Project Notes - Quality Measurement

Peter Michalski

Contents

1 Variabilities

Input Variabilities

1. boundary parameters
2. dimension
3. number of velocity directions

Calculation Variabilities

1. computational model
2. decomposition technique
3. coefficient weights
4. input check
5. encoding of output
6. exception check

Other Variabilities

1. language
2. license

2 Quality Measurement

2.1 Robustness

2.1.1 Definition

Software possesses the characteristic of robustness if it behaves "reasonably" in two situations: i) when it encounters circumstances not anticipated in the requirements specification; and ii) when the assumptions in its requirements specification are violated.

2.1.2 Metrics

Using CRASH criteria: (DOI: 10.1007/978-3-642-29032-9_16)

1. How does the system react when random but valid input is provided?

   Inject random input at interface (automated testing (selenium?) - generate list of valid input and observe if CRASH)

2. How does the system react when invalid input is provided?

   Use invalid input at interface (testing - generate list of invalid input and observe if CRASH)

3. How does the system react when required libraries are not provided?

   Not provide necessary libraries (find if external libraries are needed, not provide, observe if CRASH)

4. How does the system react when directories and/or files are mutated?

   Removing directories (mutation) (for example removal of output directory - find directories for mutation, remove, observe if CRASH)

5. How does the system react when run on OS and/or hardware that is outside of the requirement specification?

   Using OS and hardware outside of those in the reqs: (find req OS and hardware, make list outside of reqs, attempt to run the software and observe if CRASH)

2.1.3 Notes

DOI: 10.1007/978-3-642-29032-9_16:
The robustness failures are typically classified according to the CRASH criteria [540]: Catastrophic (the whole system crashes or reboots), Restart (the application has to be restarted), Abort (the application terminates abnormally), Silent

(invalid operation is performed without error signal), and Hindering (incorrect error code is returnednote that returning a proper error code is considered as robust operation). The measure of robustness can be given as the ratio of test cases that exposes robustness faults, or, from the systems point of view, as the number of robustness faults exposed by a given test suite.

2.2 Performance

2.2.1 Definition

The degree to which a system or component accomplishes its designated functions within given constraints, such as speed (database response times, for instance), throughput (transactions per second), capacity (concurrent usage loads), and timing (hard real-time demands).

2.2.2 Metrics

1. How does the application compare in CPU utilization compared to other solutions running on the same hardware?

   Measure average CPU utilization of each application using the same hardware.

2. What is the average memory usage of the application?

   Measure the average memory usage of the application using the same hardware.

3. How many concurrent usage loads can the system handle?

   Run multiple instances of the application.

4. What is the average amount of time that the system needs to provide a solution?

   Measure the average amount of time that each application takes to provide a solution.

5. How many errors does the system experience in X hours of run-time?

   Run automated tests and count the number of run-time errors.

6. How accurate, on average, are the results of the solution?

   Measure the average error of the solution when compared to a pseudo oracle.

2.2.3 Notes

2.3 Maintainability

2.3.1 Definition

The effort with which a software system or component can be modified to:

1. correct faults
2. improve performance or other attributes
3. satisfy new requirements

2.3.2 Metrics

1. Open and closed issues on Git

   Count the ratio of open to closed issues on Git

2. Average time to close issues

   Calculate the average time taken to close an issue on Git

3. Update frequency

   Count average number of days between major update releases?

4. Number of maintainers

   Count how many maintainers have worked on this project.

Metrics for Assessing a Software System's Maintainability (Oman, Hagemeister):

1. Age

   Count number of months since release.

2. Size

   Count thousands of non commented source statements (TNCSS)

3. Stability

   Calculate: Stability = 1 - Change Factor when CF is (e∧number of months * average percentage change of lines of code in number of months)

4. Defect Intensity

   Calculate: Defect intensity = e∧number of months * average percentage of defective lines of code per month

5. Subjective Product Appraisals

   Measure: Subjective Product Appraisal = 5 point (very low to very high) for language complexity, application complexity, requirements volatility, product dependencies, complexity of build, installation complexity, intensity of product use, efficiency of the software system

6. Modularity

   Measure: number of modules, average module size

7. Consistency

   Measure: std deviation of module size (TNCSS)

8. Global Data Types

   Measure: the number of global data types divided by the total number of defined data types

9. Global Data Structures

   Measure: the number of global data structures divided by the total number of defined data structures

10. Data Type Consistency

    Calculate: 1 - percentage of data structures that undergo type conversion during assignment operations

11. I/O Complexity

    Measure: the number of lines of code devoted to I/O divided by TNCSS

12. Local Data Types

    Calculate: the number of local data types divided by the total number of defined data types averaged over all modules

13. Local Data Types

    Calculate: the number of local data types divided by the total number of defined data types averaged over all modules

14. Local Data Structures

    Calculate: the number of local data structures divided by the total number of defined data structures averaged over all modules

15. Initialization Integrity

    Calculate: percentage of variables initialized prior to use averaged over all modules

16. Overall Formatting

    Calculate: percentage of blank lines in the whole program, percentage of modules with blank lines

17. Commenting

    Calculate: percentage of comment lines in program, percentage of modules with header comments

18. Statement Formatting

    Calculate: percentage of uncrowded statements (no more than one statement per line) per module averaged over all modules

19. Intramodule Commenting

    Calculate: percentage lines of comments in module, averaged over all modules

20. Subjective Evaluation of Document Descriptiveness

    Categorize based on accuracy, consistency, unambiguous

21. Subjective Evaluation of Document Completeness

    Categorize based on extent of document set, contents

22. Subjective Evaluation of Document Correctness

    Categorize based on traceability, verifiability

23. Subjective Evaluation of Document Readability

    Categorize based on organization, accessibility via indices and table of contents, consistency of the writing style, typography, and comprehensibility

