Ingegneria del Software

Corso di Laurea in Informatica per in Management

Software quality and Object Oriented Principles

Davide Rossi

Dipartimento di Informatica – Scienze e Ingegneria

Università di Bologna

Design goal

- The goal of design-related activities is to produce high-quality software systems
- What is high-quality software?

Software and quality

- **External qualities**
  Qualities the end user can perceive
  - Functional
  - Non functional
- **Internal qualities**
  Qualities related to how the software is organized

External qualities

- Correctness
- Usability
- Efficiency
- Reliability
- Integrity
- Adaptability
- Accuracy
- Robustness

Source: Steve McConnell - Code Complete

Internal qualities

- Maintainability
- Flexibility
- Portability
- Re-usability
- Readability
- Testability
- Understandability

Source: Steve McConnell - Code Complete

Internal qualities

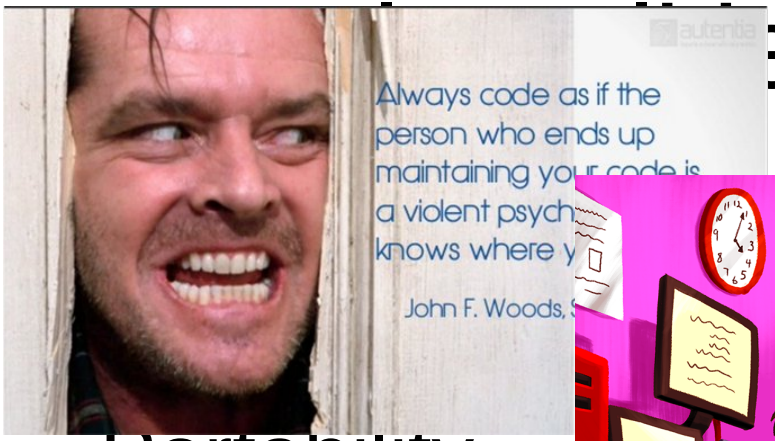
- Maintainability
- Flexibility
- Portability
- Re-usability
- Readability
- Testability
- Understandability

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

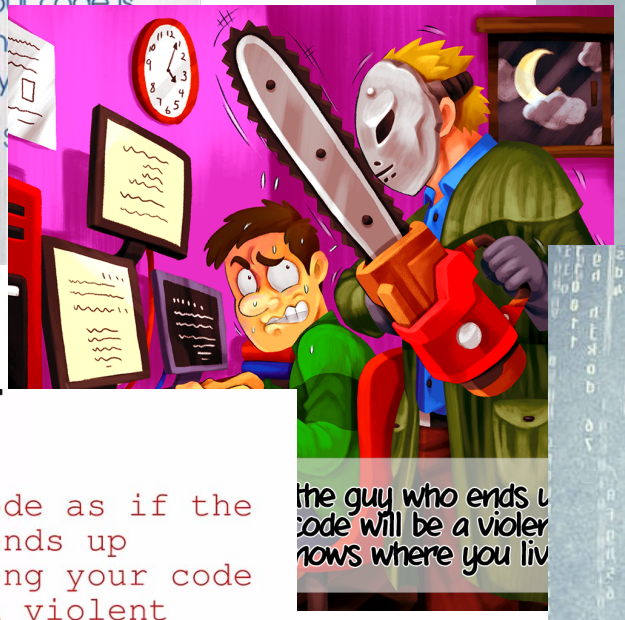
[John F. Woods]

