LOG8430: Architecture logicielle et conception avancée

Maintainability and Clean Code

Automne 2017

Fabio Petrillo
Chargé de Cours

Page 2

Who wrote this piece of code??

I can’t work like this!!

Page 3

Who wrote this piece of code??

I can’t work like this!!

After some analysis, you realize who wrote that code was **yourself**!!!

Page 4

ISO/IEC 25010 - quality model for software products

Page 5

ISO/IEC 25010 - quality model for software products

Page 6

Maintainability is the **most important** quality aspect, because we **can not achieve** the other aspects without to be able to **maintain our code easily**!!!

Page 7

Maintainability [ISO 25010, §4.2.7]

*"The degree of **effectiveness** and **efficiency** with which a product or system can be **modified** by the intended maintainers, where modifications can include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. It also includes installation of updates and upgrades. It can be interpreted as either **an inherent capability of the product or system to facilitate maintenance activities**, or the quality in use experienced by the maintainers for the goal of maintaining the product or system."*

Page 8

Analyzability [ISO 25010, §4.2.7.3]

*"The degree of effectiveness and efficiency with which it is possible to **assess the impact** on a product or system of an **intended change to one or more of its parts**, or to **diagnose** a product for **deficiencies** or causes of failures, or to identify parts to be modified"*

Page 9

Modifiability [ISO 25010, §4.2.7.4]

"The degree to which a product or system can be effectively and efficiently modified ***without introducing defects*** *or degrading existing product quality"*

Page 10

Testability [ISO 25010, §4.2.7.5]

"The degree of effectiveness and efficiency with which **test criteria can be established** for a system, product or component and tests can be **performed** to determine whether those criteria have been met."

Page 11

Modularity [ISO 25010, §4.2.7.1]

"The degree to which a system or computer program is composed of **discrete components** such that a **change to one component has minimal impact** on other components."

Page 12

Reusability [ISO 25010, §4.2.7.2]

"The degree to which an **asset** can be used in **more than one system**, or in **building other assets**."

Page 13

How to determinate the quality characteristics of maintainability?

Page 14

How to determinate the quality characteristics of maintainability?

Measuring a set of software product **properties.**

Page 15

Software product properties

- **Volume**: The overall **size** of the source code of the software product. Size is determined from the **number of lines of code** per programming language.
- **Duplication**: it concerns occurrence of **identical fragments of source code** in **more than one place** in the product.
- **Unit complexity**: The degree of **complexity** in the **units** of the source code. The notion of unit corresponds to the smallest executable parts of source code, such as **methods or functions**.
- **Module coupling**: The coupling between modules in terms of the number of **incoming dependencies** for the modules of the source code. The notion of module corresponds to a **grouping** of related units.

Page 16

Software product properties (cont.)

- **Component balance**: it is the product of the **system breakdown**, which is a rating for the number of top-level components in the system, and the

component size uniformity, which is a rating for the size distribution of those top-level components. The notion of top-level components corresponds to the first subdivision of the source code modules of a system into components, where a component is a grouping of source code modules.

- **Component independence**: Component independence is a rating for the percentage of code in modules that have **no incoming dependencies** from modules in other top-level components.

Page 17

https://www.sig.eu/files/nl/38_HeitlagerKuipersVisser-Quatic2007_08-3.pdf

Page 18

https://link.springer.com/article/10.1007%2Fs11219-011-9144-9

Page 19

https://www.sig.eu/files/en/018_SIG-TUViT_Evaluation_Criteria_Trusted_Product_Maintainability.pdf

Page 20

So, how to build **maintainable software**?

Page 21

A proposal

10 guidelines *to help you write source code that is easy to modify.*

"After 15 years of consulting about software quality, we at the Software Improvement Group (SIG) have learned a thing or two about

maintainability."

Page 22

Main principles behind the SIG's 10 guidelines

- Maintainability benefits most from adhering to **simple guidelines**.
- Maintainability is not an afterthought, but should be addressed from the **very beginning** of a development project. **Every individual contribution counts**.
- Some violations are **worse** than others. The more a software system complies with the guidelines, the more maintainable it is.

Page 23

Rating Maintainability

- maintainability is a quality characteristic on a scale
- different degrees of being able to maintain a system
- SIG divides the systems in the benchmark by **star rating**, ranging from **1 star** (**hardest** to maintain) to **5 stars** (**easiest** to maintain).
- Benchmark - results from **several hundreds** of standard system **evaluations**
- Empirical evidence that **issue resolution** and enhancements are **twice as fast** in systems with **4 stars** than in systems with **2 stars**.