24. Subjective Evaluation of Document Modifiability

    Categorize based on document set redundancies

2.3.3 Notes

Metrics for Assessing a Software System's Maintainability (Oman, Hagemeister):
Software system metrics divided into 3 categories:

1. server maturity attributes (age since release, size, stability, maintenance intensity, defect intensity, reliability, reuse, subjective product appraisals)

2. source code (control structure, information structure, typography and naming and commenting) - each of these is broken into system and component subcategories

system control structure: modularity, complexity, consistency, nesting, control coupling, encapsulation, module reuse, control flow consistency

component control structure: complexity, use of structured constructs, use of unconditional branching, nesting, span of control structures, cohesion

system information structure: global data types, global data structures, system coupling, data flow consistency, data type consistency, nesting, I/O complexity, I/O integrity

component information structure: local data types, local data structures, data coupling, initialization integrity, span of data

system typography, naming and commenting: overall program formatting, overall program commenting, module separation, naming, symbols and case

component typography, naming and commenting: statement formatting, vertical spacing, horizontal spacing, intramodule commenting

3. supporting documentation (abstraction, physical attributes)

supporting documentation abstraction: descriptiveness appraisals, completeness appraisals, correctness appraisals

supporting documentation physical attributes: readability appraisals, modifiability appraisals

2.4 Reusability

2.4.1 Definition

The extent to which a software component can be used with or without adaptation in a problem solution other than the one for which it was originally developed.

2.4.2 Metrics

2.4.3 Notes

Measuring Software Reusability (Poulin)

Taxonomy of reusability mctrics 1. Empirical methods - - Module oriented - Complexity based - Size based - Reliability based Component oriented

2.Qualitative methods - -Module oriented - Style guidelines Component oriented - Certification guidelines - Quality guidelines

3.1 Prieto-Diaz and Freeman: Their process model encourages white-box reuse and consists of finding candidate reusable modules, evaluating each, deciding which module the programmer can modify the easiest, then adapting the module. In this model they identify four module-oricntcd metrics and a fifth metric used to modify the fust four.

Program size. Reuse depends on a small module size, as indicated by lines of source code. Program structure. Reuse depends o n a simple program structure as indicated by fewer links to other modules (low coupling) and low cyclomatic complexity. Program documentation. Reuse depends on excellent documentation as indicated by a sub- jective overall rating o n a scale of 1 to 10. Programming language. Reuse depends on pro- gamming language to the extent that it helps to reuse a module written in the same programming language. If a reusable module in the same lan- guage does not exist, the degree of similarity between the target language and the one used in the module affects the difficulty of modifying the module to meet the new requirement. Reuse experience. The experience of the reuser in the programming language and in the appliea- tion domain affects the previous metrics because every programmer views a module from their own perspective. For example, programmers wtll have diffcrent views of what makes a small module, depending on their background. ?lis fifth metric serves to modify the values of the other mctrics.

—

Chidamber and Kemerer object-oriented metrics:

https://www.aivosto.com/project/help/pm-oo-ck.html:

eg weighted methods per class, number of children, coupling per class

Software reusability metrics estimation: Algorithms, models and optimization techniques (Padhy)

Cyclomatic complexity: independent paths through source code

Software Reuse and Reusability Metrics and Models (Frakes, Terry)

A new reusability metric for object-oriented software (Barnard)

A metrics suite for measuring reusability of software components (Washizaki)

Reusability Index: A Measure for Assessing Software Assets Reusability (Ampatzoglou)

2.5 Portability

2.5.1 Definition

Effort required to transfer a program between system environments (including hardware and software).

2.5.2 Metrics

2.5.3 Notes

compare effort to port to effort to redevelop

determine if the system can be ported: ram, processor, resolution, OS, browser

Issues in the Specification and Measurement of Software Portability (Mooney):
The term portability refers to the ability of a software unit to be ported (to a given environment). A program is portable if and to the degree that the cost of porting is less than the cost of redevelopment. A software unit would be perfectly portable if it could be ported at zero cost, but this is never possible in practice. Instead, a software unit may be characterized by its degree ofportability, which is a function of the porting and development costs, with respect to a specific target environment.

Some of these issues have been examined by the author [Mooney 931. This work has proposed as a metric the degree of portability of a software unit with respect to a target environment, defined as DPfsu) = 1 - (Cport(su,q) I Crd&req.e2)).
This metric relates portability to a ratio between the cost of porting (which depends on the properties of the existing software unit and on the target environment), and the cost of redevelopment (which depends on the requirements and the target environment). A series of experiments is underway to refine and validate this metric and to determine how to measure or estimate Cpoyf and Crdev. One study by Sheets [94] suggests that the metric can be badly skewed by secondary elements such as inadequate documentation for the existing software.

Designing a Measurement Method for the Portability Non-functional Requirement (Talib):
EffortNew is the total effort needed in working hours units to develop the software functionalities on a new environment

---

https://www.softwaretestinghelp.com/what-is-portability-testing/:

ISO 9126 breaks down portability testing: installability, compatability, adapatability, and replaceability.

check Installability, Adaptability, Replaceability, Compatibility or Coexistence

Installability: validate OS reqs, memory and RAM reqs, clear installation and uninstallation procedures, additional prerequisites

Adapatability: hardware and software dependency, language dependency and communication system