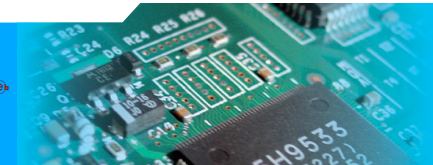


Buts et règles de la conception par objets

Noël PLOUZEAU

IRISA/ISTIC



Plan

- ④ Définition des critères de qualité
- ④ Principes généraux qui guident les décisions de conception
 - ④ Stratégies de conception
 - ④ Organisation des travaux de conception



Quality criteria

☉ Definitions from the **ISO 9126 standard**

☉ Functionality

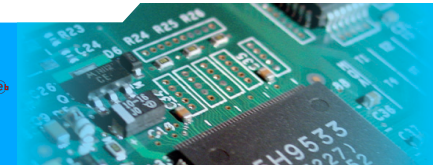
☉ Usability

☉ Maintainability

☉ Efficiency

☉ Reliability

☉ Portability



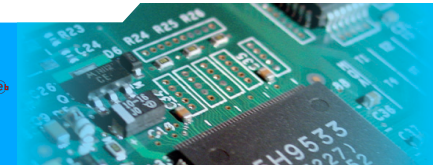
Main quality criteria for this lecture

⦿ Functionality

- ⦿ Does the system work as specified in the requirements?

⦿ Robustness

- ⦿ How does the system react to unexpected events or data?
- ⦿ This includes some parts of cybersecurity (resistance to attacks)



Main quality criteria for this lecture

- © Maintainability
 - © How difficult is testing, failure diagnosis, bug correction?
- © Reusability
 - © How difficult is the adaptation of the system to new requirements, to new deployment platforms?



A definition from Uncle Bob

- © The primary purpose of architecture is to **support the life cycle** of the system. Good architecture makes the system easy to understand, easy to develop, easy to maintain, and easy to deploy. The ultimate goal is to **minimise the lifetime cost** of the system and to **maximise programmer productivity**.
- © *From Martin, Robert C. Clean Architecture (Robert C. Martin Series) (p. 137). Pearson Education. Kindle Edition.*



Some techniques to help

© This lecture will present some popular techniques that help to get good quality

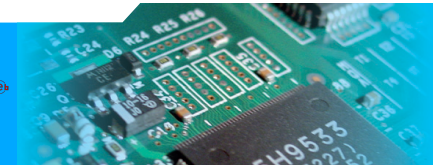
© SOLID

© DRY

© YAGNI

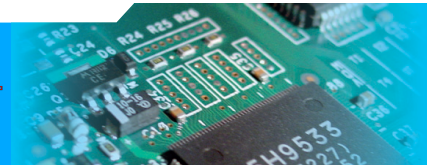
© KISS

© CBD



SOLID

- Proposed by “Uncle Bob” (Robert C. Martin)
- This acronym stands for
 - **S**ingle responsibility
 - **O**pen/closed principle
 - **L**iskov substitution principle
 - **I**nterface segregation
 - **D**ependency inversion



Single responsibility

- A class must address one concern, not several
- A given problem is solved by the cooperation of a group of classes
 - Often described by a preexisting design pattern
- This promotes modularity and reusability
- Corresponding bad smell: the megaclass



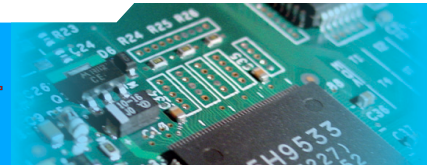
Open/closed principle (B. Meyer)

- Open for extension
 - Helps reusability
 - Extend by inheritance (carefully!)
 - Extend by delegation, composition
- Closed for modification
 - Do not modify the existing class source to suit your extension or adaptation needs



Liskov substitution principle

- Defined by Barbara Liskov, a major researcher in software engineering
- Any object of a given type T can be replaced with another object of type U when U is a subtype of T
 - interface U extends interface T
 - or subclass U extends class T
 - or class U implements interface T



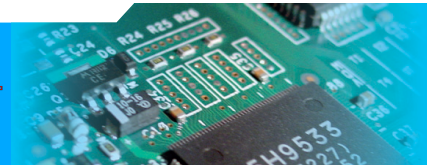
Liskov substitution principle

- More precisely
 - contravariance on inputs: a subtype U must deal with the constraints dealt by T , as a minimum
 - operation parameter types
 - preconditions on operations
 - covariance on outputs: the results produced by U cannot be super types of the results of T
 - return value types
 - postconditions



Example of Liskov substitution

```
© interface Printing {  
    © public void printDocument(Document d);  
    © }  
    © class PostscriptPrinter implements Printing {  
        © @Override  
        © public void printDocument(Document d);  
    © }
```



