Assignment: Code complexity, coverage

1 Goal

In this assignment, you will experiment with complexity and coverage metrics. The goals are to get an understanding and appreciation of the benefits and drawbacks of metrics and their tools, and to create new test cases or to enhance existing tests that improve statement or branch coverage. You will perform these tasks on an open-source project of reasonable size and complexity (see below).

Deliverables

You will fork or clone the source code repository of the project of your choice, and perform modifications there. The URL of the cloned repository will be submitted on Canvas. You can use the report template from Canvas. You are free to adapt it or use a different format as long as you cover the same content. The report is uploaded as a file.

2 Open source metrics tools

We recommend lizard as tool to measure metrics. It supports

- C/C++ (incl. C++14),
- Java,
- C# (C Sharp),
- JavaScript,
- Objective-C,
- Swift,
- Python,
- Ruby,
- TTCN-3,
- PHP,
- Scala,
- GDScript,
- Golang,
- Lua.

The github repository of the project has the latest version, but we recommend using the latest stable release. The program has a simple command line interface, which gives you an immediate overview of key metrics such as NCLOC (non-comment lines of code) and CCN (cyclomatic complexity number).

If you happen to use a programming language that is not supported by lizard and for which you cannot find a measurement tool, you can use the command "`wc -l`" to count the lines of code (including comments) per file.

Open source coverage measurement tools

Several code coverage tools for Java are available. A shortlist:

- Cobertura: Support for ant, command line, maven.
- OpenClover: Support for ant, maven, grails, Eclipse, IntelliJ.
- jacoco: Integrated with Eclipse.

For Python, we recommend Coverage.py. For gcc users, gcov is well integrated with the compiler itself.

3 Tasks

3.1 Part 0: Project choice

Choose a project, or a module of a project, written in a programming language you would like to work with. We will only consider code written in that language. Use an open source project which

- uses an open source initiative-approved license;
- uses automated unit or system testing on the part written in your language of choice;

- existing branch coverage (by the given tests) must be clearly less than 100 %;
- has at least two contributors outside your own course group;
- the programming language must be a Turing-complete language, not a markup language; and
- has at least 10,000 lines of code (10 K LOC); this applies to the programming language you are using, as counted by "`cloc`"; files in other languages than the one you use do not count.

Please register your project in the project sheet (see Canvas for the link). Note that each group must work on a different project! If two or more groups end up choosing the same large project, you can choose different sub-modules, as long as they are larger than 10 K lines each.

3.2 How to find a project

Note: Before you dive in deep, ensure that the project fulfills the requirements of this lab! The links below are just suggestions. Lists of projects may also include projects that are not suitable for this assignment; you are free to use your own choice as long as the criteria above are fulfilled.

1. Choose a project you are using yourself (e.g., a program or library).
2. Choose from a list of beginner-friendly projects. **Note:** A lot of these projects are too small for this assignment, so count the lines of code first before you start!
3. Choose an Apache project. Choose from this list. Note that these projects have not been fully vetted for test automation, so please ensure you can automatically run all necessary tests.

Other suggestions:

- NASA Open Source: over 350 space-related open source projects (not all of which are big/active enough to be suitable!), using different programming languages.
- LyX, a document editor with a LaTeX back-end (C++ with GUI).
- RTEMS, a light-weight real-time OS (C). Warning: This is a difficult project because it requires building the whole cross-compilation tool chain itself from scratch.
- Wesnoth, a turn-based strategy game (mostly C++, some Lua).

3.3 Onboarding

How good is the "onboarding" documentation?

1. How easily can you build the project? Briefly describe if everything worked as documented or not:
   (a) Did you have to install a lot of additional tools to build the software?
   (b) Were those tools well documented?
   (c) Were other components installed automatically by the build script?
   (d) Did the build conclude automatically without errors?
   (e) How well do examples and tests run on your system(s)?
2. Do you plan to continue or choose another project?

3.4 Part 1: Complexity measurement

Run a complexity measurement tool such as lizard on the code base. If you cannot find a tool for your platform, run "`wc -l`" to measure the lines of code per file, and count the lines of code for some large functions using a text editor.

Count the cyclomatic complexity of large functions of the code (not necessarily the largest ones, in case you cannot sort them automatically by size). Exclude third-party code and generated code in this. If you have a tool, list ten (10) functions or methods with high complexity. In addition to that (or if you have no tool), count the cyclomatic complexity of *five* complex functions by hand, either by choosing five out of the ten functions shown by the tool, or, if no tool was available, five functions that appear to be complex.

Note: This document assumes your group has five members; if your group is smaller, total numbers scale with the group size: 8 functions with four active group members, or six with three active members.

Note: Use at least two group members to count the complexity separately, to get a reliable results. Use a third member if the two counts differ.

