Risk-based Object Oriented Testing

Linda H. Rosenberg, Ph.D.	Ruth Stapko	Albert Gallo
NASA GSFC	SATC NASA, Unisys	SATC NASA, Unisys
Code 302	Code 300.1	Code 300.1
Greenbelt, MD 20771	Greenbelt, MD 20771	Greenbelt, MD 20771
301-286-0087	301-286-0101	301-286-8012
Linda.Rosenberg@gsfc.nasa.gov	Rstapko@pop300.gsfc.nasa.gov	Al.Gallo@gsfc.nasa.gov

Software testing is a well-defined phase of the software development life cycle. Functional ("black box") testing and structural ("white box") testing are two methods of test case design commonly used by software developers. A lesser known testing method is risk-based testing, which takes into account the probability of failure of a portion of code as determined by its complexity. For object oriented programs, a methodology is proposed for identification of risk-prone classes.

Risk-based testing is a highly effective testing technique that can be used to find and fix the most important problems as quickly as possible. Risk can be characterized by a combination of two factors: the severity of a potential failure event and the probability of its occurrence. Risk can be quantified by using the equation

$$\text{Risk} = \sum p(E_i) * c(E_i),$$

Where i =1,2,…,n. n is the number of unique failure events, E_i are the possible failure events, p is probability and c is cost.

Risk-based testing focuses on analyzing the software and deriving a test plan weighted on the areas most likely to experience a problem that would have the highest impact [McMahon]. This looks like a daunting task, but once it is broken down into its parts, a systematic approach can be employed to make it very manageable.

The severity factor $c(E_i)$ of the risk equation depends on the nature of the application and is determined by domain analysis. For some projects, this might be the critical path, mission critical, or safety critical sections. Severity assessment requires expert knowledge of the environment in which the software will be used as well as a thorough understanding of the costs of various failures. Musa addresses how to estimate the severity of software failures in the discussion of "Operational Profiles" in his book, *Software Reliability Engineering*. Both severity and probability of failure are needed before risk-based test planning can proceed. Severity assessment is not addressed here because it involves so much application-specific knowledge. Instead we confine the remainder of the discussion to the first part of the risk equation, ranking the likelihood of component failures, $p(E_i)$, and a way to capture the information directly from the source code, independent of domain knowledge.

The first task of risk-based testing is to determine how likely it is that each part of the software will fail. It has been proven that code that is more complex has a higher

incidence of errors or problems [Pfleeger]. For example, cyclomatic complexity has been demonstrated as one criterion for identifying and ranking the complexity of source code [McCabe]. Therefore, using metrics to predict module failures might simply mean identifying and sorting them by complexity. Then using the complexity rankings in conjunction with severity assessments from domain risk analysis would identify which modules should get the most attention. But module complexity is a univariate measure, and it could fail to detect some very risk-prone code. In particular, object oriented programming can result in deceptively low values for common complexity metrics. The nature of object oriented code calls for a multivariate approach to measure complexity [Rosenberg].

We are going to narrow the topic further and focus specifically on object oriented software. The Software Assurance Technology Center (SATC) at NASA Goddard Space Flight Center has identified and applied a set of six metrics for object oriented design measurement. These metrics have been used in the evaluation of many NASA projects and empirically supported guidelines have been developed for their interpretation. The metrics are defined as follows:

1. Number of Methods is a simple count of the different methods in a class.
2. The Weighted Methods per Class (WMC) is a weighted sum of the methods in a class [Chidamber]. If the weights are all equal, this metric is equivalent to the Number of Methods metric. The Cyclomatic Complexity [McCabe] is used to evaluate the minimum number of test cases needed for each method. Weighting the methods with their complexities yields a more informative class metric.
3. Coupling Between Objects (CBO) is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends [Chidamber]. Coupled classes must be bundled or modified if they are to be reused.
4. The Response for a Class (RFC) is the cardinality of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class [Chidamber].
5. Depth in Tree (DIT) - The depth of a class within the inheritance hierarchy is the number of jumps from the class to the root of the class hierarchy and is measured by the number of ancestor classes. When there is multiple inheritance, use the maximum DIT.
6. Number of Children (NOC) - The number of children is the number of immediate subclasses subordinate to a class in the hierarchy.

Having defined the metrics, we need interpretation guidelines to assist in identifying those areas of code deemed to be at high risk. For over three years, the SATC has been collecting and analyzing object oriented code written in both C++ and Java. Over 20,000 classes have been analyzed, from more than 15 programs. The results of the analyses have been discussed with project managers and programmers to identify threshold values

that do a good job of discriminating between "solid" code and "fragile" code.* Once the individual metric thresholds were determined, analysis revealed that a multivariate approach provided an excellent basis for planning risk-based testing.

When we first began to apply some of the traditional metrics to object oriented code, we saw that their values were generally much lower than we were accustomed to seeing for functionally written code. Judging by the old thresholds, the OO code appeared to be much less complex and much more modular than the non-OO legacy code. But because of the fundamentally different way an OO system is built, the low numbers were often very deceptive – ignoring the interactions between classes, and missing the complexities due to the use of inheritance. The following threshold values for the individual metrics were derived from studying the distributions of the metrics collected.