Example of Liskov substitution

- interface SignedDocument extends Document {...}
- Printing printer = new PostscriptPrinter();
- SignedDocument signedDocument = ...;
- Document doc = signedDocument;



To sum up Liskov's substitution principle

- Any derived (ie child) type (ie interface or class) can be used transparently in place of the base (ie mother) class
- Thus a child class is not authorised to have constraints on the its environment that the base class does not have
- Famous example of bad application
 - a derived Square class that extends a Rectangle class
 - on a square a change of width changes the height also => there is a dependency that is hidden if the square is substituted for a rectangle



Interface segregation

- An object should depend on operations it needs only
- A way to ensure this principle is defining classes that implement a set of several interfaces
- Objects of this class are seen as implementing only a subset of this interface set



Example of interface segregation

- interface Printing {...}
- interface Accounting {
 - public void setPageQuota(User user, int newQuota);
- }
- class PostscriptPrinter implements Printing, Accounting { ...}
- Printing printer = new PostscriptPrinter();
- // printer does not need Accounting does not depend on it



Dependency inversion principle

- ④ “Details should depend on abstractions. Abstractions should not depend on details” (Robert C. Martin)
- ④ No implementation should depend on other implementations: “High level modules should not depend on low level modules. Both should depend on abstractions” (Robert C. Martin)
- ④ Therefore all implementations should depend on abstractions of implementations, *ie* interfaces in Java or C#, pure virtual classes in C++



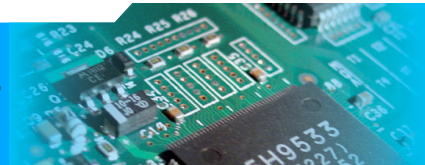
Example of inversion principle

```
Ⓢ class WordProcessor {  
    Ⓢ private Printing printerToUse;  
    Ⓢ WordProcessor(Printing printer) {  
        Ⓢ printerToUse = printer;  
    Ⓢ }  
    Ⓢ // This class knows nothing about implementations of Printing  
    Ⓢ // There is no call to constructors, the dependency is passed in the  
        constructor of WordProcessor  
    Ⓢ }
```



Resolution of dependencies

- To get a reference to an object, you can
 - instantiate it using a call to new
 - but... **violation of the dependency inversion principle**, your code depends on a class, ie a detail
 - get the reference through a call to a registry (factory)
 - good way of hiding the details , ie the class of the object
 - let a dependency injection framework create and set the reference



Example of dependency resolution

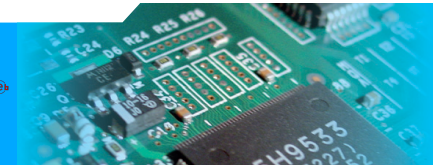
• Old style:

• `public class WordProcessor {`

• `private Printing printer = new PostscriptPrinter();`

• `}`

• `// What should I do if I want to change the printer implementation class?`



Example of dependency resolution

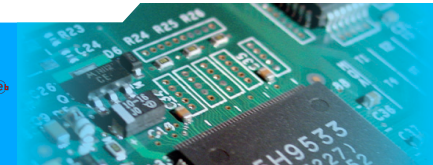
Registry style:

```
import fr.istic.aco.PrinterFactory;
```

```
public class WordProcessor {
```

```
    private Printing printer =  
        PrinterFactory.getPrinter();
```

```
}
```



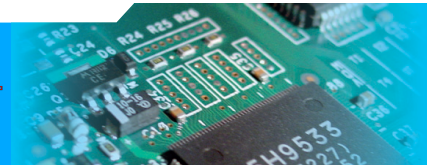
Example of dependency resolution

- Dependency injection style:
 - public class WordProcessor {
 - private Printing printer;
 - WordProcessor(Printer printer) {
 - this.printer = printer;
 - }
-



Architecture description

- In an external configuration file (eg Spring):
 - `<beans>`
 - `<bean name=« wordProcessor"
class=« WordProcessor">`
 - `<constructor-arg><ref bean="aPrinter" /></
constructor-arg>`
 - `</bean>`



(continued)

- `<bean name="aPrinter" class="PostscriptPrinter">`

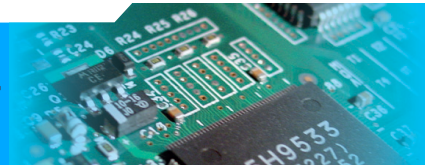
- `</bean>`

- `</beans>`

- At the launch of the application, a dependency resolution engine

- reads the XML file

- instantiates the objects and interconnects them



The DRY principle

- ⦿ Don't repeat yourself
 - ⦿ Never duplicate pieces of code
 - ⦿ Otherwise in the best case you will have to repeat yourself when updating the code
 - ⦿ In most cases you will miss or forget one of the copies
- ⦿ Use internal utility operations and methods to write a piece of code once only