1. What are your results? Did everyone get the same result? Is there something that is unclear? If you have a tool, is its result the same as yours?
2. Are the functions/methods with high CC also very long in terms of LOC?
3. What is the purpose of these functions? Is it related to the high CC?
4. If your programming language uses exceptions: Are they taken into account by the tool? If you think of an exception as another possible branch (to the catch block or the end of the function), how is the CC affected?
5. Is the documentation of the function clear about the different possible outcomes induced by different branches taken?

3.5 Part 2: Coverage measurement & improvement

Identify one or two code coverage tools that work for your platform and use them on your chosen project.

3.5.1 Task 1: DIY

Implement branch coverage by manual instrumentation of the source code for ten functions with high cyclomatic complexity. Use a separate development branch or repo for this, as this code is not permanent. The simplest method for this is as follows:

1. Identify all branches in the given functions; assign a unique number (ID) to each one.
2. Before the program starts (before the first instruction in "main" or as the first step in the unit test harness), create data structures that hold coverage information about specific branches.
3. Before the first statement of each branch outcome (including to-be-added "else" clauses if none exist), add a line that sets a flag if the branch is reached.
4. At the end of the program (as the last instruction in "main" or at the end of all unit tests), write all information about the taken branches taken to a file or console.

Note: It is perfectly fine if your coverage tool is inefficient (memory-wise or time-wise) and its output may be hard to read. For example, you may allocate 100 coverage measurement points to each function, so you can easily divide the entries into a fixed-size array.

1. What is the quality of your own coverage measurement? Does it take into account ternary operators (condition ? yes : no) and exceptions, if available in your language?
2. What are the limitations of your tool? How would the instrumentation change if you modify the program?
3. If you have an automated tool, are your results consistent with the ones produced by existing tool(s)?

3.5.2 Task 2: Coverage improvement

In the ten functions with high cyclomatic complexity, what is the current branch coverage? Is branch coverage higher or lower than in the rest of the code (if you have automated coverage)?

Keep a record (copy) of your coverage before you start working on new tests. Furthermore, **make sure you add the new tests on a different branch** than the one you used for coverage instrumentation.

Having identified "weak spots" in coverage, try to **improve coverage with additional test cases.**

1. Identify the requirements that are tested or untested by the given test suite.
2. Document the requirements (as comments), and use them later as assertions.
3. Create new test cases as needed to improve branch coverage in the given functions. Can you call the function directly? Can you expand on existing tests? Do you have to add additional interfaces to the system (as public methods) to make it possible to set up test data structures?

4. If you have 100 % branch coverage, you can choose other functions or think about *path coverage.* You may cover all branches in a function, but what does this mean for the combination of branches? Consider the existing tests by hand and check how they cover the branches (in which combinations).

3.5.3 Task 3: Refactoring plan

Is the high complexity you identified really necessary? Is it possible to split up the code (in the five complex functions you have identified) into smaller units to reduce complexity? If so, how would you go about this? **Document your plan.**

4 Grading criteria

4.1 Pass (P)

1. You identify ten functions/methods with high complexity, and document the purpose of them, and why the complexity should be high (or not).
2. You manually count the complexity of five functions (on paper or as comments).
3. You clearly describe how to refactor five complex functions into smaller functions.
4. The purpose of each of the high-complexity methods is documented in detail w.r.t. the different outcomes resulting in branches in the code.
5. You create a working ad-hoc coverage tool that at least measures coverage of normal branches (if, while) for ten functions.
6. Each group member writes at least two new tests that improve coverage, as evidenced by the coverage tool. All tests have at least one meaningful assertion (other than "assert(true)" or equivalent).

4.1.1 Presentation

In the grading session, the group has to

- present some of the complex functions in the grading session,
- show complexity metrics and explain their ramifications, and
- show coverage before and after the new tests (measured by external tools and their own tool).

Furthermore, the coverage measurement implementation and some of the new test cases will be inspected.

4.2 Pass with distinction (P+)

Note: It is possible to achieve a P+ individually in this project, as long as the whole group passes the assignment. In this case, the points below apply individually. Use "Assignment #3, extra coverage" to submit your solution, and leave a comment on Canvas or in the statement of contributions.

4.2.1 3 out of these 5 points:

1. Each group member writes at least four new or enhanced unit tests (twice as many as for P).
2. You use your issue tracker and systematic commit messages to manage your project.
3. You carry out some of Task 3: you refactor at least two of the functions with high cyclomatic complexity to reduce it by at least 35 % (for example, by splitting the functions).
4. You get a patch with new tests or a refactoring accepted. Note: the patch must be submitted by the assignment deadline, but it may be accepted later; as long as it is accepted by the end of the course, this point is counted. (Please notify us if this extra point is necessary for a P+.)
5. Something else that is extraordinary, at the discretion of the examiner.

5 Ethics

Recall that the student code of conduct forbids any kind of plagiarism, between groups in the course, or based on content taken on the internet.