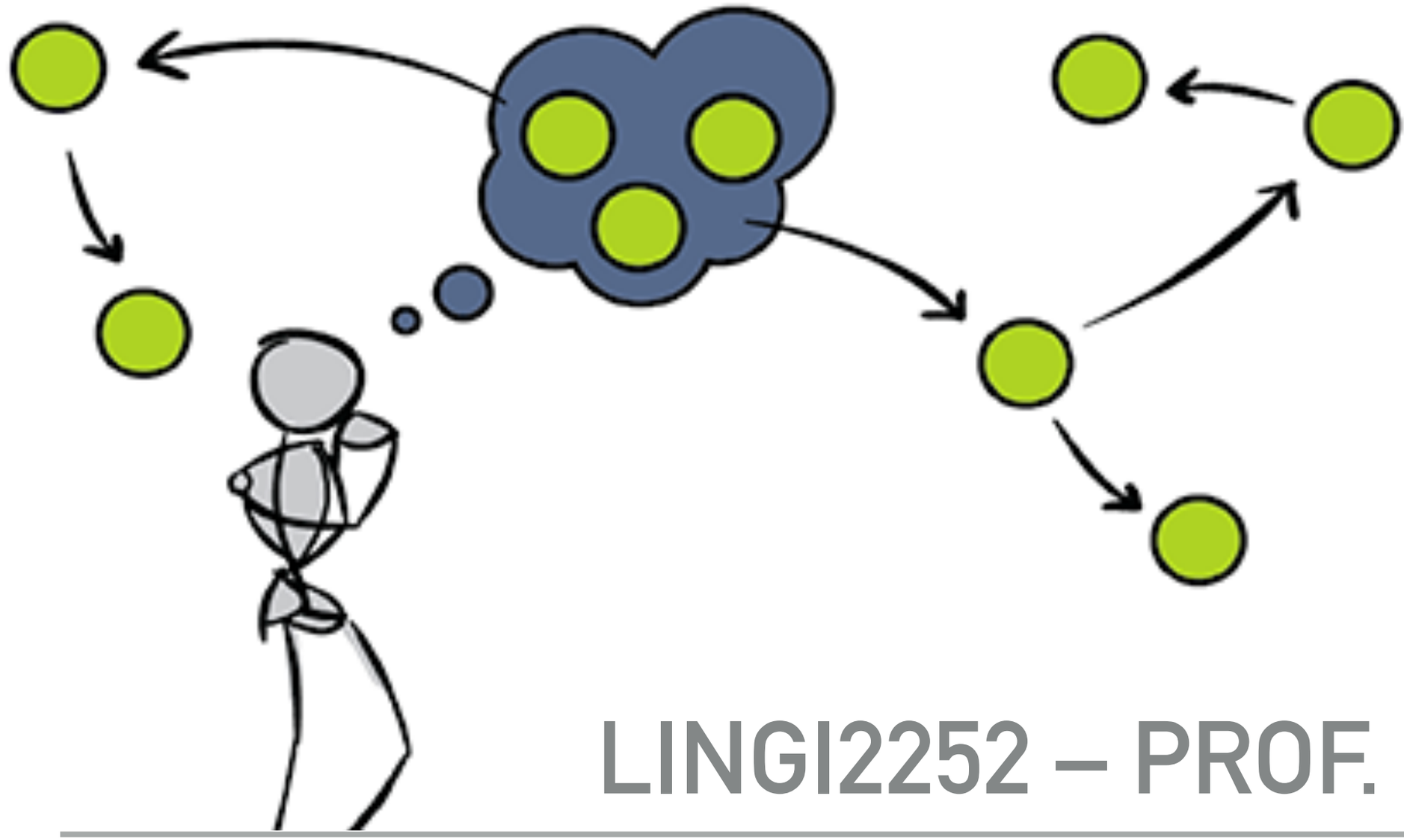
Object-Oriented Design Heuristics

Presentation · January 2017

DOI: 10.13140/RG.2.2.30387.84003

1 author:

