Software Maintainability Ontology in Open Source Software

Celia Chen
qianqiac@usc.edu
ARR 2018, USC

How do others measure software maintainability?

Most popular methods: Automated analysis of the code

- Maintainability Index

- Technical Debt

Less popular methods:

- Reuse cost models that estimate maintainability of potentially reusable components based on human-assessed maintainability aspects such as code understandability and structure

Pros of both approaches

Pros of Automated Analysis:

- Very popular and often used in maintenance practice
- Very easy to use

Pros of Human-assessed Analysis:

- More accurately reflect the actual maintenance effort spent on tasks
- Reflect the overall quality of the software

Cons of both approaches

Cons of Automated Analysis:

- Composite metrics, which makes it hard to determine which of the metrics cause a particular total value
- The metric values means differently on the type of programming language, the programmer, the perception of the quality of code, etc.
- Points out problematic area but don’t reflect the overall quality of the software

Cons of Human-assessed Analysis:

- Expensive and hard to obtain
- Difficult to measure since quality can be subjective to different programmers
- Lack of detailed information on the problematic areas

What to do?

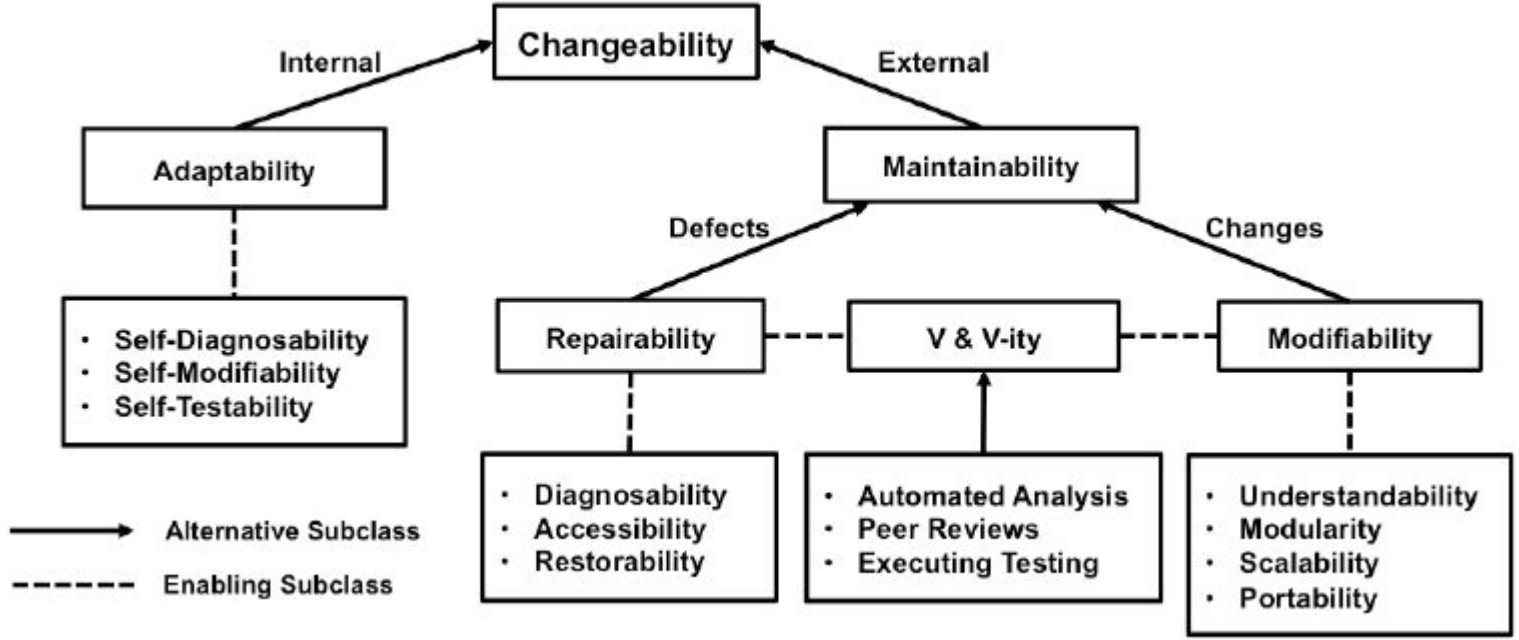
Software quality ontology and framework

- Valuable knowledge is available.
- Software quality ontology provides a structured way to look at quality.
- However, in practice, organizations choose not to use them.
- Can we map software quality ontology to software in practice?

An empirical study on Mozilla products

- We conducted an empirical study on several Mozilla products to see whether an existing ontology can be expressed in their products.
- Since bugs reflect the problems occured in developing products, we used bug reports as the study subject to examine the relationship with ontology.
- Our goal is to map SQ concerns to bugs and then to the root causes of the bugs to study how these SQ changes in relate to the bugs and their root causes.

Software Maintainability Ontology



Root causes

These were summarized from existing literature, validated by expert in both theory and practice.

