Maintainability and Agile development

Author: Mika Mäntylä
<mika.mantyla@soberit.hut.fi>
ISO 9126 Software Quality Characteristics

- **Functionality**: Are the required functions available in the software?
- **Reliability**: How reliable is the software?
- **Portability**: How easy is it to transfer the software to another environment?
- **Maintainability**: How easy is it to modify the software?
- **Usability**: How easy is it to transfer the software to another environment?
- **Efficiency**: How efficient is the software?

**ISO/IEC 9126**
Quality attributes in ISO9126

- **External and Internal Quality**
  - **Functionality**
    - Suitability
    - Accuracy
    - Interoperability
    - Security
    - Compliance
  - **Reliability**
    - Maturity
    - Fault tolerance
    - Recoverability
  - **Usability**
    - Understandability
    - Learnability
    - Operability
    - Attractiveness
  - **Efficiency**
    - Time behavior
    - Resource utilization
  - **Maintainability**
    - Analyzability
    - Changeability
    - Stability
    - Testability
  - **Portability**
    - Adaptability
    - Installability
    - Co-existence
    - Replaceability
ISO 9126 Software Quality Characteristics

How easy is it to transfer the software to another environment?

How easy is it to modify the software?

ISO/IEC 9126

Portability

Maintainability
Design and Structure Quality

- The software internal structure greatly affects many software quality attributes
  - Maintainability, portability, reliability, and functionality are influenced by internal structure quality

- Reasons for controlling internal software structure
  - The effort in future development activities depends on the current structure
  - Agreement on what is acceptable software structure
  - To stop bad programmers
    - Anecdotal evidence suggest that some programmers actually do more harm than good
  - To prevent harmful side effects of software evolution. Laws of software evolution [Lehman & Belady]
    - Software which is used in a real-world environment must change or become less and less useful in that environment
    - As an evolving program changes, its structure becomes more complex, unless active efforts are made to avoid this phenomenon
Assessing Design and Structure Quality

- Source code metrics
- Reverse-Engineering
- Subjective Quality Indicators
Metrics and software structure - Cases

- Study on OO metrics and maintenance effort from US industry (1993)
  - Two commercial systems written in Classic-Ada
  - Data collection took three years
  - Maintenance effort was measured in number of lines changed per class
  - Study combined eight measures and showed that the combined measure predicted accurately the maintenance effort

- Construction of Maintainability index at HP (1994)
  - They created a measure that combined three dimension of maintainability
    - 1) control structure, 2) information structure, 3) typography, naming, commenting
  - Measure was adjusted according to HP’s programmers opinions
  - Analysis assisted HP in
    - Buy-versus-build decision,
    - Controlling software entropy over several versions
    - Identify change prone subcomponents
    - Finding targets and assessing the efforts of reengineering
Metrics and software structure – Cases cont’d

- Code metrics controlling internal software quality
  - Reports from HP (1994) tell that the cyclomatic complexity of FORTRAN programs was not allowed to exceed 14
    - This was based on previous modules change rate data
- F-Secure has limits for nested blocks, cyclomatic complexity, etc
- Metrics can offer a way to quantify vague quality concepts like “Maintainability” and “Flexbility”
- Code metrics can protect your assets from the harmful side-effects of software evolution
- Comments metrics tools as part of software process
  - At end of each project metrics are calculated to:
    - “This type of information is also needed... we see if there are some real big structural problems”
  - Developers opinion:
    - “Yes they have been useful... It confirmed previous opinions
    - “I did not bother to study the fancier metrics”
Metrics - Counter Points

- Problems in using metrics to control software internal quality
  - You should define your own limits based on historical data
  - “Measures don’t really tell whether it is well programmed”
- Comments on source code metrics
  - “How could that metric be useful... only thing we could use it would be to measure how much the code has improved after refactoring
  - “Quantity of e.g. couplings is not important. Quality of couplings is important”
  - “Metrics are phony... Code can not be forced to certain number of lines, this can only mess up things... Piece of code must implement clear distinct entity”
Metrics - Summary

- However, by measuring software structure quality you send a message of its importance
- Programs:
  - CCCC can provide code metrics for C, C++, Java
    - Offers also threshold limits for the metrics
  - To measure duplication: Same (sourceforge) and CloneFinder (commercial)
Assessing Design and Structure Quality

- Source code metrics
- Reverse-Engineering
- Subjective Quality Indicators
Reverse-Engineering Tools and software structure

- Reverse Engineering is the process of analyzing a subject system to
  - Identify the system's components and their interrelationships
  - Create representations of the system in another form or at a higher level of abstraction

- Reverse Engineering is used to
  - Re-document the system
    - “Agile documenting”
  - Part of architecture reconstruction (Nokia)
    - Browsing visual model of source code and using application knowledge to combine
  - Analyze whether the design matches the actual source

