Agitar™
TECHNOLOGIES

Measuring Risk to Improve Java Software Quality

Introduction

The resources available to ensure the quality of a software project are nearly always limited. So it's important to know the potential for a certain method, class or project to possess quality problems in order to prioritize testing, reengineering and bug fixing efforts. The AgitarOne Management Dashboard reports a Risk Metric, which objectively measures the risk of software as a function of two critical factors:

- code complexity
- test coverage

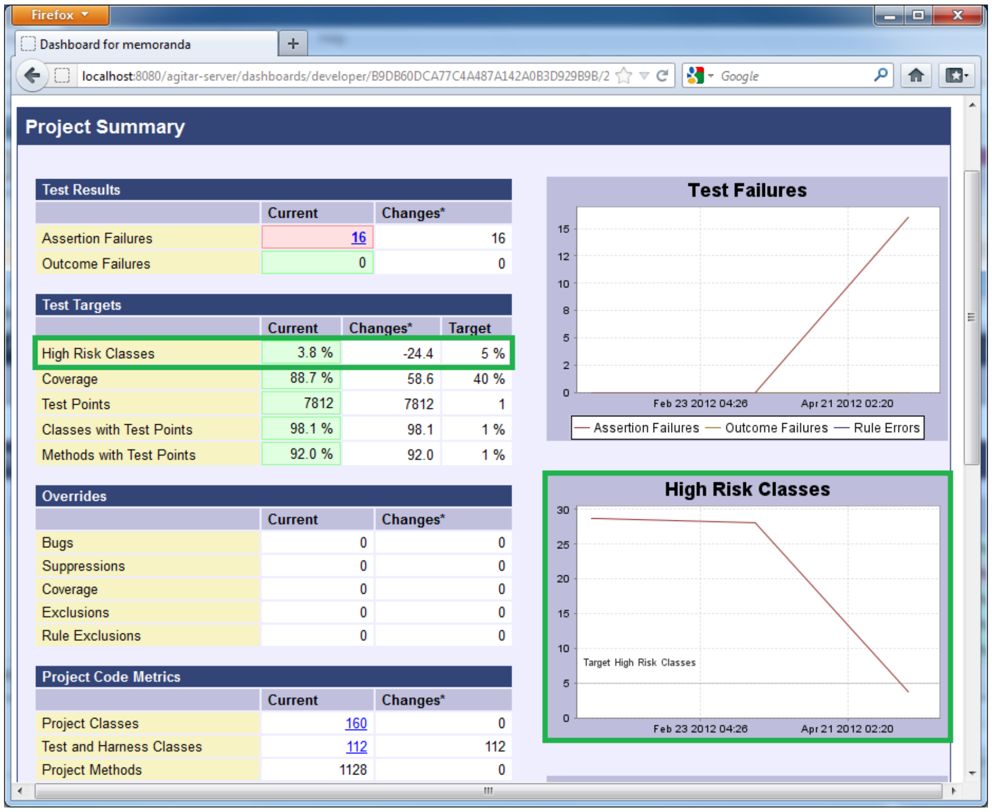
With risk metric values available, the development team can then focus its efforts on the classes, methods and projects where risk is highest. This approach results in the highest possible quality levels while ensuring the best use of scarce development and testing resources.

Challenge of managing legacy code

Most software projects have a wide variation of software risk, in terms of the complexity and the test coverage of their classes and methods. Additionally, the high risk areas may not be well known to management. Those classes that have been modified since the last release or test cycle may be particularly vulnerable, but especially so if they are of high risk relative to complexity and test coverage. Modifying existing code of moderate or high complexity without thorough testing runs a higher risk of introducing regressions.

The problem with many current development and maintenance efforts is that there is no easy way to identify the risk of each area of the code. This lack of transparency in turn makes it difficult to prioritize quality improvement efforts. Without a process to point out where risk is high, it's hard to determine what code needs work. Chances are that a considerable portion of the resources that are expended on quality improvement are wasted on low-risk code while at the same time not nearly enough effort is expended on high-risk code.