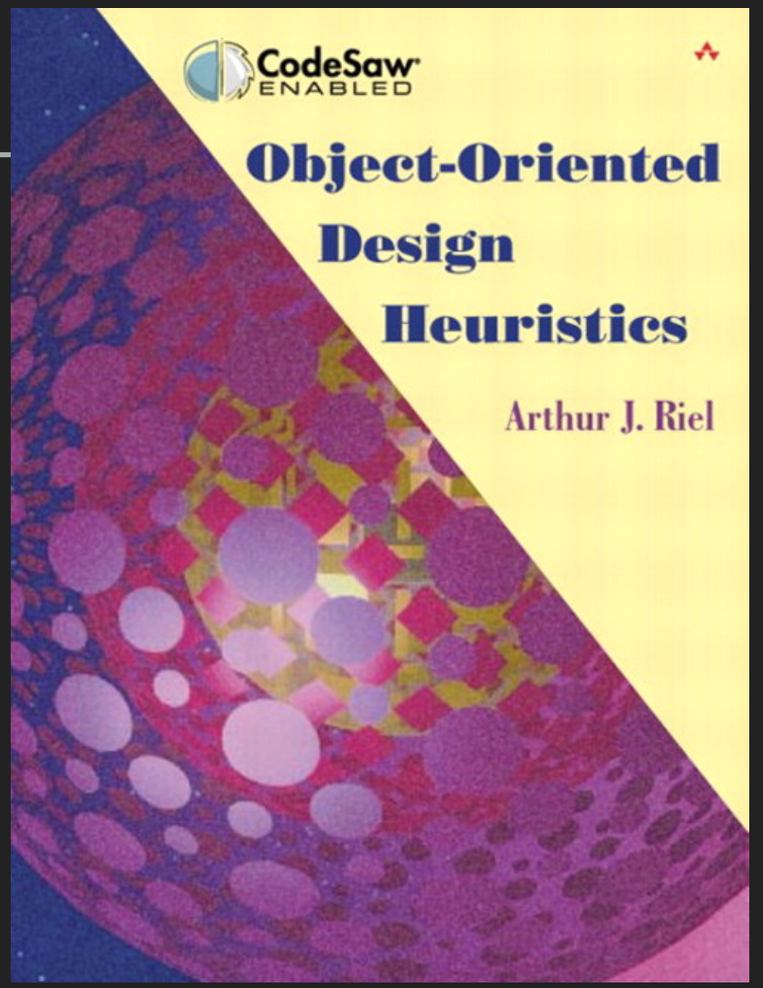
Kim Mens
Université Catholique de Louvain - UCLouvain
239 PUBLICATIONS **2,249** CITATIONS



OBJECT-ORIENTED DESIGN HEURISTICS*

* These slides are part of the course LINGI2252 “Software Maintenance and Evolution”, given by Prof. Kim Mens at UCL, Belgium

OBJECT-ORIENTED DESIGN HEURISTICS



Arthur J. Riel
Object-Oriented Design Heuristics

Other sources:

A presentation by Prof. Kenneth M. Anderson, University of Colorado Boulder, on OO Design Heuristics (April 2002)

A presentation by Prof. Harald Gall, University of Zurich, on OO Design Heuristics (2006)

A blog by Marc Hoeijmans on Riel's design heuristics (June 2012)

WHEN IS A SYSTEM IS WELL-DESIGNED?

How to know if the OO design of your system is good / bad / in between?

If you ask an OO guru : "a design is good when it feels right"

How to know when it feels right?

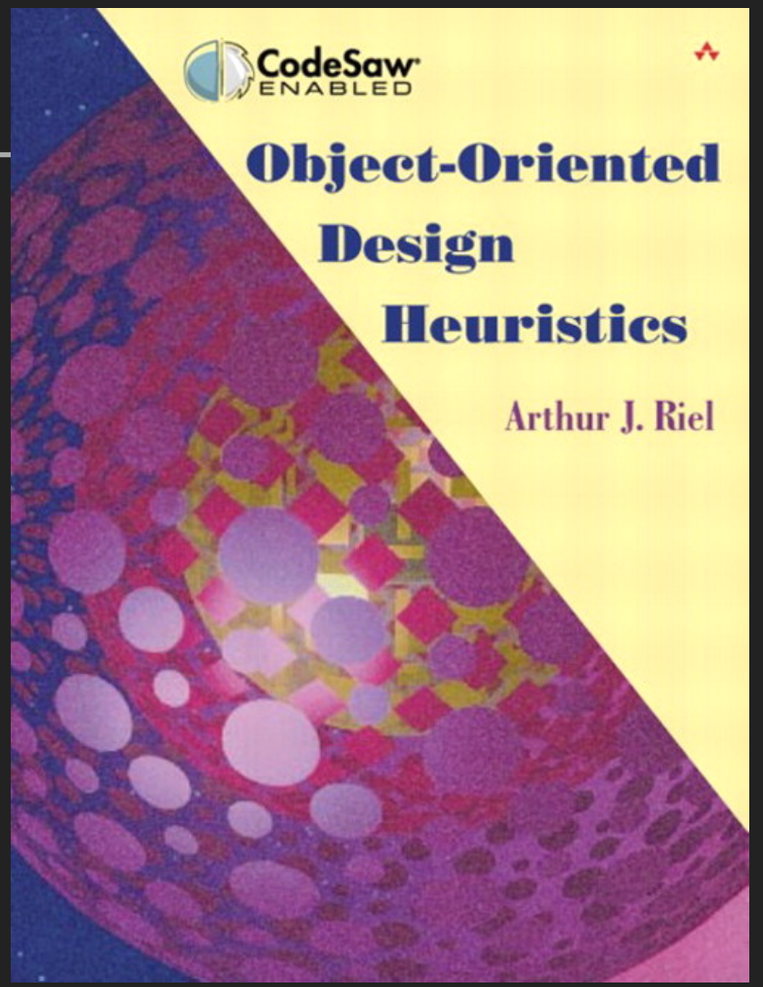
Subconsciously, a guru runs through a list of design heuristics built up through previous design experience.

If the heuristics pass, then the design feels right.

If they do not pass, the design does not feel right.

Riel's book tries to make these heuristics explicit.