- Comments on reverse-engineering
  - “The idea is stop the decay of software architecture”
  - “It’s a good way to see associations that are not suppose to be there”
Assessing Design and Structure Quality

- Source code metrics
- Reverse-Engineering
- Subjective Quality Indicators
Subjective Quality Indicators

- Agile development belivies in people over tools
  - Subjective quality indicators is natural approach
  - Human judgement is needed for all design and structure decision since they are dependent on the context
- Different approaches
  - Structured Design Principles
  - Anti-Patterns
  - Code Smells
Structured Design Principles

- Structured design was used before OOD and it presented ideas of low coupling, and high cohesion
- **Low coupling** means that software elements e.g. routines, classes, modules, sub-systems, etc should not be heavily connected to each other.
- **High cohesion** means that closely related functionalities or responsibilities should be inside single software element.
  - E.g. Class that performs several unrelated operations has low cohesion.
- Some of the problems of high coupling are described below
  - Changing one software element will also force you to change the other elements that have couplings with the first element
  - Software element is more difficult to understand since it relies heavily on other elements
  - Reusing such software element is more difficult
  - More rigorous testing is needed for software element that is heavily coupled
Anti-Patterns and Bad Code Smells

- Describe some of the common mistakes in software development
  - Subjective
Anti-Pattern examples – Lava Flow

- **Problems emerge:**
  - Development
- **Also know as:**
  - Dead Code
- **Root Causes:**
  - Avarice, Sloth
- **Description:**
  - Complex code that seems important, but nobody knows what it does
  - Classes that are not clearly related to whole systems
  - Old design considerations can be seen in the code

- **Causes:**
  - Proof-of-concept prototype placed in production
  - Single-Developer
  - Changing requirements
  - Trial code kept in hand just-in-case

- **Solution**
  - Process
    - CM will ensure that code can be thrown away
    - Minimize scope-creep
    - Is prototype good enough for production
    - System discovery
Bad Code Smells – Examples 1/2

- **Bloaters**
  - Something has grown so large it cannot be effectively handled
  - Smell in this category
    - Long Method, Large Class, Primitive Obsession, Long Parameter List, Data Clumps
  - These smells likely grow little by little
    - Hopefully nobody designs e.g. Long Methods

- **Object-Orientation Abusers**
  - Object Oriented concept is not fully understood
  - Smell in this category
    - Switch statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces, Parallel inheritance hierarchies

- **Change Preventers**
  - These smells make changing the system unnecessarily difficult
  - Smell in this category
    - Divergent change, Shotgun surgery
Bad Code Smells – Examples 2/2

- **Dispensables**
  - All code needs effort to understand and maintain
  - If code is not used or redundant it needs to be removed
  - Smell in this category:
    - Duplicate code, Speculative Generality

- **Encapsulators**
  - Encapsulation is one of OO design principles
    - However it can go too far
    - There can be too little of it
  - Smell in this category:
    - *Message Chains and Middle Man*

- **Couplers**
  - Low coupling between objects/classes is one the desirable goals of OO software
  - Smell in this category:
    - Feature Envy, Inappropriate Intimacy
Bad Smell vs. Anti-Patterns

- Some overlap between smells and Anti-Patterns
  - E.g. Large Class = Blob
- Smells are more concrete and much more closer to code
- Smells are easier to fix
  - Less effort is needed to remove single smell
  - The Refactorings given are more precise
- Smell can be thought as more specified development level Anti-Patterns
- Anti-Patterns cover large range of topics
  - Architecture, Project Management, Process, Role, Technology
Summary: Assessing Design and Structure Quality

- **Metrics Pro’s**
  - Provides objective numbers
  - Does not require in depth understanding of the system
  - Highlight the importance of software structure

- **Reverse-engineering Pro’s**
  - Provides higher level view of the source code
  - Can be compared with the intended design

- **Subjective QI Pro’s**
  - Utilizes the application knowledge
  - Easy to apply
  - Motivation to fix

- **Metrics Con’s**
  - Does not say anything about the correctness (quality)
    - e.g. correct associations

- **Reverse-engineering Con’s**
  - Requires in depth understanding of the system

- **Subjective QI Con’s**
  - Differences between people
  - Comparison between systems difficult
    - “My baby” syndrome
References

Topics

- Refactoring
  - Refactoring tools – Taxonomy, comparison, practical experiences
- Quality attributes
  - Quality attribute X and agile development
    - X = Security / Maintainability / Efficiency / Reliability
- Design by Contract
  - Developer level QA: Design by contract & Unit test
- Human factors
  - Individual differences in producing quality
    - Quality = Finding bugs / Maintainable code / Productivity (more features) / Error-free code