- Number of methods (NOM) - ≤ 20 preferred, ≤ 40 acceptable per class. The counting tool included explicit constructors and destructors in the method counts, so these thresholds are inflated. Taking that into account, the recommended number of actual implemented methods translates to under 10 per class.
- Weighted Methods per Class (WMC) - ≤ 25 preferred, ≤ 40 acceptable. The number of methods and the complexity of those methods are a predictor of how much time and effort is required to develop and maintain the class. While the NOM may be inflated by the beneficial use of constructors, WMC provides a better idea of the true total complexity of a class.
- Response for Class (RFC) - ≤ 50. We have seen very few classes with RFC over 50. If the RFC is high, this means the complexity is increased and the understandability is decreased. The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class, complicating testing and debugging. Making changes to a class with a high RFC will be very difficult due to the potential for a ripple effect.
- RFC/NOM ≤ 5 for C++, ≤. 10 for Java. This adjusted RFC metric does a good job of sifting out classes that need extensive testing, according to developer feedback. The Java language enforces the use of classes for everything, which automatically drives up the value of this metric.
- Coupling Between Objects (CBO) - ≤ 5. A high CBO indicates classes that may be difficult to understand, reuse or maintain. The larger the CBO, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Low coupling makes the class easier to understand, less prone to errors spawning, promotes encapsulation and improves modularity.
- Depth in Tree > 5 means that the metrics for a class probably understate its complexity. DIT of 0 indicates a "root"; the higher the percentage of DIT's of 2 and 3 indicate a higher degree of reuse. A majority of shallow trees (DIT's < 2) may represent poor exploitation of the advantages of OO design and inheritance. On the other hand, an abundance of deep inheritance (DIT's > 5) could be overkill, taking great advantage of inheritance but paying the price in complexity. When there is such

*It should be noted that the values of some of the OO metrics depend just as much on the design as they do on the actual coding. Much of the complexity of an OO system is fully determined before the programmers begin to write the code. Design complexity measurement is another topic that deserves researchers' attention.

liberal use of inheritance, the aforementioned class metrics will understate the complexity of the system.

- Number of Children (NOC) The greater the number of children, the greater the likelihood of improper abstraction of the parent and need for additional testing, but the greater the number of children, the greater the reuse since inheritance is a form of reuse. While there is no "good" or "bad" number for NOC, its value becomes important when a class is found to have high values for other metrics. The complexity of the class is passed on to all of its child classes and total system complexity is greater than it seemed at first glance.

A single metric should never be used alone to evaluate code risks, it takes at least two or three to give a clear indication of potential problems. Therefore, for each project, the SATC creates a table of high risk classes. High risk is identified as a class that has at least two metrics that exceed the recommended limits. Table 1 is an example of information that would be given to a project. The classes that exceed the expected limits are shaded.

Class	**# Methods**	**CBO**	**RFC**	**RFC/NOM**	**WMC**	**DIT**	**NOC**
Class 1	54	8	536	9.9	175	1	0
Class 2	7	6	168	24	71	4	0
Class3	33	4	240	7.2	105	2	0
Class7	54	8	361	6.7	117	2	2
Class8	62	6	378	6.1	163	2	0
Class 10	63	7	235	3.7	156	2	0
Class 11	81	10	285	3.5	161	2	0
Class 12	42	5	127	3.0	69	3	0
Class 14	20	17	324	16.2	139	4	4
Class 18	46	5	186	4.0	238	1	3

Table 1 : High Risk Java Classes

The purpose of the above information is to identify the classes at highest risk for error. While there is insufficient data to make precise ranking determinations, there is enough information to justify additional testing of classes which exceed the recommended specifications. It is up to the project to determine the criticality of these and the other classes to make the final determination on testing. Allocating testing resources based on these two factors, severity and likelihood of failures, amounts to risk-based testing.

Object oriented software metrics can be used in combination to identify classes that are most likely to pose problems for a project. The SATC has used the data collected from thousands of object oriented classes to determine a set of benchmarks that are effective in identifying potential problems. When problematic classes are also identified by domain experts as critical to the success of the project, testing can be allocated to mitigate risk. Risk-based testing will allow developers to find and fix the most important software problems earlier in the test phase.

References

Chidamber S.R. & Kemerer, C.F., “Towards a Metrics Suite for Object Oriented Design"” Proc. OOPSLA, 1991.

Li, W. & Henry, S., “Maintenance Metrics for the object Oriented Paradigm”, 1st Int’l. Software Metrics Symposium, Baltimore MD, 1993.

McCabe, Thomas J., “A Complexity Measure*”, IEEE Transactions on Software Engineering* SE-2, pp 308-320, 1976

McMahon, Keith, "Risk Based Testing", ST Labs, WA, 1998.

Pfleeger, S.L. and Palmer, J.D., “Software Estimation for Object Oriented Systems,” Int’l. Function Point Users Group Fall conference, San Antonio TX, 1990

Rosenberg, Linda, and Gallo, Albert, "Implementing Metrics for Object Oriented testing", Practical Software Measurement Symposium, 1999.