OBJECT-ORIENTED DESIGN HEURISTICS



Arthur J. Riel

Object-Oriented Design Heuristics

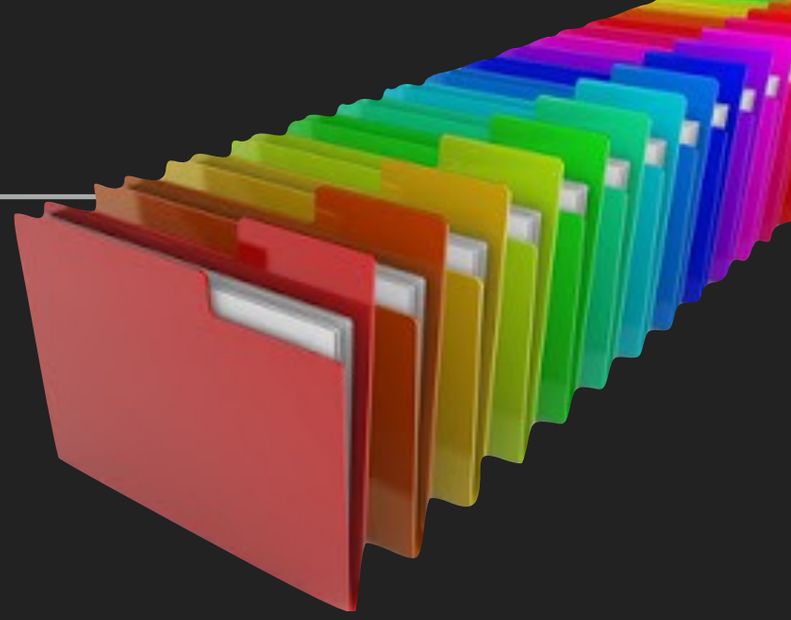
~60 language-independent guidelines

Offering insights into OO design improvement

No hard rules, only heuristics : can be ignored when not relevant

Targeted to programmers to improve their OO programming and design skills

Based on Riel's experience as CS teacher and OO A&D consultant



CATEGORIES OF OO DESIGN HEURISTICS

A. Classes and Objects

B. Topology of Procedural vs. Object-Oriented Applications

C. Relationships Between Classes and Objects

D. The Inheritance Relationship

E. Multiple Inheritance

Other Categories:

F. Association Relationship

G. Class-Specific Data and Behaviour

H. Physical object-oriented design

A WORD OF WARNING…

Not all heuristics work together perfectly.

Some may even be directly opposed !

There are always trade-offs to be made in analysis and design.

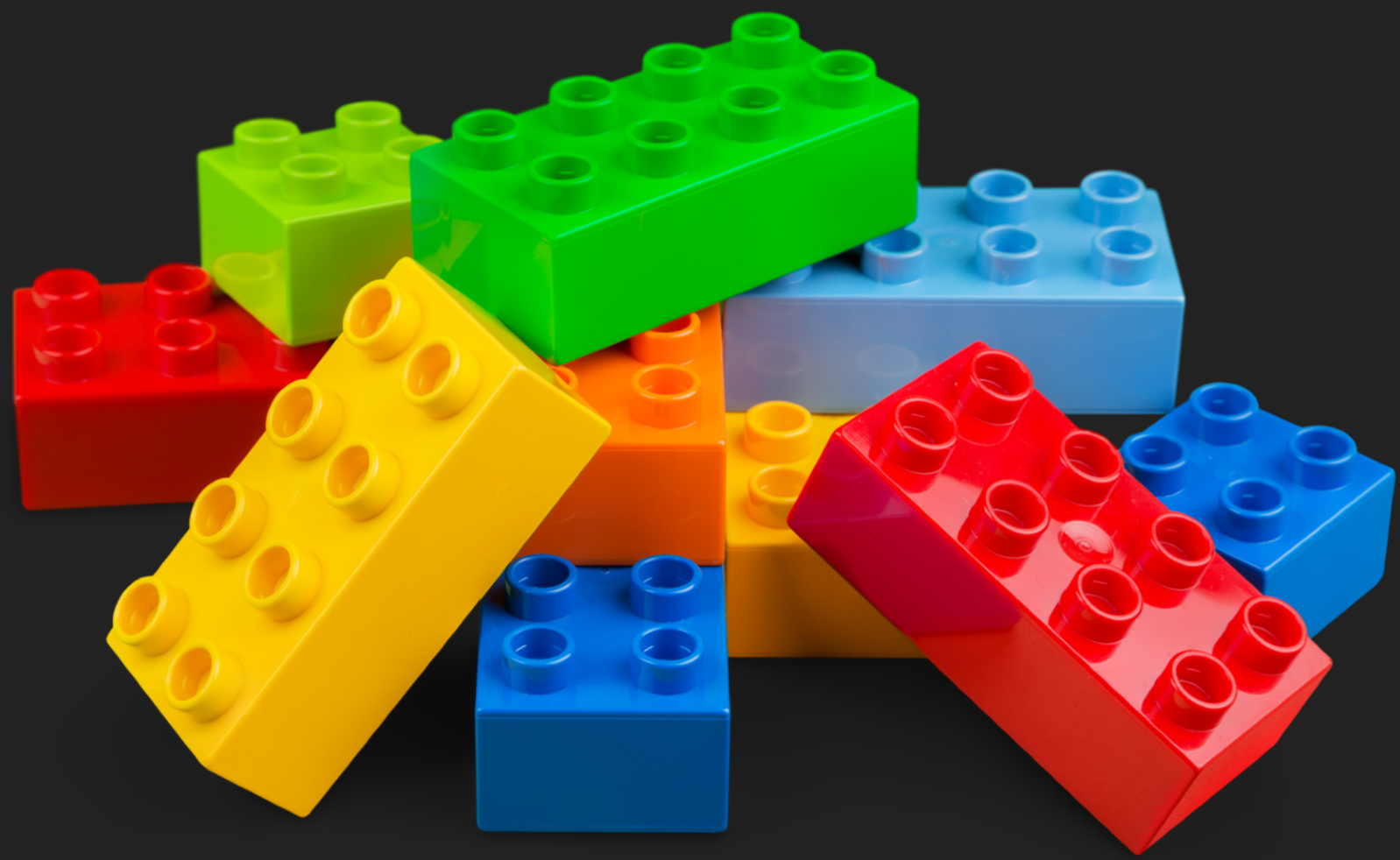
E.g., a change to reduce complexity may reduce flexibility.

