AST-based source code analysis and visualization

Melinda Simon

Developing complex software systems is a long-term, difficult process which requires a lot of planning. We have to guarantee the quality of the system, which means in this case that the software works as it is expected and satisfies the needs of the procurers. To lower the costs of testing and avoiding later bugfixes, the system is continously analyzed by automated testing during the development from the very beginning. These tests can be unit tests, which can validate and/or verify the implementation of single unit (e.g. a class or a function) or integration test cases, which investigates the cooperation between different components of the system [2].

Although we have some good tools to use when writing unit tests (JUnit [3] and other JUnit-based tools), the data used for testing have to be chosen manually. To ease our work we can test and visualize the code coverage of our unit test by various open-source frameworks, such as Coverclipse or CodeCover and a lot more, which shows how much of a software's source code has been tested and can point to the weaknesses of testing in our project.

Similarly to unit tests, we would like to see the code coverage when running integration test to highlight the riskious parts of the project. There are several tools to run integration tests and some of these can count information on integration tests' code coverage, e.g. Sonar project [4]. It is not enough to have a good code coverage by integration tests if we have a poor result from unit tests' code coverage. Having code coverage for both unit tests and integration tests, we have to combine these informations and use a new metrics to indicate the quality of the developed code and it's test suites. For this purpose I created a simple Eclipse-plugin, which analyzes the Abstract Syntax Tree (AST) [1], counts the code coverage from calls started from a different component of the system. This plugin can visualize the colored AST and print a report containing various information about the code coverage when a call comes from a different module, e.g. which methods were involved, what parts of these methods were traversed how many times, and so on. Know the weak points of our integration tests we can improve our test cases and grant the quality of the software more effectively.

References

[1] Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D.: "Compilers: Principles, Techniques, and Tools", 2nd Edition, Addison-Wesley (2006)

[2] Perry, W. E.: "Effective Methods for Software Testing", 2nd Edition, John Wiley and Sons, (2000)

[3] JUnit.org Resources for Test Driven Development, `http://junit.org`

[4] Sonar Project, `http://www.sonarsource.org`