Comparing Automated and Human Maintainability Assessment Approaches

Celia Chen, Reem Alfayez, Kamonphop Srisopha, Barry Boehm, Lin Shi
Agenda

- Definition of software maintenance and maintainability
- The importance of measuring maintainability
- Automated maintainability metrics
- Human-assessed maintainability index
- Main research question
- Experimental setup
- Data analysis and results
- Conclusion
Software Maintainability

**Software maintenance** - "the modification of a software product after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."

**Software maintainability** - "the ease with which maintenance can be carried out."
We define software maintainability as

“The average time (effort) required to first understand the source code and then perform maintenance tasks (e.g., fixing existing issues and adding new feature requests)”

The importance of measuring maintainability

- For OSS community
- For OSS adopter
- For in-house software developer

Koskinen’s survey (2009) found that

- 75-90% : Business and command-control software
- 50-80% : Cyber physical system

Software costs are incurred during maintenance
Low maintainability

Difficult to modify

Increase maintenance cost

Difficult to find solutions for bugs

Increase effort
Automated Maintainability Metrics

Maintainability Index (MI)

○ An index that represents the ease of maintaining the code
○ Widely used in the industry
Maintainability Index (MI)*

\[
MI_{woc}^{(source\ file)} = 171 - 5.2 \times \ln HV - 0.23 \times CC - 16.2 \times \ln LLOC
\]

\[
MI_{wc}^{(source\ file)} = 50 \times \sin \sqrt{2.46 \times CM}
\]

\[
MI^{(source\ file)} = MI_{woc}^{(source\ file)} + MI_{wc}^{(source\ file)}
\]

\[
MI = \frac{\sum MI^{(source\ file)}}{Number\ of\ Source\ files}
\]

<table>
<thead>
<tr>
<th>Halstead Volume (HV)</th>
<th>Cyclomatic complexity (CC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count of lines (LLOC)</td>
<td>Percent of lines of comments (CM)</td>
</tr>
</tbody>
</table>

*MI is developed by the University of Idaho in 1991 by Oman and Hagemeister*
Automated Maintainability Metrics-Technical Debt

"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.”

- Ward Cunningham, 1992
Technical Debt

SonarQube: is an open source platform for continuous inspection of code quality by performing static code analysis.

Based on SQALE method
# Human-Assessed Maintainability Metrics

## COCOMO II software understanding (SU) factors

**Rating scale for Software Understanding Increment (SU)**

<table>
<thead>
<tr>
<th>Factor</th>
<th>Very Low</th>
<th>Low</th>
<th>Nominal</th>
<th>High</th>
<th>Very High</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structure</strong></td>
<td>Very low cohesion, high coupling, spaghetti code.</td>
<td>Moderately low cohesion, high coupling.</td>
<td>Reasonably well structured; some weak areas.</td>
<td>High cohesion, low coupling.</td>
<td>Strong modularity, information hiding in data/control structures.</td>
</tr>
<tr>
<td><strong>Application Clarity</strong></td>
<td>No Match between program and application worldviews.</td>
<td>Some correlation between program and application.</td>
<td>Moderate correlation between program and application.</td>
<td>Good correlation between program and application.</td>
<td>Clear match between program and application worldviews.</td>
</tr>
<tr>
<td><strong>Self-Descriptiveness</strong></td>
<td>Obscure code; documentation missing, obscure or obsolete.</td>
<td>Some code commentary and headers; some useful documentation.</td>
<td>Moderate level of code commentary, headers, documentation.</td>
<td>Good code commentary and headers; useful documentation; some weak areas.</td>
<td>Self-descriptive code; documentation up-to-date, wellorganized, with design rationale.</td>
</tr>
</tbody>
</table>
Main research question

Which metrics can more accurately reflect software maintainability?

Maintainability

Automated Maintainability Metrics (Maintainability Index and Technical Debt)

Human-Assessed Maintainability Metrics (COCOMO II software understanding factors)
Context Selection

Selecting Criteria

- Has more than one official release
- The latest stable release
- Has online issue tracking system\(^1\)
- Has fully accessible source code
- Relatively reasonable size\(^2\)

\(^1\)Bugzilla, Jira, or SourceForge.
\(^2\)Less than 250KSLOC (avg 51KSLOC)
## Characteristics of project data sources

<table>
<thead>
<tr>
<th>Language</th>
<th>Number of Projects</th>
<th>Average SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP</td>
<td>6</td>
<td>35,200</td>
</tr>
<tr>
<td>Java</td>
<td>5</td>
<td>67,145</td>
</tr>
</tbody>
</table>