You have to decide which heuristic makes most sense for your particular context.

Heuristics are not “golden” rules that always work.

A. CLASSES AND OBJECTS

The Building Blocks of the Object Oriented Paradigm



HIDING DATA

Heuristic A.1

All data should be hidden within its class.

When developers say :

"I need to make this piece of data public because..."

They should ask themselves :

"What is it that I'm trying to do with this data and why cannot the class perform that operation for me?"

Does it really have to be exposed to others?

Does this piece of data really belong in this class?

NO DEPENDENCE ON CLIENTS

Heuristic A.2

> Users of a class must be dependent on its public interface, but a class should not be dependent on its users.

Why?

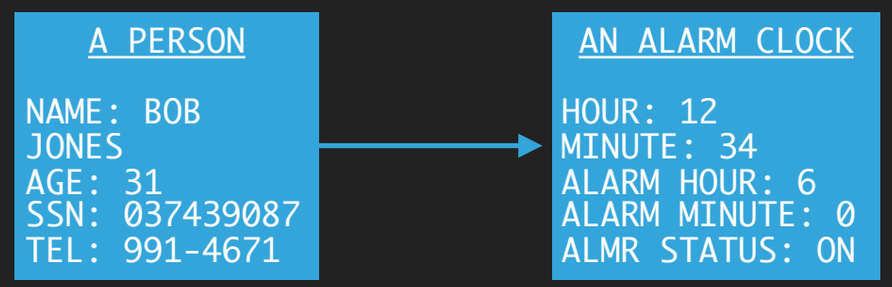
NO DEPENDENCE ON CLIENTS

Heuristic A.2

> Users of a class must be dependent on its public interface, but a class should not be dependent on its users.

Why? Reusability !

For example, a person can make use of an alarm clock, but the alarm clock shouldn't know about the person. Otherwise the alarm clock couldn't be reused for other persons.



ONE CLASS = ONE RESPONSIBILITY

A class should be **cohesive**.
Try to have **one** clear responsibility per class.

Heuristic A.8

A class should capture one and only one key abstraction.

SMALL CLASSES

Heuristic A.3

Minimise the number of messages in the protocol of a class.

(protocol of a class means the set of messages to which an instance of the class can respond)

Keep it small.

The problem with large public interfaces is that you can never find what you are looking for.

A smaller public interface make a class easier to understand and modify.

SUPPORTING POLYMORPHISM AND COMMUNICATION

Heuristic A.4

Implement a minimal public interface that all classes understand.

E.g., operations such as copy (deep versus shallow), equality testing, pretty printing, parsing from a ASCII description, etc.

Why?

To be able to send the same message to different objects.

To be able to substitute them.

CLEAR PUBLIC INTERFACE

Keep it clean: Users of a class do not want to see operations in the public interface that they are not supposed to use, cannot use, or are not interested in using.

Heuristic A.5

Do *not* put implementation details such as common-code private functions into the public interface of a class.

Heuristic A.6

Do *not* clutter the public interface of a class with items that clients are not able to use or are not interested in using.

MINIMISE CLASSES INTERDEPENDENCIES

Strive for **loose coupling** !

Heuristic A.7

A class should only use operations in the public interface of another class or have nothing to do with that class.

STRENGTHEN ENCAPSULATION

A class should be **cohesive**. Move data close to behaviour.

Heuristic A.9

Keep related data and behaviour in one place.

(Similar to the “Move Method” refactoring pattern.)

Heuristic A.10

Spin off non-related information into another class.

(Similar to the “Extract Class” refactoring pattern.)

ROLES VS. CLASSES

Heuristic A.11

Be sure the abstractions that you model are classes and not simply the roles objects play.

Are mother and father different classes or rather roles of Person?

No magic answer: depends on the domain.

Do they have different behaviour? If so, then they are more likely to be distinct classes.

B. TOPOLOGIES OF PROCEDURAL VS. OO APPLICATIONS

This category of heuristics is about the use of non-OO structures in OO applications.

Procedural topologies break an application down to functions sharing data structures.

In such a topology it is easy to see which functions access which data structures,

but difficult to see which data structures are used by which functions.