Risk metric determines which areas of the code need work



Project Summary

Test Results

	Current	Changes*
Assertion Failures	16	16
Outcome Failures	0	0

Test Targets

	Current	Changes*	Target
High Risk Classes	3.8 %	-24.4	5 %
Coverage	88.7 %	58.6	40 %
Test Points	7812	7812	1
Classes with Test Points	98.1 %	98.1	1 %
Methods with Test Points	92.0 %	92.0	1 %

Overrides

	Current	Changes*
Bugs	0	0
Suppressions	0	0
Coverage	0	0
Exclusions	0	0
Rule Exclusions	0	0

Project Code Metrics

	Current	Changes*
Project Classes	160	0
Test and Harness Classes	112	112
Project Methods	1128	0

Figure 1: Dashboard project summary

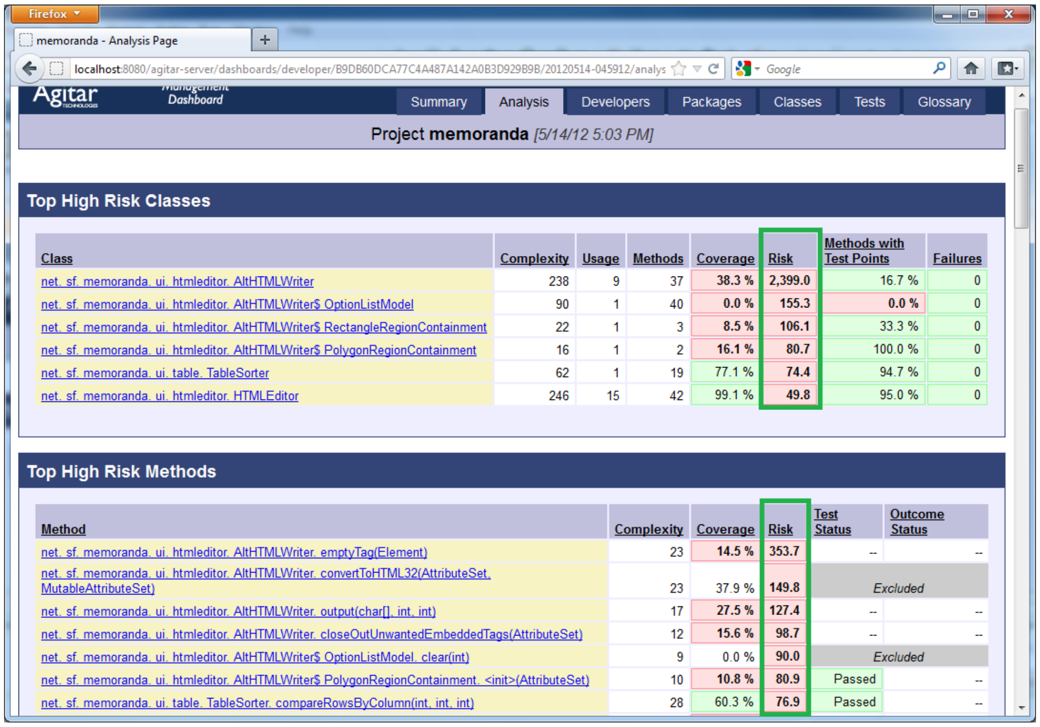
The AgitarOne Management Dashboard provides a series of metrics that enable executives, developers and development managers to understand and track the quality status of Java software projects. It calculates the complexity of each method using McCabe's cyclomatic complexity algorithm, which measures decisions and thereby also provides an indication of a minimum number of paths to test for increased reliability. The more complex the structure of code, the harder it is to understand, the longer it will take to produce, the more likely it is to have a defect, the harder it is to change, the more difficult it will be to reuse and the harder it is to test.

An independent researcher who was originally skeptical about the value of this metric, Richard Sharpe, performed a historical analysis of tens of thousands of source code files. For each file, he applied cyclomatic complexity metrics and analyzed the defect rates. The results showed a near-linear correlation between cyclomatic complexity and defect rates. For example, files having a cyclomatic complexity value of 11 had the lowest probability of being fault-prone (28%) while files containing cyclomatic complexity values of 74 and up had a 98% plus probability of being fault-prone.

The AgitarOne Management Dashboard also calculates test coverage by counting the lines, conditions, and exit paths of a method that are tested and providing the percentage that are tested. The Management Dashboard then calculates the method's risk as the complexity of the method mitigated by test coverage. The method risk calculation for a method m is:

$$\text{Risk}_m=\text{complexity}_m^{\,2} \times (1-\text{coverage}_m/100)^3 + \text{complexity}_m$$

The risk metric provides an estimate of the reliability of code. Code that is overly complex or insufficiently tested is more prone to breaking at the slightest touch. The risk metric can be used to focus code improvement and/or testing efforts.


