Project Notes - Quality Measurement

Peter Michalski

Contents

1 Variabilities

1. category
2. dimensions
3. computational model
4. decomposition technique
5. exception check
6. language
7. license
8. parallel
9. turbulent
10. complex geometry
11. non-Newtonian fluids
12. dependencies

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

Review my notes "Is this GitHub Project Maintained? Measuring the Level of Maintenance Activity of Open-Source Projects" and "Longitudinal Evaluation of Open-Source Software Maintainability"

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

Poulin (non-direct source):

1. Size

   Count lines of source code.

2. Structure

   Count links to other modules.

3. Documentation

   Subjective rating on a scale of 1 to 10.

4. Language

   Subjective rating based on how common the language is.

5. Simple interfaces

   Count the average number of passed variables to modules.

6. Few calls to other modules

   Count the average number of calls to other modules.

7. Cyclomatic complexity

   Measure cyclomatic complexity

8. Self descriptiveness

   Measure average percentage of comment lines in module

9. Number of problem reports

   Measure average number of outstanding defects in the module.

Chidamber and Kemerer:

1. Methods per class

   Measure the average number of methods defined in each class (want to keep low).

2. Number of children

   Measure the number of immediate child classes derived from a base class (wan this to be high).

3. Coupling between object classes

   Measure the average number of methods or variables per class that are used by other classes (want to keep this low).

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

1. Methods per class (WMC)

   Measure the number of methods per class.

2. Depth of inheritance tree (DIT)

   Measure the highest length from the rood to the node.

3. Number of children (NOC)

   Measure the average total number of total number of sub-classes.

4. Method complexity (MC)

   Measure the average total number of methods present in a class.

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

1. Coupling (calls to foreign classes)
2. Number of methods
3. Number of attributes
4. Meaningful name
5. Documentation

6. Lines of code
7. Comment lines
8. Depth of inheritance
9. Number of children
10. Cohesion
11. Class variables
12. Complexity of data structure
13. Interface parameters
14. Calls to foreign classes
15. Calls to library classes
16. Access (public, protected, private)

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

1. Number of classes in a module
2. Number of properties
3. Interface complexity
4. Documentation quality
5. Number of open bugs

2.4.3 Notes

Measuring Software Reusability (Poulin)

Taxonomy of reusability metrics:
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)

Chidamber and Kemerer object-oriented metrics (https://www.aivosto.com/project/help/pm-oo-ck.html):

The Chidamber and Kemerer metrics suite originally consists of 6 metrics calculated for each class: WMC, DIT, NOC, CBO, RFC and LCOM1. The original suite has later been amended by RFC, LCOM2, LCOM3 and LCOM4 by other authors.

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

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

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

Same as installability?

1. OS platforms

   Measure the average time (or steps taken) to port the program between a list of OS environments.

2. OS versions

   Measure the average time (or steps taken) to port the program between a list of OS versions.

3. Hardware platforms

   Measure the average time (or steps taken) to port the program between a list of hardware platforms.

4. Number of external libraries needed

   Count the number of external libraries needed to set up the environment.

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.

The principal types of portability usually considered are binary portability (porting the executable form) and source portability (porting the source language representation). Binary portability offers several advantages, but is possible only across strongly similar environments.Source portability assumes availability of source code, but provides opportunities to adapt a software unit to a wide range of environments. In this paper we consider only source portability.The porting process has two principal components which we call transportation and adaptation. Transportation is physical movement; this may not be trivial since compatible media must be used and various types of representation conversion may be required. Adaptation is any modification that must be performed on the original version; we take this to mean both mechanical translation such as by language processors, and manual modification by humans

Designing a Measurement Method for the Portability Non-functional Requirement (Talib):

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