Example of DRY application

```
• class Counter implements Subject {  
    • public void setValue(int v) {  
        • value = v;  
        • notifyObservers();  
    • }  
}
```



Example of DRY application

```
© public void clear() {  
    © value = 0;  
    © notifyObservers();  
    © }  
  
© private void notifyObservers() { ... }  
  
© } // class Counter
```



The KISS and YAGNI principles

- 🕒 KISS = Keep It Simple, Stupid
 - 🕒 Make simple tasks simple to do
 - 🕒 Cut tasks into simple tasks, easy to do and connect
- 🕒 YAGNI = You Ain't Gonna Need It
 - 🕒 Prefer extensibility and adaptability to complexity and useless generality



The CBD technique (B. Meyer)

- CBD means contract-based design, proposed by Bertrand Meyer
- It increases the precision of service operation definition
- Predicates known as preconditions and postconditions complement an operation's constraints on the parameter types and the return type



Preconditions on an operation

- An operation call is allowed only if **all** preconditions evaluate to true
- If an operation is called and **some** of its preconditions evaluate to false
 - the call fails
 - and it is the **caller's** fault, not the called method's implementer fault



Postconditions

- © If **all** preconditions are true then
 - © execution of the operation's method must bring the object to a state where **all** postconditions are true
 - © if **some** preconditions are false then the method fails and it the method implementer's fault



Why the name Design by contract?

- Because for each operation the set of preconditions and postconditions must be **explicit**
- Contract stake holders: caller and callee
- The caller's part in the contract states which preconditions the caller must ensure before calling the operation
- The callee's part states which postconditions the callee must ensure **if** all the preconditions are satisfied



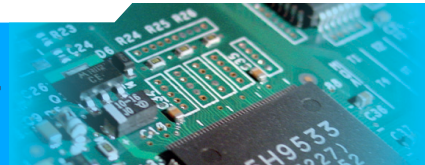
Why the name Design by contract?

- ⦿ A contract is bound to an operation
- ⦿ The pieces of code that must obey the contract are methods
 - ⦿ the method that calls the operation (caller's code)
 - ⦿ the method that implements the operation (callee's code)



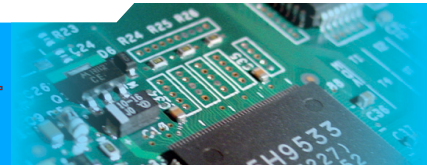
Relations between the LSP and CBD

- The LSP says that a subtype S cannot require more constraints than the base type B
 - Preconditions of S 's operations can be weaker, but not stronger
- A subtype cannot provide a result with less constraints than the base type
 - Postconditions of S 's operations can be stronger, but not weaker



Example (in Java)

```
© interface I {  
    © // pre : x > 0  
    © // post : result >= 0  
    © public int compute(int x);  
    © }  
    ©
```



Example (in Java)

```
• class GoodAlgorithm implements I {  
    • @Override  
    • // pre : true  
    • // post :  $x > 1$   
    • public int compute(int x)  
    • }
```



Example (in Java)

- class **BadAlgorithm** implements I {
 - @Override
 - // pre : $x > 1$ // INVALID: stronger req than base type
 - // post : true // INVALID: weaker result than base type
 - public int compute(int x)
- }

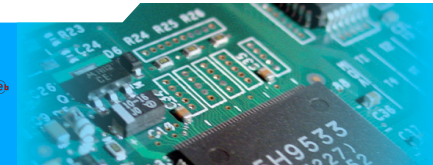


Another example of CBD

- The Stack example (on blackboard)
 - Cf ACO2018/Stack on gitlab
 - Clear contract in the Javadoc



Bad smells and antipatterns



Common general problems of a design

- © What happens when principles are not followed?
- © The problems are the dark side of quality attributes
 - © **Rigidity** from fear of breaking the system when modifying it
 - © **Fragility** because testing is not automated, there are too many internal dependencies
 - © **Viscosity**: extension by juxtaposition because the changes are too difficult to do



What are bad smells?

- Often when you look at a given architecture you see problematic decisions that prevent quality: bad smells
- Bad smells can be found in design, in code, in documentation, in tests,...
- The cause might be lack of knowledge and training, haste, blind copy paste (hello stackoverflow ;), legacy constraints, etc
- Refactoring can remove bad smells, but it is costly (hence the expression “technical debt”)



Bad smell: Duplicated code

- It comes from not applying the DRY principle
 - Copy a part of code here and paste it elsewhere
 - Usually not too difficult to remove using private methods
- Sometimes the duplicates are different but conceptually do the same thing: create a method with parameters



Warning: do not blindly eliminate duplicates

- © If you have two pieces of software that look the same, check whether this is
 - © essential: they are bound to evolve in the same way in the future; eliminate duplication
 - © accidental: they can evolve differently in the future; do not eliminate duplicates, doing so will bring problems in the future