Firefox
memoranda - Analysis Page
localhost:8080/agitar-server/dashboards/developer/B9DB60DCA77C4A487A142A0B3D929B9B/20120514-045912/analys
Google
Agitar
Management Dashboard
Summary
Analysis
Developers
Packages
Classes
Tests
Glossary

Project **memoranda** *[5/14/12 5:03 PM]*

Top High Risk Classes

Class	Complexity	Usage	Methods	Coverage	Risk	Methods with Test Points	Failures
net. sf. memoranda. ui. htmleditor. AltHTMLWriter	238	9	37	38.3 %	2,399.0	16.7 %	0
net. sf. memoranda. ui. htmleditor. AltHTMLWriter$ OptionListModel	90	1	40	0.0 %	155.3	0.0 %	0
net. sf. memoranda. ui. htmleditor. AltHTMLWriter$ RectangleRegionContainment	22	1	3	8.5 %	106.1	33.3 %	0
net. sf. memoranda. ui. htmleditor. AltHTMLWriter$ PolygonRegionContainment	16	1	2	16.1 %	80.7	100.0 %	0
net. sf. memoranda. ui. table. TableSorter	62	1	19	77.1 %	74.4	94.7 %	0
net. sf. memoranda. ui. htmleditor. HTMLEditor	246	15	42	99.1 %	49.8	95.0 %	0

Top High Risk Methods

Method	Complexity	Coverage	Risk	Test Status	Outcome Status
net. sf. memoranda. ui. htmleditor. AltHTMLWriter. emptyTag(Element)	23	14.5 %	353.7	--	--
net. sf. memoranda. ui. htmleditor. AltHTMLWriter. convertToHTML32(AttributeSet, MutableAttributeSet)	23	37.9 %	149.8	*Excluded*	
net. sf. memoranda. ui. htmleditor. AltHTMLWriter. output(char[], int, int)	17	27.5 %	127.4	--	--
net. sf. memoranda. ui. htmleditor. AltHTMLWriter. closeOutUnwantedEmbeddedTags(AttributeSet)	12	15.6 %	98.7	--	--
net. sf. memoranda. ui. htmleditor. AltHTMLWriter$ OptionListModel. clear(int)	9	0.0 %	90.0	*Excluded*	
net. sf. memoranda. ui. htmleditor. AltHTMLWriter$ PolygonRegionContainment. <init>(AttributeSet)	10	10.8 %	80.9	Passed	--
net. sf. memoranda. ui. table. TableSorter. compareRowsByColumn(int, int, int)	28	60.3 %	76.9	Passed	--

Figure 2: Risk reporting in Dashboard

The AgitarOne Management Dashboard reports risk for methods, classes, developers, packages and projects as shown in Figure 2. For example, when reporting class risk, it takes the risk above the threshold for each method in the class and adds it together. It then multiplies this sum by a usage factor. Methods that are not high risk do not contribute to the class risk. The Dashboard analyzes the code and produces reports that list the number of classes, how complex the different classes and methods are, what is the level of testing coverage for the different classes and methods, and what is the level of risk for each class and method.

Testing goals can be defined and tracked for the project and for individual classes, and then summarized for each developer and for the overall project. Summary status information is provided at the individual project level or rolled up over several projects. Key metrics such as the overall code coverage, total number of tests, percentage of classes and methods with direct tests and number of test failures are plotted on trend charts for easy analysis. The Dashboard's e-mail notification delivers key metrics to managers' and developers' inboxes.

Improving the quality of your code

The AgitarOne Management Dashboard provides a targeted, measured and automated way to get a clear overview of your developer testing effort. The Dashboard enables developers, development managers and corporate managers to track quality status of software projects. Objective metrics help set priorities and measure progress. Non-behavior changing refactoring patterns can be used to reduce code complexity. Code complexity and risk analysis reports can be used to identify which classes would benefit the most from additional coverage and more comprehensive testing.

The AgitarOne Management Dashboard provides a roadmap to continuously improve the quality of your code. You might start by finding the highest risk class or method and take a look at why it is prone to break. Is the problem primarily the complexity of the code or the lack of test coverage or a combination of both? Look at why your tests do not cover problematic lines or branches. Is the code too complex or too hard to observe? Either write more tests to increase coverage or employ non-behavior changing refactoring patterns to make the code easier to test. Metrics targets can be set just below current numbers so if code is added, tests fail, risk increases or test points decrease, the build will fail immediately.

As developers incorporate objective metrics into their workflow, it will become possible to detect and fix bugs sooner, deliver tests with each class and design better software. Developers can track risk metrics over time to generate continual improvements in code quality. Over time, code coverage will approach 100%, the quality of tests will improve and the body of tests will be able to detect nearly every unit-level bug. As the application changes and new features are added, tests can be kept synchronized with the updated code. Software teams will be able to refine and expand the code base and be confident that these changes will not degrade code quality.

Conclusion

The AgitarOne Management Dashboard provides a measurement of complexity and test coverage to identify code that is likely to produce regression bugs that will not be caught by your current tests if it is changed. The Dashboard enables teams to set priorities based on risk, establish targets for both the team and individual developers, measure progress, correctly allocate resources and continuously improve software quality.

For more information about AgitarOne, please visit us at www.agitar.com.