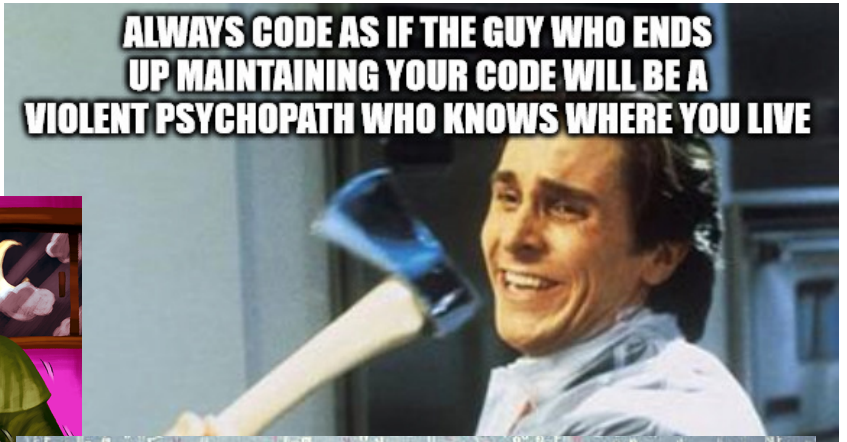
Source: Steve McConnell - Code Complete

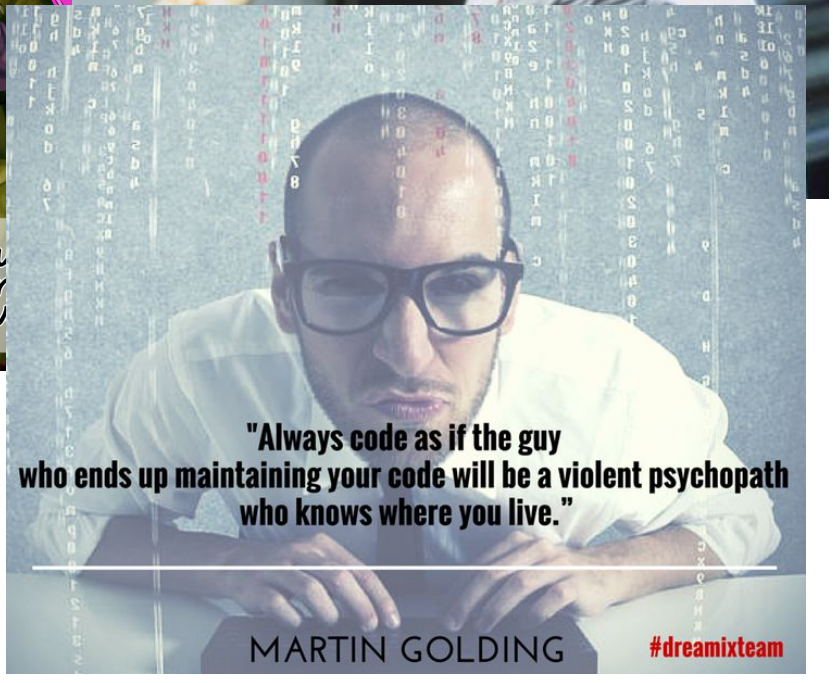
es

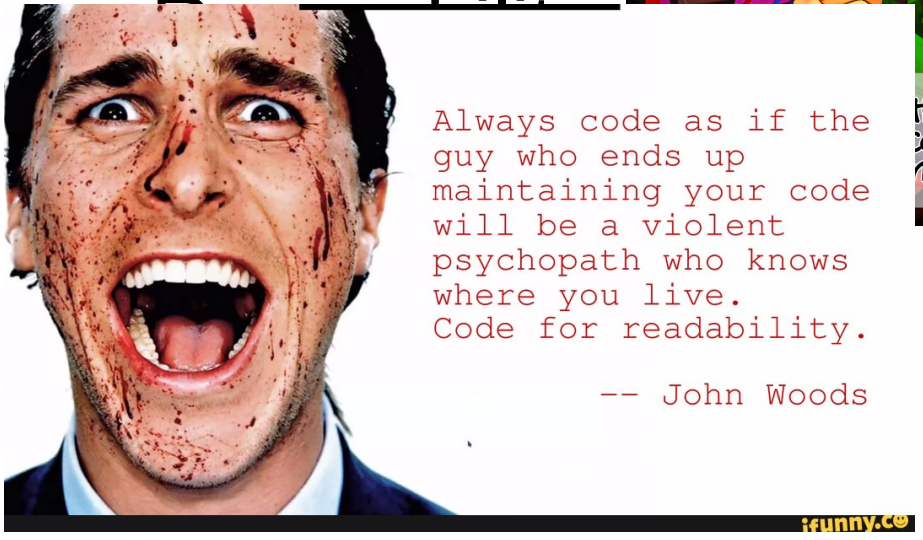


- Portability









Source: Steve McConnell - Code Complete

Software is not write-once

Costs associated with software evolution are high, an estimated 50%–90% of total software production costs.