Changing a data structure may have unintended consequences on the functions using it.

Object-oriented topologies try to keep the data closer to the behaviour.

TYPICAL PROBLEMS

Two typical problems that arise when developers familiar with procedural techniques try to create an OO design:

The God Class

A single class that drives the application;
all other classes are data holders.

Proliferation of Classes

Problems with modularisation taken too far.

Too many classes that are too small in size/scope make the system hard to use, debug and maintain.

AVOID CREATING GOD CLASSES

Do not create God classes that control all other classes.

Heuristic B.12

> Distribute system intelligence horizontally as uniformly as possible, that is the top level classes in a design should share the work uniformly.

Heuristic B.13

> Do not create god classes or god objects in your system.
>
> Be very suspicious of classes whose name contains DRIVER, MANAGER, SYSTEM, SUBSYSTEM, etc.

BASIC CHECKS FOR GOD CLASS DETECTION

Heuristic B.14

Beware of classes that have many accessor methods defined in their interface.

This *may* imply that related data and behaviour are not being kept in the same place.

Heuristic B.15

Beware of classes that have too much non-communicating behaviour, that is, methods that only operate on a proper subset of their data members.

God classes often exhibit a great deal of non-communicating behaviour.

AVOID CLASS PROLIFERATION

Heuristic B.18

Eliminate irrelevant classes from your design.

Irrelevant classes often only have get, set, and print methods.

Heuristic B.19

Eliminate classes that are outside the system.

Principle of domain relevance.

Heuristic B.20

Do not turn an operation into a class.

Be suspicious of any class whose name is a verb or is derived from a verb, especially those which have only one piece of meaningful behaviour.

Ask yourself if that piece of meaningful behaviour needs to be migrated to some existing or undiscovered class.

HOW TO MODEL AN OBJECT-ORIENTED APPLICATION?

Heuristic B.17

> Model the real world whenever possible.
>
> (This heuristic is often violated for reasons of system intelligence distribution, avoidance of god classes, and the keeping of related data and behaviour in one place.)

What if you want two different UIs for the same model?

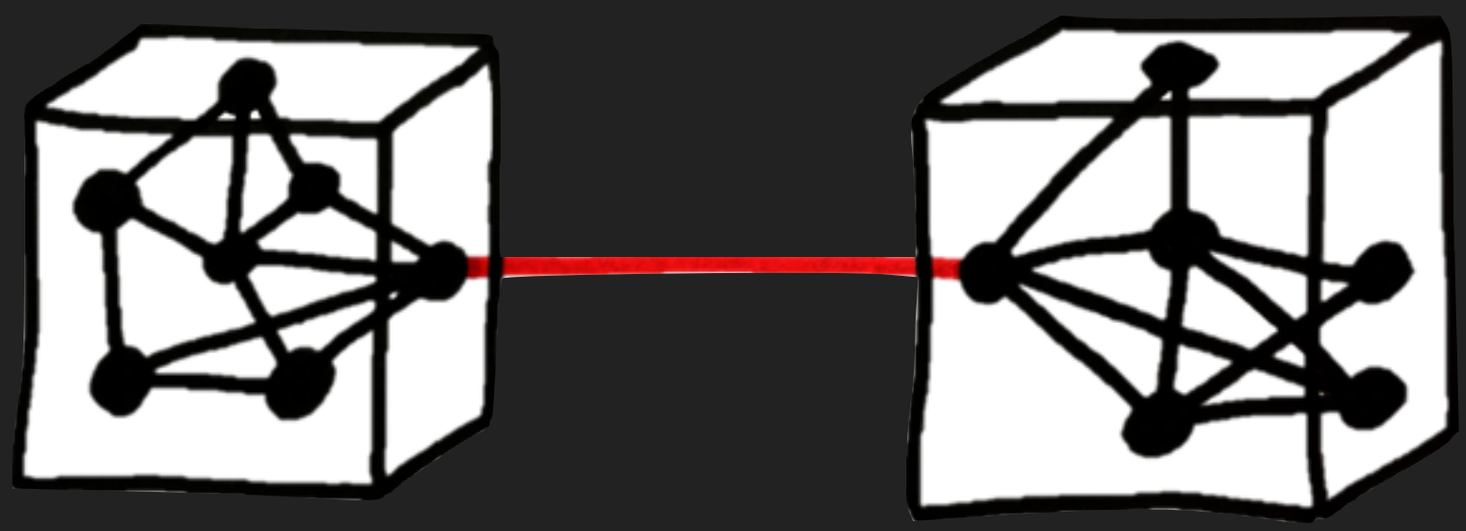
Heuristic B.16

> In applications that consist of an object oriented model interacting with a user interface, the model should never be dependent on the interface. The interface should be dependent on the model.

C. RELATIONSHIPS BETWEEN CLASSES AND OBJECTS

As a general guideline :

- High cohesion inside classes and objects
- Loose coupling between classes and objects



MINIMIZING COUPLING BETWEEN CLASSES

STRIVE FOR
LOOSE COUPLING !

Heuristic C.22

Minimize the number of classes with which another class collaborates.

Look for situations where one class communicates with a group of classes.

Ask if it is possible to replace the group by a class that contains the group.