![Codeception](image1.png)

![JdrJava](image2.png)

![PHPUnit](image3.png)

![DocFetcher](image4.png)

![Composer](image5.png)

![Joomla](image6.png)
Experimental Setup

6 maintainers

- Industrial experience
- Requirements capture
- Analysis and Design
- Quality management
- Software Testing
- User Interface Design
Experimental Setup

- There are 44 tasks total.
- Tasks are assigned randomly to students and a task could be assigned to multiple maintainers.
- Maintainers have to report effort spent on task and answer questionnaire (consist of questions derived from COCOMO II SU factors) and provide rationales.
- If a maintainer could not finish the assigned task, the maintainer has the option to either continue working on the same task the following week or work on a new task.
- Each maintainer has a different total number of assigned projects and tasks based on their availability and experience.

### Examples of Maintenance Task

<table>
<thead>
<tr>
<th>Project</th>
<th>Task Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocFetcher</td>
<td>Search tabs within one window (more than one search at the same time)</td>
<td>Feature Request</td>
</tr>
<tr>
<td>PhpUnit</td>
<td>Abstract class inheritance issues</td>
<td>Bug</td>
</tr>
</tbody>
</table>
# Questions derived from the COCOMO II SU factors

| Structure                          | ● How well are the codes organized?  
|                                  | ● How well are the classes defined in terms of class structure?  
|                                  | ● How well are the variables named?  
|                                  | ● How well are the classes named?  
|                                  | ● Are the classes highly coupled?  
| Application Clarity               | ● How well does the software match its application worldviews?  
|                                  | ● Are you able to understand the features as described?  
| Self-Descriptiveness              | ● How good are the comments?  
|                                  | ● Are there sufficient meaningful comments within the source code?  
|                                  | ● How self-descriptive are the codes?  
|                                  | ● How well is the documentation written?  
|                                  | ● Does the software have sufficient documentation to describe interfaces, size, or performance?  
|                                  | ● How well does the current documentation match the current software?  |
**An example answer from one of the maintainers**

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>How well are the code organized?</td>
<td>7 Well defined class, good modularity of the code</td>
</tr>
<tr>
<td>How well are the comments?</td>
<td>There are modules without any comments and it becomes difficult to know what the functions are doing without reading and understanding the code beneath.</td>
</tr>
<tr>
<td>How well are the variables named?</td>
<td>6 Names should have been more descriptive/meaningful</td>
</tr>
<tr>
<td>How well are the classes named?</td>
<td>9 Class names indicate what they are doing on a high level</td>
</tr>
<tr>
<td>How well are the classes defined in terms of class structure?</td>
<td>Structuring is good, but it's easy to get lost in the code as many classes have almost similar named functions and as a code reader, this confuses if one is mapping class definition with the function calls</td>
</tr>
</tbody>
</table>

**Please rank the tasks based on difficulty. 1 being really hard, 5 being really easy**

1. Change the icon

   There was any easy way to do this, just replace the image file in the folder with the new one with the same name. If done through code, I would rate this 4 as I had to search where the image was being loaded.

**Did you use any forums or mailing list to help you finish the tasks?**

| Yes

**If yes, how was your experience interacting with people?**

9 Was told the fix within 30-40mins interval of posting the question
Data Analysis

• The comparison is performed using Pearson product-moment correlation coefficient.
• The results are intended as statistically significant at 0.05 (p < 0.05), which equals to a confidence level of 95%.
• Any p-values that is well below 0.05 can be concluded as a strong correlation
• Any Pearson coefficient (R) that is below 0 can be concluded as negative correlation
## Correlation Coefficients Matrix between Automated Maintainability Metrics and Average Effort Spent

<table>
<thead>
<tr>
<th>Average Effort</th>
<th>TD</th>
<th>MI</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>-0.35835</td>
<td>-0.83996</td>
</tr>
<tr>
<td>R Standard Error</td>
<td>0.09684</td>
<td>0.03272</td>
</tr>
<tr>
<td>t</td>
<td>-1.15153</td>
<td>-4.18987</td>
</tr>
<tr>
<td>p-value</td>
<td>0.27918</td>
<td>0.00121</td>
</tr>
<tr>
<td>H0 (5%)</td>
<td>accepted</td>
<td>rejected</td>
</tr>
</tbody>
</table>
### Human-assessed Maintainability Metrics