Internal Implementation	Interface	External
Data design/usage	Functionality design/usage	Development/deployment environment
Resource allocation/usage	GUI design/usage	External Tools/infrastructure
Exception handling	Unexpected interactions	Test cases/suites
Functionality		Previous releases
Performance		
Semantic		
Documentation/Comments		

Study Subject

- We chose 11 products from Mozilla Community, 6 client-based software and 5 server-based software.
- For products that have less than 1000 bugs, we sampled 30% of the bugs. Otherwise, we sampled 10% of the bugs.
- In total, we examined 6371 bugs for this study.
- Experts in software quality (2 professors, 1 senior researcher, 1 phd student) and members of the Mozilla community (1 senior contributor, 7 active contributors) were recruited to tag each bug.
- In order to indicate the random sampling error in our results, we calculated the margin of error for the sample.

Research Questions

1. As software becomes mature, how does software maintainability change? What are the dominant SQs contributing to software maintainability?
2. Which SQ problems tend to appear more in high severity bugs? Compare quick-fix with long-fix bugs, which SQs tend to get hurt more?
3. How do the subgroups of Maintainability SQs relate to each other?

RQ1

We grouped the bugs annually for each product, from the earliest bug found on Bugzilla to 2018.

For example:

The earliest bug reported for Firefox for Android was in 2009 so bugs reported between 2009 and 2010 were grouped into period 1.

Results of RQ1

- As software evolves, the client-based software showed an decrease in maintainability issues, which means the maintainability of these products increased as the software evolves. Client-based software took less time to fix bugs that were tagged with maintainability.
    - This contradicts with the Lehman's Laws #7 - “Declining Quality”
- However, the server-based software showed an increase in maintainability issues, which means the maintainability of these products decreased as the software evolves.

Results of RQ1

- The actual contribution of each SQ varies among products.
- During earlier phase, accessibility is the dominant type; while during later phase (after 5 years), portability is the dominant type for client-based software while understandability is the dominant type for server-based software.
- However, they all showed a trend of increasing in portability and decreasing of accessibility.

RQ2 - Severity

We group bug reports based on the Severity tag into 4 groups:

- Blocker
- Critical
- Major
- Others (including enhancement, trivia, others, minor, normal, etc.)

Results of severity

- Among all the blocker bugs, 45.61% express maintainability related problems.
- Among all the critical bugs, 23.98% express maintainability related problems.
- Among all the major bugs, 34.12% express maintainability related problems.
- Among all the other bugs, 22.51% express maintainability related problems.
- Among all the blocker, critical and major bugs, the #1 SQ contributes to maintainability is portability.
- Among all other bugs, the #1 SQ contributes to maintainability is understandability.

RQ2 - Fix time

- We graphed the distribution of fix time of all the sampled bugs and picked 10% of each tail to represent quick-fix and long-fix bugs.
- Average time of quick-fix: 0.24 days
- Average time of long-fix: 2006.12 days

Results of fix time

- Among quick-fix bugs, 26.92% express maintainability related problems.
- Among long-fix bugs, 25.48% express maintainability related problems.
- Quick-fix bugs have most understandability issues, with the top root causes of semantic problem and interface functionality problem.
- Long-fix bugs have most portability issues, with the top root causes of external Development and deployment environment problem.

RQ 3: Dependency

- We examined all the bugs that depend on other bugs.
- There are in total 306 bugs that depend on other bugs.
- Number of dependent bugs vary from 1 to 98. Most of the bugs depend on 1 or 2 other bugs.
- For bugs that depend on multiple bugs, we combine consecutive SQ.
  - Bug ID 2341538 depends on bugs 234187, 3419203, 1123194 and 5319832.
  - Accessibility, Understandability, n/a, n/a, n/a
  - We record this pattern as: Accessibility <- Understandability, n/a
  - The order of the tags of the dependent bugs doesn’t matter.
- We found 97 distinct patterns.

Results of dependency

- Accessibility contains the most bugs that are within the same product.
- Portability and understandability contains the most bugs that are reported in different products.
- Modularity contains the most bugs that depend on more than one other bugs.

Top 5 patterns observed in Mozilla

Original Bug Tag	Original Bug - Root Cause	Root cause details	Dependent Bug(s) Tag	Original Bug - Root Cause	Root cause details	Occurence	% of the occurence within each SQ	% of the occurrence with the total number of bugs
portability	internal Implementation	performance	N/A			42	32.43%	13.68%
modularity	internal Implementation	Resource allocation/usage	accessibility	Internal Implementation	functionality	33	57.89%	10.75%
accessibility	Interface	GUI	accessibility	interface	functionality	18	34.62%	5.86%
scalability	internal Implementation	performance	N/A			16	57.14%	5.21%
accessibility	interface	unexpected interaction	accessibility	interface	functionality	13	26.92%	4.23%

Contribution of this empirical study

- Validates the importance of understanding software maintainability
- Provides a set of interesting observations for researchers and practitioners to look at maintainability from a non code analysis point of view
- The dependency patterns of bugs may contribute to existing bug prediction approaches
- Introduces an opportunity to automate the process of mapping a valuable knowledge of software quality ontology into practice

Future work

- Automatically map a bug report to a quality concern, thus, forming a complete picture of the current quality status of a project.
  - We are developing an automatic approach through utilizing fuzzy rules to find patterns of each Maintainability SQ to locate bugs that can potentially be mapped into a SQ and then use word embedding, semantic meaning of the bug summary and domain knowledge to automatically map a quality concern to a bug.
- A large-scale empirical study on open source and closed source projects to study the change in maintainability and its subgroup SQs.
- A comparison study on the result of quality tags and the automated analysis