Page 24

Benchmarking of software product quality

Page 25

Source code evaluation procedure

Page 26

SIG maintainability ratings

Page 27

Rating Maintainability

Page 28

A generic grouping of concepts and in Java

Page 29

The 10 SIG's guidelines

- **Write short units of code**: shorter units (that is, methods and constructors) are easier to analyze, test, and reuse.
- **Write simple units of code**: Units with fewer decision points are easier to analyze and test.
- **Write code once**: duplication of source code should be avoided at all times, since changes will need to be made in each copy. Duplication is also a source

of regression bugs.

- **Keep unit interfaces small**: units (methods and constructors) with fewer parameters are easier to test and reuse.
- **Separate concerns in modules**: modules (classes) that are loosely coupled are easier to modify and lead to a more modular system.

Page 30

The 10 SIG's guidelines (cont.)

- **Couple architecture components loosely**: top-level components of a system that are more loosely coupled are easier to modify and lead to a more modular system.
- **Keep architecture components balanced**: A well-balanced architecture, with not too many and not too few components, of uniform size, is the most modular and enables easy modification through separation of concerns.
- **Keep your codebase small**: a large system is difficult to maintain, because more code needs to be analyzed, changed, and tested. Also, maintenance productivity per line of code is lower in a large system than in a small system.

Page 31

The 10 SIG's guidelines (cont.)

- **Automate development pipeline and tests**: automated tests (that is, tests that can be executed without manual intervention) enable near-instantaneous feedback on the effectiveness of modifications. Manual tests do not scale.
- **Write clean code**: Having irrelevant artifacts such as TODOs and dead code in your codebase makes it more difficult for new team members to become productive. Therefore, it makes maintenance less efficient.

Page 32

Write Short Units of Code

- Limit the length of code units (methods) to **15 lines of code**.
- Small units are
  - easy to understand
  - easy to test
  - easy to reuse

Page 33

Minimum thresholds for a 4-star unit size rating

Page 34

Three quality profiles for unit size

Page 35

Write Simple Units of Code

- Limit the number of **branch points per unit to 4 (or limit code McCabe complexity to 5)**.
- A branch point is a statement where execution can take more than one direction depending on a condition (ex. *if* and *switch* statements).
- Do this by splitting complex units into simpler ones and avoiding complex units altogether.
- This improves maintainability because keeping the number of branch points low makes units easier to modify and test.
- We must put **a limit** on complexity!
- Simple units ease testing

Page 36

https://dzone.com/articles/cyclomatic-complexity-as-a-quality-measure

Page 37

Minimum thresholds for a 4-star unit complexity rating

Page 38

Three quality profiles for unit complexity

Page 39

Write Code Once

- Do not copy code.
- Do this by writing reusable, generic code and/or calling existing methods instead.
- This improves maintainability because when code is copied, bugs need to be fixed at multiple places, which is inefficient and error-prone.
- The fundamental problem of duplication is not knowing whether there is another copy of the code that you are analyzing, how many copies exist, and where they are located.
- Duplicated code contains a bug, the same bug appears multiple times.
- **Never** reuse code by **copying and pasting** existing code fragments!

Page 40

Minimum thresholds for a 4-star duplication rating

Page 41

Three code duplication quality profiles

Page 42

Keep Unit Interfaces Small

- Limit the number of parameters per unit to at most 4.
- Do this by extracting parameters into objects.
- This improves maintainability because keeping the number of parameters low makes units easier to understand and reuse.

Page 43

What is the problem in this code?

Page 44

Possible solution: parameter refactoring

Refactoring the parameters

Page 47

Minimum thresholds for a 4-star unit size rating

Page 48

Three quality profiles for unit interfacing

Page 49

Separate Concerns in Modules

- Avoid **large modules** in order to achieve **loose coupling** between them.
- Measuring using the number of **incoming calls** from other classes (***fan-in***) of all methods in the class.
- Do this by assigning responsibilities to **separate modules** and **hiding implementation** details behind **interfaces**.
- **Coupling** means that **two parts of a system** are somehow connected when changes are needed. When we change ***A***, we must change ***B***.
- The problem with these classes is that they become a **maintenance hotspot**.
- **Apply Single Responsibility Principle!**