MINIMIZING COUPLING BETWEEN CLASSES

STRIVE FOR
LOOSE COUPLING !

Related heuristics :

Heuristic C.23 : Minimize the number of message sends between a class and its collaborator.

(Counter example: Visitor design pattern)

Heuristic C.24 : Minimize the *amount of* collaboration between a class and its collaborator, that is, the *number of* different messages sent.

Heuristic C.25 : Minimize fanout in a class, that is *the product of* the number of messages defined by the class and the messages they send.

ABOUT THE USE RELATIONSHIP

When an object “uses” another one it should get a reference to it in order to interact with it

Ways to get references :

(containment) contains instance variables of the class of the other object

the other object is passed as argument

asked to a third party object (mapping...)

instance creation of the other object and then interact with it

CONTAINMENT AND USES

Heuristic C.26

If a class contains objects of another class, then the containing class should be sending messages to the contained objects

that is, the containment relationship should always imply a “uses” relationship.

Heuristic C.34

A class must know what it contains, but should never know who contains it.

(Do not depend on your users.)

COHERENCE IN CLASSES

STRIVE FOR
HIGH COHESION !

Heuristic C.27 :

Most of the methods defined on a class should be using most of the data members most of the time.

A class should be **cohesive**.

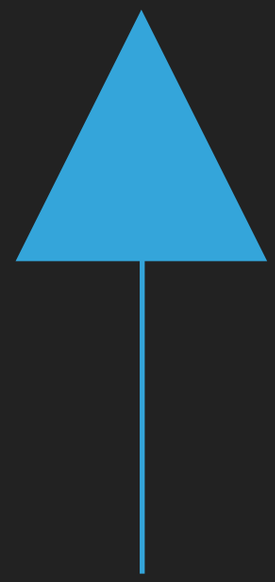
Heuristic C.28 :

Classes should not contain more objects than a developer can fit in his or her short-term memory. A favourite value for this number is six (or seven).

Heuristic C.29 :

Distribute system intelligence vertically down narrow and deep containment hierarchies.

D. THE INHERITANCE RELATIONSHIP



INHERITANCE DEPTH

Heuristic D.39

In theory, inheritance hierarchies should be deep - the deeper the better

Heuristic D.40

In practice, however, inheritance hierarchies should be no deeper than an average person can keep in his or her short term memory. A popular value for this is six.

ABSTRACT CLASSES = BASE CLASSES

Heuristic D.41.

> All abstract classes must be base classes.

Heuristic D.42 :

> All base classes must be abstract classes.

Base class = class with at least one subclass

Abstract class = class with at least one abstract method

These heuristics are controversial !

ABSTRACT CLASSES = BASE CLASSES : CONTROVERSIAL !

Heuristic D.41 : All abstract classes must be base classes.

Intuition : why make a method abstract if you won't concretise it in a subclass?

Counter-example : application frameworks

Heuristic D.42 : All base classes must be abstract classes.

Intuition : methods overridden in the subclasses should be abstract in the superclass

Not necessarily true :

they can have a default behaviour or value in the superclass;

the subclass may add only new methods

BASE CLASSES IN INHERITANCE HIERARCHIES

Heuristic D.37

Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes.

A superclass should not know its subclasses.

Heuristic D.38

All data in a base class should be private; do not use protected data.

Subclasses should not use directly data of superclasses.

Heuristic D.51

It should be illegal for a derived class to override a base class method with a NOP method, that is, a method that does nothing.

COMMONALITIES IN INHERITANCE HIERARCHIES

Heuristic D.43

Factor the commonality of data, behaviour, and/or interface, as high as possible in the inheritance hierarchy.

Heuristic D.45

If two or more classes have common data (variables) and behaviour (that is, methods), then those classes should each inherit from a common base class that captures those data and methods.

COMMONALITIES IN INHERITANCE HIERARCHIES

Heuristic D.44

> If two or more classes share only common data (no common behaviour), then that common data should be placed in a class that will be contained by each sharing class.

Heuristic D.44.bis

> If two or more classes share only common interface (i.e. messages, not methods) then they should inherit from a common base class only if they will be used polymorphically.

AVOID TYPE CHECKS (ESSENTIAL !)

Heuristic D.46

Explicit case analysis on the type of an object is usually an error.

or at least bad design : the designer should *use* *polymorphism* instead in most of these cases

indeed, an object should be responsible of deciding how to answer to a message

a client should just send messages and not discriminate messages sent based on receiver type

AVOID CASE CHECKS

Heuristic D.47

Explicit case analysis on the value of an attribute is often an error.

The class should be decomposed into an inheritance hierarchy, where each value of the attribute is transformed into a derived class.

INHERITANCE = SPECIALISATION

Heuristic D.36

Inheritance should be used only to model a specialisation hierarchy.

Do not confuse inheritance and containment.

Containment is black-box.
Inheritance is white-box.

Heuristic D.52

Do not confuse optional containment with the need for inheritance.

Modelling optional containment with inheritance will lead to a proliferation of classes.

DYNAMIC SEMANTICS

Heuristic D.48

Do not model the dynamic semantics of a class through the use of an inheritance relationship.

An attempt to model dynamic semantics with a static semantic relationship will lead to a toggling of types at run time.