SquaRE – ISO 25010

Software product Quality Requirements and Evaluation.

An evolution of ISO 9126 (and ISO 14598).

Three quality models: software product quality model, data quality model, quality in use model.

ISO 25010

Software product quality (characteristics can be measured internally or externally).

- Functional suitability
- Reliability
- Performance efficiency
- Operability
- Security
- Compatibility
- Maintainability
- Portability

ISO 25010

Quality in use

- Effectiveness
- Efficiency
- Satisfaction
- Safety
- Usability

Do we need a framework to assess quality ?

- Yes, at times.
- But most of the times we just want our software to be reusable and **designed for change**.

Do we need a framework to assess quality ?

- Yes, at tim
- But most tware to be reus

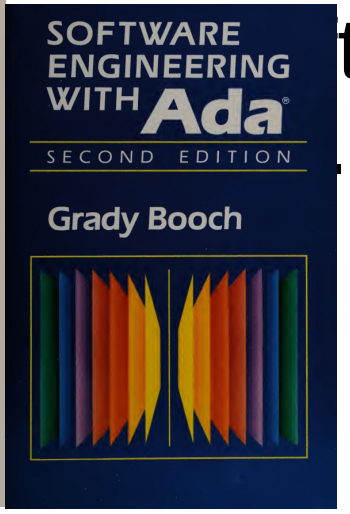
4.1 Goals of Software Engineering

A most obvious goal in software development is that the solutions meet the stated requirements. However, complete or consistent requirements specifications are rarely available, especially for very large systems. The user (and implementor) often has an incomplete understanding of the problem and so usually evolves the requirements during the development of the system. Furthermore, the problem is compounded whenever we have parallel hardware and software development, as is often the case with embedded computer systems. Finally, we must accept the reality of changes in requirements over the life cycle of our software systems. As we noted in Chapter 2 more resources are spent in the maintenance phase than in any other phase of the software life cycle. Large software systems don't die; they simply get modified.

Recognizing that change is a constant factor in software development, we must have a set of goals that transcends the effects of such change. In their classic paper, O. T. Ross, J. B. Goodenough, and C. A. Irvine describe these goals. "Four properties that are sufficiently general to be accepted as goals for the entire discipline of software engineering are modifiability, efficiency, reliability, and understandability" [1].

Modifiability

Modifiability is a difficult goal to achieve and to measure. Essentially, it "implies controlled change, in which some parts or aspects remain the same while others are altered, all in such a way that a desired new result is obtained" [2]. We may have to modify a software system for one of two reasons: first, we may have to respond to a change in the requirements of the system and second, we may have to correct an error that we introduced earlier in the development process.



OO Principles

We design OO systems, so we have to correctly identify our objects and find the *right mix* of encapsulation, inheritance and polymorphism in order to obtain high-quality software.

Principles can be used to guarantee the maximization of software qualities.

Basic OO concepts

- **Abstraction**: focus on essential characteristics (w.r.t. the perspective of the viewer).
- **Encapsulation**: hide the details (your status).
- **Inheritance**: behavior and state can be specialized.
- **Polymorphism**: behavior depends on who you are.

OO Principles

The critical design tool for software development is a mind well educated in design principles.

[C. Larman]

Design smells

- Rigidity
- Fragility
- Immobility
- Viscosity
- Needless complexity
- Needless repetition
- Opacity

Source: Robert C. Martin - Agile Principles, Patterns, and Practices

Design smells

- **Rigidity** is the tendency for software to be difficult to change, even in simple ways. A design is rigid if a single change causes a cascade of subsequent changes in dependent modules. The more modules that must be changed, the more rigid the design.
- **Fragility** is the tendency of a program to break in many places when a single change is made. Often, the new problems are in areas that have no conceptual relationship with the area that was changed. Fixing those problems leads to even more problems.