Page 50

http://www.javabrahman.com/programming-principles/single-responsibility-principle-with-example-in-java/

Page 51

How to apply

- **Split Classes** to Separate Concerns
- **Hide** Specialized Implementations Behind **Interfaces**

- Replace Custom Code with **Third-Party Libraries/Frameworks**

Page 52

Module coupling risk categories

Page 53

Three quality profiles for module coupling

Page 54

Couple Architecture Components Loosely

- Achieve **loose coupling** between top-level components.
- Do this by **minimizing** the relative amount of **code** within modules that is **exposed** to (i.e., can receive calls from) modules in **other components**.
- It improves maintainability because independent components ease isolated maintenance.
- Components should be loosely coupled; that is, they should be **clearly separated** by having few entry points for other components and a limited amount of information shared among components.
- Low Component Dependence Allows for Isolated Maintenance
- Low Component Dependence Separates Maintenance **Responsibilities**

Page 55

High hard to change in isolation

Page 56

Designed versus implemented architecture

Page 57

Three quality profiles for component independence

Page 58

Keep Architecture Components Balanced

- Balance the number and **relative size of top-level** components in your code.
- Do this by organizing source code in a way that the number of components is close to 9 (i.e., between 6 and 12) and that the components are of approximately equal size.
- This improves maintainability because balanced components ease locating code and allow for isolated maintenance.

Page 60

Keep Your Codebase Small

- A **codebase** is a collection of source code that is stored in **one repository**

- **Keep your codebase as small as feasible.**
- Do this by avoiding codebase growth and actively reducing system size.
- This improves maintainability because having a small product, project, and team is a success factor.

Page 61

Probability of project failures by project size

Source: The Economics of Software Quality by Capers Jones and Olivier Bonsignour (Addison-Wesley Professional 2012). The original data is simplified into man-years (200 function points/year for Java).

Page 62

Large Codebases Are Harder to Maintain

Page 63

Automated Tests

- **Automate tests** for your codebase.
- Do this by writing automated tests using a **test framework**.
- This improves maintainability because automated testing makes development **predictable** and less risky.
- Measure **coverage** to determine whether there are enough tests.
- We should aim for at least 80% line coverage with a sufficient number of tests—that is, as many lines of test code as production code.

Page 64

BetterCodeHub - the guidelines implementation

https://bettercodehub.com/

Page 65

Page 66

https://codescene.io

Page 67

https://pmd.github.io/

Page 68

http://checkstyle.sourceforge.net/index.html

Page 69

https://www.sonarqube.org

Page 72

Clean Code

Page 74

"Writing clean code is what you must do in order to call yourself a professional."

—Robert C. Martin

Page 75

Clean Code

- Writing clean code requires the **disciplined** use of a myriad **little techniques** applied through a painstakingly acquired sense of "**cleanliness**."
- **Strategy** for applying those techniques to **transform bad code** into **clean code**, applying **series of transformations** until it becomes an **elegantly** coded system.
- Clean code is **focused**, simple and direct.
- Clean code always looks like it was written by **someone who cares**.
- Reduced **duplication**, high **expressiveness**, and **early** building of **simple abstractions**.

Page 76

Meaningful Names

- Use intention-revealing names (and use camelCase)
  - int d; // elapsed time in days
  - int **elapsedTimeInDays**;
- Avoid disinformation
  - eg.: use *account**List*** only it is a List.

- Make meaningful distinctions
  - public static void copyChars(char a1[], char a2[]) {
  - public static void copyChars(char **source**[], char **destination**[]) {
- Use Pronounceable names
  - class DtaRcrd102 { private final String pszqint = "102";
  - class **Customer** { private final String **recordId** = "102";
- Use Searchable Names
  - int **realDaysPerIdealDay** = 4;

Page 77

Meaningful Names

- Avoid Encodings (avoid Hungarian notation or member prefixes)
  - Boolean bBusy;
  - Boolean **busy**;
  - String m_desc;
  - String **description**;
- Pick one word per concept -> be consistent!
  - fetch, retrieve or get? controller or manager?
- Avoid using the same word for two purposes
- Use solution domain names
  - Account**Visitor** -> visitor pattern!