Bad smell: Long method

- Sign of unnecessary complexity and limited reuse
- Terrible if uncommented, still bad if commented
- Split into a group of cooperating methods
- Better yet: dispatch these methods in cooperating objects (define a *collaboration*)
- See the relation with the *Long comments* bad smell



Bad smell: Mega class

- When a given class has many responsibilities and a lots of code
- Often produces by copy pasting non object oriented code (frequently found in C++, since C is a subset of C++)
- Solution
 - Redistribute the responsibilities using several classes



Bad smell: Long parameter list

- Ⓢ Not specific to OO code
 - Ⓢ I have used libraries with about 10 parameters for many operations
- Ⓢ Solution: define data structure to store parameters
- Ⓢ We will study the Builder design pattern



Bad smell: Shotgun surgery

- One functional change implies many modifications in many classes
- Sign of too many dependencies between objects/classes
- Solution : Extract the small parts in the classes to build a new class where all the parts are grouped together



Bad smell: Feature envy

- When a piece of code in a given class A is mostly just calls to code of another class B (rather than calls to other class A methods or access to A's attributes)
- Not much point in having this method in class A, move it to B completely (or at least the part that concerns B)



Bad smells: Data clumps

- When computations often use together pieces of data that are in various distinct classes
- Solution: put the parts that work together into the same class



Bad smell: Switch statements

- ⦿ Instead of relying of polymorphism of object-oriented languages, the code contains an explicit switch statement
- ⦿ Very clumsy way of coding variants of behaviour
- ⦿ Produces code that is fragile and difficult to extend
- ⦿ Solution: use subtypes and polymorphism
- ⦿ Let us take an example

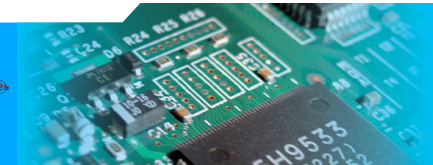


Example: stinky code

© See the repository on ISTIC's gitlab

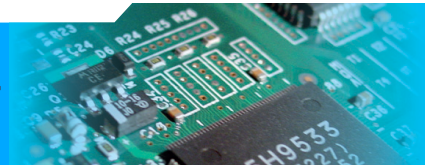
© <https://gitlab.istic.univ-rennes1.fr/plouzeau/ACO2018/tree/master/BadSwitch/src/fr/istic/nplouzeau>

©



Bad smell: message chains

- Example, some code in class A:
 - `Printers.getPrinters().findPrinterByName().getJobs().findUser("kate").getLogs().last().getPageCount()`
- Not always a bad smell, but it is the sign of a high coupling between the class A and Printers, Printer, Job, User, PrintLog
- Can be reduced by having a specialised class that does page counts, but it is a trade-off with the Middle man bad smell (a class that does nothing but propagate calls)



Bad smell: Comments

- Lengthy comments to explain a paragraph of code
 - This means that your code is too complex
 - Split it into shorter methods with meaningful names and responsibilities
- Obvious comment, for instance
 - `fileLineCount++; // Increment the line counts`
- In short try to make the code self explanatory



Remarks on tag comments

- Javadoc comments

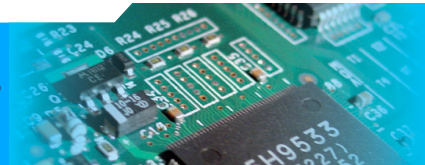
- These ones are special, as they basically document your structural API, which is not an option

- Tag comments

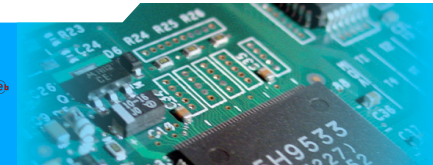
- `// TODO: find the largest item in the directory`

- `// FIXME: throws NPE on an empty directory`

- `// IMPROVE: this code has terrible performance for large directories`

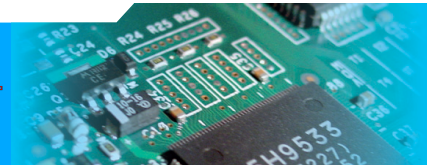


Conclusion on the Goals and rules part



Keep the principles in mind

- 🎯 Every time you are hesitating between several possibilities
- 🎯 For each possibility evaluate its SOLID, DRY,... rules compliance
- 🎯 Pick the one with the best score
- 🎯 Document your decision
- 🎯 Plan and prepare the corresponding tests



Conclusion on bad smell problems

- Many bad smells come from too many and too complex interdependencies (cf SOLID principles): structural problems
- Others come from insufficient or incorrect abstractions (eg not Liskov principle compliant): type problems
- Others come from poorly designed code (by the way it is not OO specific): code problems