Heuristic D.49

Do not turn objects of a class into derived classes of the class.

Be very suspicious of any derived class for which there is only one instance.

Heuristic D.50

If you think you need to create new classes at run time, take a step back and realise that what you are trying to create are objects. Now generalise these objects into a new class.

FRAMEWORKS

Heuristic D.53

> When building an inheritance hierarchy, try to construct reusable frameworks rather than reusable components.

E. MULTIPLE INHERITANCE

PROVE MULTIPLE INHERITANCE

Avoid using multiple inheritance when possible. (It's too easy to misuse it).

Heuristic E.54

> If you have an example of multiple inheritance in your design, assume you have made a mistake and then prove otherwise.

Most common mistake: Using multiple inheritance in place of containment

QUESTION IT

Heuristic E.55

Whenever there is inheritance in an OO design, ask yourself two questions:

(a) Am I a special type of the thing from which I am inheriting?

(b) Is the thing from which I am inheriting part of me?

A yes to (a) and no to (b) would imply the need for inheritance.

A no to (a) and a yes to (b) would imply the need for composition instead?

– Is an airplane a special type of fuselage? No (the fuselage is the body of an airplane)

– Is a fuselage part of an airplane? Yes

QUESTION IT

Heuristic E.56

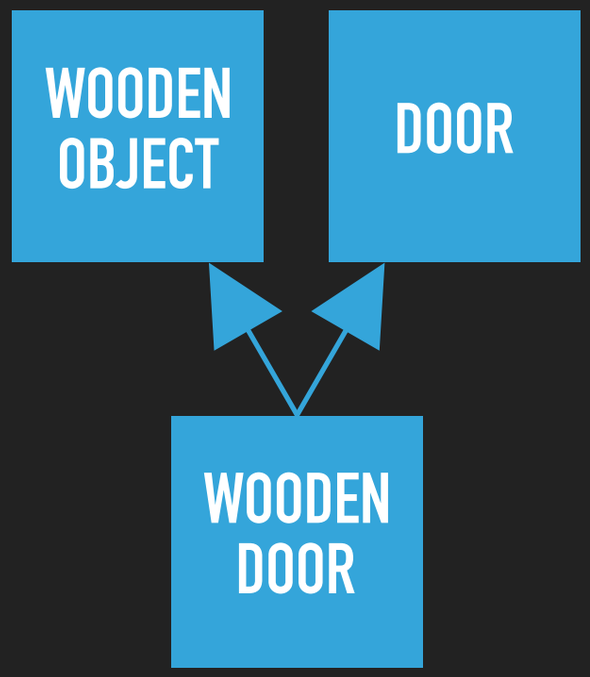
> Whenever you have found a multiple inheritance relationship in an OO design, be sure that no base class is actually a derived class of another base class.

i.e. *accidental* multiple inheritance.

MULTIPLE INHERITANCE

So, is there a valid use of multiple inheritance?

- Yes, subtyping for combination



When defining a new class that is a special type of two other classes and those two base classes are from different domains

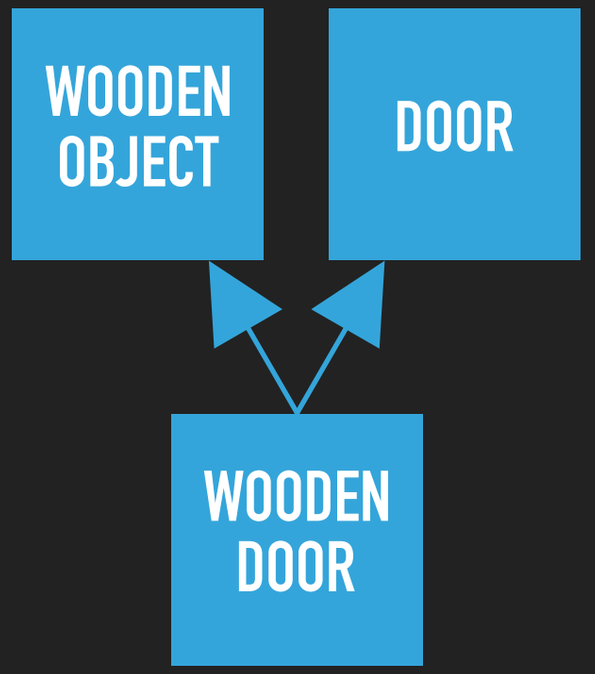
MULTIPLE INHERITANCE

Is a wooden door a special type of door? Yes
Is a door part of a wooden door? No

Is a wooden door a special type of wooden object? Yes
Is a wooden object part of a door? No

Is a wooden object a special type of door? No
Is a door a special type of wooden object? No

All Heuristics Pass!



OTHER CATEGORIES OF OBJECT-ORIENTED DESIGN HEURISTICS

F. The Association Relationship

G. Class-Specific Data and Behaviour

H. Physical object-oriented design

F. THE ASSOCIATION RELATIONSHIP

Heuristic F.57 : Containment or Association?

When given a choice in an OO design between a containment and association, choose the containment relationship.

G. CLASS-SPECIFIC DATA AND BEHAVIOUR

Heuristic G.58 : No global bookkeeping

Do not use global data or functions to perform bookkeeping information on the objects of a class. Class variables or methods should be used instead.