Page 78

Functions

- Functions should be **very small**

  "How short should a function be? In 1999 I went to visit Kent Beck at his home in Oregon. We sat down and did some programming together. At one point he showed me a cute little Java/Swing program that he called Sparkle. .... When Kent showed me the code, I was struck by ***how small all the functions were****. I was used to functions in Swing programs that took up miles of vertical space.* ***Every function in this program was just two, or three, or four lines long****. Each was transparently obvious. Each told a story. And each led you to the next in a compelling order.* ***That's how short your functions should be!****"*

Page 79

Functions

- Functions should do **only one thing** and **well**.
- Reading code from **top to bottom**: the StepDown Rule
- **Avoid switch** statements -> use **abstract factories and polymorphism**
- Function arguments -> more than **two arguments** should be avoided.
- **Flag arguments** are **ugly** -> split into **specific functions** without arguments
- **Have no side effects** -> promises to do one thing, but it also does other **hidden** things
- **Output** arguments should be **avoided**.
  - public void appendFooter(StringBuffer report)
  - report.appendFooter();
- **Prefer exceptions** to returning error codes

Page 81

Formatting

- **Choose formating rules and follow it!**
- **Vertical Formatting class**: typically 200 lines long; upper limit of 500 lines.
- **Each line** represents an expression or a **clause**, and each **group of lines**

represents a **complete thought**. Those **thoughts** should be **separated** from each other with **blank lines**.

- **Keep our lines short** -> around 45 characters and above 80 (max. 120).
- **Use indentation**!

Page 82

Design - Reasons to create a class

- Model **real-world** objects
- Model abstract objects
- **Reduce** and isolate **complexity**
- **Hide** implementation details
- Limit effects of **changes**
- **Hide** global data
- **Streamline** parameter passing
- Make **central** points of control
- Facilitate **reusable** code

- **Package** related operations

Page 83

Class Quality - Checklist

- Abstract Data Types
  - Have you thought of the classes in your program as abstract data types and evaluated their interfaces from that point of view?
- Abstraction
  - Does the class have a central purpose?
  - Is the class well named, and does its name describe its central purpose?
  - Does the class's interface present a consistent abstraction?
  - Does the class's interface make obvious how you should use the class?
  - Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?
  - Are the class's services complete enough that other classes don't have to meddle with its internal data?
  - Has unrelated information been moved out of the class?

Page 84

Class Quality - Checklist

- Abstraction (cont.)
  - Have you thought about subdividing the class into component classes, and have you subdivided it as much as you can?
  - Are you preserving the integrity of the class's interface as you modify the class?
- Encapsulation
  - Does the class minimize accessibility to its members?
  - Does the class avoid exposing member data?
  - Does the class hide its implementation details from other classes as much as the programming language permits?
  - Does the class avoid making assumptions about its users, including its derived classes?
  - Is the class independent of other classes? Is it loosely coupled?

Page 85

Class Quality - Checklist

- Inheritance
  - Is inheritance used only to model "is a" relationships - that is, do derived classes adhere to the Liskov Substitution Principle?

- Does the class documentation describe the inheritance strategy?
- Do derived classes avoid “overriding” non-overridable routines?
- Are common interfaces, data, and behavior as high as possible in the inheritance tree?
- Are inheritance trees fairly shallow?
- Are all data members in the base class private rather than protected?

- Other Implementation Issues
  - Does the class minimize direct and indirect routine calls to other classes?
  - Is all member data initialized in the constructor?
  - Have you investigated the language-specific issues for classes in your specific programming language?

Page 86

Quality Routines - checklist

- Is the reason for creating the routine sufficient?
- Have all parts of the routine that would benefit from being put into routines of their own been put into routines of their own?
- Is the routine’s name a strong, clear verb-plus-object name for a procedure or a description of the return value for a function?
- Does the routine’s name describe everything the routine does?
- Have you established naming conventions for common operations?
- Does the routine have strong, functional cohesion - doing one and only one thing and doing it well?

- Do the routines have loose coupling—are the routine's connections to other routines small, intimate, visible, and flexible?
- Is the length of the routine determined naturally by its function and logic, rather than by an artificial coding standard?

Page 87

Quality Routines - parameter-passing checklist

- Does the routine's parameter list, taken as a whole, present a consistent interface abstraction?
- Are the routine's parameters in a sensible order, including matching the order of parameters in similar routines?
- Does the routine have two parameters (max. four)?
- Is each input parameter used?
- Is each output parameter used?
- Does the routine avoid using input parameters as working variables?
- If the routine is a function, does it return a valid value under all possible circumstances?