**Result and its Analysis**

<table>
<thead>
<tr>
<th>Average Effort</th>
<th>Code Organization</th>
<th>Class Names</th>
<th>Variable Names</th>
<th>Class Structure</th>
<th>Coupling</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>-0.91552</td>
<td>-0.8762</td>
<td>-0.87884</td>
<td>-0.83562</td>
<td>-0.84778</td>
<td>-0.94013</td>
</tr>
<tr>
<td>R Standard Error</td>
<td>0.01798</td>
<td>0.02581</td>
<td>0.02529</td>
<td>0.03353</td>
<td>0.03125</td>
<td>0.01291</td>
</tr>
<tr>
<td>t</td>
<td>-6.82751</td>
<td>-5.45405</td>
<td>-5.52591</td>
<td>-4.56361</td>
<td>-4.79558</td>
<td>-8.27519</td>
</tr>
<tr>
<td>p-value</td>
<td>0.00008</td>
<td>0.0004</td>
<td>0.00037</td>
<td>0.00136</td>
<td>0.00098</td>
<td>0.00002</td>
</tr>
<tr>
<td>H0 (5%)</td>
<td>rejected</td>
<td>rejected</td>
<td>rejected</td>
<td>rejected</td>
<td>rejected</td>
<td>rejected</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Average Effort</th>
<th>Application Clarity</th>
<th>Comment Quality</th>
<th>Self-descriptive Code</th>
<th>Documentation Quality</th>
<th>Self-Descriptiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>-0.87032</td>
<td>-0.94669</td>
<td>-0.92139</td>
<td>0.03936</td>
<td>-0.93557</td>
</tr>
<tr>
<td>R Standard Error</td>
<td>0.02695</td>
<td>0.01153</td>
<td>0.01678</td>
<td>0.11094</td>
<td>0.01386</td>
</tr>
<tr>
<td>t</td>
<td>-5.30159</td>
<td>-8.81582</td>
<td>-7.11247</td>
<td>0.11817</td>
<td>-7.94815</td>
</tr>
<tr>
<td>p-value</td>
<td>0.00049</td>
<td>0.00001</td>
<td>0.00006</td>
<td>0.90853</td>
<td>0.00002</td>
</tr>
<tr>
<td>H0 (5%)</td>
<td>rejected</td>
<td>rejected</td>
<td>rejected</td>
<td>accepted</td>
<td>rejected</td>
</tr>
</tbody>
</table>
The comparison

Which metrics can more accurately reflect software maintainability?

Human-Assessed Maintainability Metrics (COCOMO II software understanding factors)

Automated Maintainability Metrics (Technical Debt)
Conclusion

Which metrics can more accurately reflect software maintainability?

The three approaches can be synergetic in estimating relative maintainability effort, and in identifying key parts of the software most needing maintainability improvement, rather than trying to pick a best single approach.

Technical Debt and MI:
- Help in identifying the particular parts of the software most needing maintainability improvement, while using a relatively low level of human effort.
- Help prioritize the parts of the software that would most benefit from human maintainability assessment and maintenance effort estimation.
Backup Slides
Halstead Volume

According to Halstead, a computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operands.

Operators include:
- Reserved words (while, if, do, class, etc)
- Qualifier (const, static)
- expressions and arithmetic operators (+, >, =)
- etc.

Operand includes:
- numeric constant
- literal
- identifiers
- etc.

\[ n_1 = \text{number of distinct operator} \]
\[ n_2 = \text{number of distinct operands} \]
\[ N_1 = \text{Total number of occurrences of operators} \]
\[ N_2 = \text{Total number of occurrences of operands} \]

Program Length: \[ N = N_1 + N_2 \]
Vocabulary Size: \[ n = n_1 + n_2 \]

Program Volume: \[ N \times \log_2(n) \]
McCabe’s Cyclomatic Complexity

- Cyclomatic Complexity aims to capture the complexity of a code function/method in a single number. The metric develops a Control Flow Graph (CFG) that measures the number of linearly independent paths through a program module*

\[ CC = E - N + 2 \times P \]

\[ E = \text{number of edges} \]
\[ N = \text{number of nodes} \]
\[ P = \text{number of module/ connected function/method}. \]

```java
void Cyclomatic_example() {
    int i = 1;
    while(i<10){
        if(i==3){
            System.out.println("Here i = 3");
        } else{
            System.out.printf("i is %d", i);
        }
        i++;
    }
}
```

\[ CC = 8 - 7 + (2 \times 1) = 3 \]

*http://www.tutorialspoint.com/software_testing_dictionary/cyclomatic_complexity.htm*
Logical Line of Code

```java
/*
 * This function is an example of comment
 */

private int expected = 15;
public void guessNumber (int guess) {
    if (guess == expected) {
        System.out.println("Yes, you are correct");
    } else {
        System.out.println("No, you guess it wrong");
    }
}
```

Logical Line of Code attempts to measure the number of executable expressions or statements.