Design smells

- A design is **immobile** when it contains parts that could be useful in other systems, but the effort and risk involved with separating those parts from the original system are too great. This is an unfortunate but very common occurrence.
- **Viscosity** of the software: some options to make changes in a software system preserve the design; others do not. When the design-preserving methods are more difficult to use than the hacks, the viscosity of the design is high.
- **Viscosity** of environment: when the development environment is slow and inefficient.

Design smells

- **Needless complexity** of a design in when it contains elements that aren't currently useful. This frequently happens when developers anticipate changes to the requirements and put facilities in the software to deal with those potential changes.
- **Needless repetition**: copy and paste may be useful text-editing operations, but they can be disastrous code-editing operations.
  When the same code appears over and over again, in slightly different forms, the developers are missing an abstraction.

Design smells

- **Opacity** is the tendency of a module to be difficult to understand. Code can be written in a clear and expressive manner, or it can be written in an opaque and convoluted manner. Code that evolves over time tends to become more and more opaque with age. A constant effort to keep the code clear and expressive is required in order to keep opacity to a minimum.

Dependencies

- The root cause for most smells can be traced back to **dependency management**.
- *Dependency is the key problem in software development at all scales*. [K. Beck]
- Dependencies are potential paths for the diffusion of changes.
  - From UML's definition of dependency: "Indicates that changes to one model element [...] can cause changes in another model element".

SOLID

- Single responsibility principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

SRP: Single responsibility principle

- A class should have one, and only one, reason to change.
- Each responsibility is an axis of change.
- If a class has more than one responsibility, the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class's ability to meet the others. This kind of coupling leads to **fragile** designs.

OCP: open-closed principle

- A class should be open for extension, but closed for modification[(1)].
  - Can be generalized to: software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
- Failing to respect OCP leads to a change resulting in cascades of changes (**rigidity**). OCP advises us to refactor our design to avoid that.

(1) Bertrand Meyer - Object Oriented Software Construction

Refactoring

Refactoring is a disciplined technique for restructuring a software system so that its internal structure is modified without changing its external behavior

LSP: Liskov substitution principle

- The relation among classes in a hierarchy should be sub-typing
- “What is wanted here is something like the following substitution property: If for each object *o1* of type *S* there is an object *o2* of type *T* such that for all programs *P* defined in terms of *T*, the behavior of *P* is unchanged when *o1* is substituted for *o2* then *S* is a **subtype** of *T*.”(1)

(1) Barbara Liskov - Data Abstraction and Hierarchy

LSP: Liskov substitution principle

If a method *f*, accepting as argument a reference to *B*, misbehaves when is passed a reference to an instance of *D*, subclass of *B*, then *D* is **fragile** in the presence of *f*.

The Liskov Substitution Principle is one of the prime enablers of OCP.

Rules to guarantee respect of LSP

- Structural
  - Contravariance of method parameter types
  - Covariance of method return types
  - No new exceptions

Rules to guarantee respect of LSP

- Behavioral
  - Preconditions cannot be strengthened in the subtype
  - Postconditions and invariants cannot be weakened in the subtype
  - The history constraint must be respected
    - Subtypes can only change inherited state elements accordingly with the allowed mechanisms present in the supertype

ISP: interface segregation principle

The dependency of one class to another one should depend on the smallest possible interface.

Or: clients should not be forced to depend on methods they do not use.

Failing to respect this principle can lead to unneeded dependencies and to degenerate implementations of interfaces, causing needless complexity and potential violations of LSP.

DIP: dependency inversion principle

Depend upon Abstractions.

A) High level modules should not depend upon low level modules. Both should depend upon abstractions.

B) Abstractions should not depend upon details. Details should depend upon abstractions.

Layered architectures

All well structured object-oriented architectures have clearly-defined layers, with each layer providing some coherent set of services through a well-defined and controlled interface. [G. Booch]

Layered is a family of very common architectural styles.

Beware of layers

A naive application of the layered style can easily lead to a violation of the DIP.

A common solution is to make lower layers dependent upon a service interface declared in upper layers.