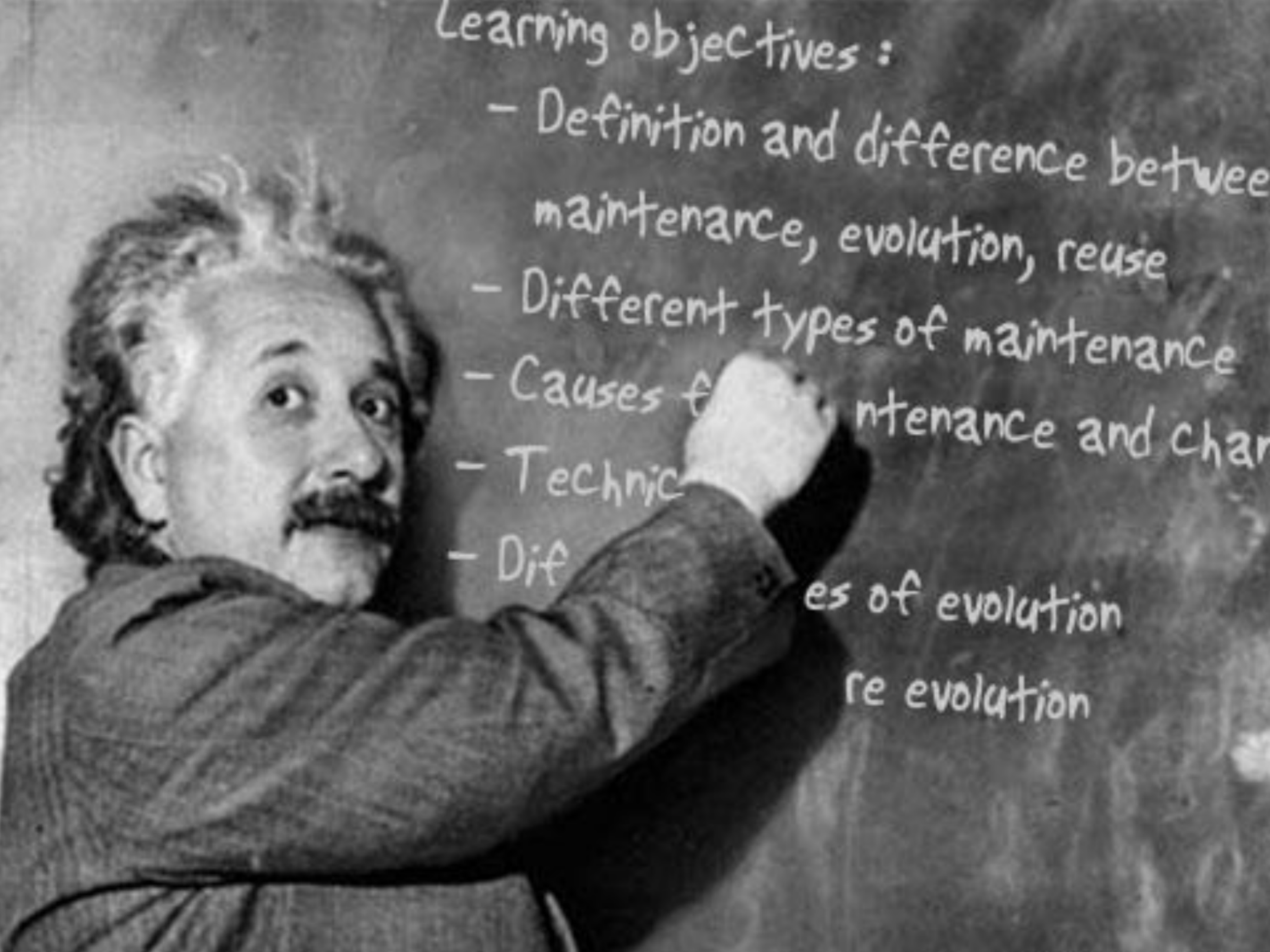
H. PHYSICAL OBJECT-ORIENTED DESIGN

Heuristic H.59

OO designers should not allow physical design criteria to corrupt their logical designs. However, physical design criteria often are used in the decision-making process at logical design time.

Heuristic H.60

Do not change the state of an object without going through its public interface.


Learning objectives :
- Definition and difference betwee
maintenance, evolution, reuse
- Different types of maintenance
- Causes f ntenance and char
- Technic es of evolution
- Dif re evolution

SUMMARY

Use the guidelines for :

- insightful analysis
- critical reviews
- as guide for better OO design
- to build reusable components and frameworks

POSSIBLE QUESTIONS

- Give and explain at least 2 design heuristics about the relation between a subclass and its superclass.
- Discuss the design heuristics which state that "All abstract classes must be base classes" and "All base classes should be abstract classes". Do you agree with these heuristics? Under what conditions?
- Several design heuristics are related to the need for high cohesion. Discuss 2 such heuristics and their relation with cohesion.
- Several design heuristics are related to the need for loose coupling. Discuss 2 such heuristics and their relation with coupling.
- Discuss the following design heuristic "Explicit case analysis on the type of an object is usually an error."
- Give a concrete example of and discuss when multiple inheritance would be a valid design solution.