<table>
<thead>
<tr>
<th>Physical Line of Code</th>
<th>Logical Line of Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>
Technical Debt Calculations

Debt (in man days) = cost_to_fix_duplications
+ cost_to_fix_violations
+ cost_to_comment_public_API
+ cost_to_fix_uncovered_complexity
+ cost_to_bring_complexity_below_threshold

Where:
- Duplications = cost_to_fix_one_block * duplicated_blocks
- Violations = cost_to_fix_one_violation * mandatory_violations
- Comments = cost_to_comment_one_API * public_undocumented_api
- Coverage = cost_to_cover_one_of_complexity * uncovered_complexity_by_tests (80% of coverage is the objective)
- Complexity = cost_to_split_a_method * (function_complexity_distribution >= 8) + cost_to_split_a_class * (class_complexity_distribution >= 60)
Algorithm 1: Algorithm of calculating final ratings for COCOMO II SU factors

1. Weight each question rating by task difficulty rating and student experience rating:

\[ \text{Rating}_{\text{Question}} = \text{OriginalRating}^{\text{Rating}_{\text{TaskDifficulty}} \times \text{Rating}_{\text{StudentExperience}}} \]  

(1)

2. Calculate a COCOMO II SU factor rating per week:

\[ \text{Rating}_{\text{SuperWeek}} = \frac{\sum_{n=1}^{N_{\text{numOfQuestions}}} (\text{Rating}_{\text{Question}})_n}{N_{\text{numOfQuestions}}} \]  

(2)

3. Calculate a COCOMO II SU factor rating per project:

\[ \text{Rating}_{\text{SuperProject}} = \frac{\sum_{n=1}^{N_{\text{numOfWeeks}}} (\text{Rating}_{\text{SuperWeek}})_n}{N_{\text{numOfWeeks}}} \]  

(3)

4. Repeat above steps for all COCOMO II SU factors and all projects.

5. Normalize the project ratings to a scale of 0 to 10:

\[ \text{NormalizedR}_1 = \frac{(\text{Rating}_{\text{SuperProject}})_1 - \min(\text{Rating}_{\text{SuperProject}})}{\max(\text{Rating}_{\text{SuperProject}}) - \min(\text{Rating}_{\text{SuperProject}})} \times 10 \]  

(4)

\[ \text{NormalizedR}_i = \frac{(\text{Rating}_{\text{SuperProject}})_i \times \text{NormalizedR}_{i-1}}{(\text{Rating}_{\text{SuperProject}})_{i-1}} \]  

(5)

where \((\text{Rating}_{\text{SuperProject}})_1\) is the first not minimum data point in the dataset, \(\text{Rating}_{\text{SuperProject}}\) are all the project level COCOMO II SU factor ratings and \(\text{NormalizedR}_i\) is the \(i^{th}\) normalized data.
Project and Tasks distribution

<table>
<thead>
<tr>
<th>Student</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Projects</td>
<td>11</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>Number of Finished Tasks</td>
<td>35</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>13</td>
<td>31</td>
</tr>
<tr>
<td>Number of Unfinished Tasks</td>
<td>9</td>
<td>8</td>
<td>3</td>
<td>9</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>Number of Total Tasks</td>
<td>44</td>
<td>20</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>44</td>
</tr>
<tr>
<td>% of Finished Tasks</td>
<td>79.55</td>
<td>60.00</td>
<td>75.00</td>
<td>43.75</td>
<td>65.00</td>
<td>70.45</td>
</tr>
</tbody>
</table>
Questions

Do you have any industrial experience as a software engineer?
If yes:
1. How many years of experience do you have?
2. For each job, what was your role?
3. For each job, please identify the type/nature of your work. (e.g. web development)

OSS experience:
1. Have you used any open source software? If yes, what are they?
2. Have any of the projects you worked on before (including industrial and class projects) integrated any OSS components? If yes, what are the OSS components?