Risk Measurement for the Real (and Imperfect) World

Darryl Nicholson
ContactDarrylNicholson@gmail.com

Abstract

Walk a mile in my in my shoes. Calibrated risk taking is one of your core skills. Time to market is a measure of success. Your product is not tangible but rather a service you provide. Your skills as a QA engineer are primarily measured by your ability to accurately articulate the risk of a production release in a Software as a Service (SaaS) environment. Your clients are fixated on schedules; yet they demand and expect the delivery of high quality software as a matter of course. You are responsible for risk management and release readiness all the while knowing that your QA team's core competency is judged by how well the team gauges what is "just enough" regression testing. In such an environment, how do you define your methodology so that you can determine acceptable risk?

Do you find my shoes comfortable or are you developing blisters? Well, we routinely deliver software which successfully navigates between the financial dictates of swiftness to market and the professional QA engineer's passion for perfection. This paper will outline how our software development life cycle copes with the new QA reality of SaaS. I will specify how we use software tools and procedures to measure, define and mitigate risk. I will outline how we traverse our path to production and balance speed with quality.

"Just enough" regression testing relies on the idea that there exists a "Minimum Regression Set" (MRS) that is quantifiable and that deterministically exercises the System Under Test (SUT). In my paper, I will explain the whys, the hows and specifics of this methodology. MRS is defined and measured per production release and allows you to evaluate the regression risk of any given software deployment at any point in time. Basically, the risk statement is the percentage of executed test cases derived from calibrated set where one measures API and code coverage. This paper outlines the methodologies used to reduce the MRS to smallest set of test cases needed to provide "just enough" testing so you can deliver high quality software as quickly as possible…ahead of your competitors.

Biography

Darryl Nicholson has been VP of QA & Programs at Vesta Corporation for the past five years where he has morphed the traditional QA department into a modern, risk-tolerant culture focused on service delivery. He has the conflicting responsibilities of both Quality Assurance (where he is measured by the quality of the code release) and Project Management (where he is measured by time to market). Fascinated with the business drivers of software development, he has a unique approach to QA. A veteran of four startups (Preside, Oresis, Q-Optics/Elematics, Polyserve), Darryl is a strong Telecom Engineer with solid software QA and sales engineering expertise.

Darryl has a Computer Engineering degree from the Royal Military College (RMC) of Canada.

1 Introduction

Risk. A simple word that drives an industry, fosters intense discussions, generates philosophical discussions and creates and destroys entire careers. Risk management defies mathematical precision, challenges perceptions and causes endless debates.

This paper is about regression testing and how careful calibration of your product test plan can drive risk management for any given software release. It's about how to define a quantifiable Minimum Regression Set (MRS) that deterministically exercises the System Under Test (SUT). It's about how to develop and articulate a testing plan to clients in a use case language that all stakeholders understand. I propose to walk through our processes and methodologies to illustrate how we manage risk in an imperfect world where everyone has an opinion, wants to review the results and demands that the schedule be shortened!

A complete test plan for a software release can have multiple phases and numerous components. For example: Regression testing, load and performance testing, automated testing, new feature/functionality testing, API certification, etc. This paper is simply about Regression testing and describes a minimalistic approach that seeks to find balance between software quality and the economic reality of testing where every test case run costs time and money. We seek to define what is "Just enough". In this paper, I will explore how components of our software development life cycle help cope with QA in a "Software as a Service" (SaaS) environment. I will specify how we use software tools and procedures to measure, define and comprehend risk as it applies to test case execution in our imperfect world.

2 Background

In our SaaS environment, we sell services that rely on our software. We are not directly compensated for lines of code we deliver but for speed to market of new services and stability of our current production features. Our clients derive a significant portion of their revenue from these services so stability and high quality software are expected. As our revenue model is tied to production services many of the traditional software development models make no sense.

In this environment, one needs to manage risk and deliver quickly to drive revenue. Caution simply doesn't pay. As you cut corners and get lucky time and time again, cultural tolerance for risk grows. One needs a balanced process to keep all stakeholders and key business owners apprised of the decisions being made and the inherent risks being taken. It is easy to become blasé when negative consequences are avoided time and time again with fire drills and luck. However, Mr. Murphy can't be bribed forever. A testing methodology based on MRS quantifies the risks taken for each release and is easy for non-technical decision makers to understand.

Our Production environment is a mix of isolated stacks of dedicated customer software based on a common framework that rely on a Service Oriented Architecture (SOA) to deliver services. This hybrid model was built for speed to market, innovation and meant to handle lots of changes. Unfortunately, complexity increases with every release and the testing time to execute new feature & regression testing continually grows. Testing time is constantly under pressure to reduce and demonstrate maximum test execution in the shortest possible time frame. Since our test case repository was growing exponentially with rapid feature addition, we had a problem that had to be solved.

3 The Problem

Over the past couple of years, we have engaged customers and stakeholders in a formalized test case review process. While client engagement at this level is an overall positive experience, the side effect is continuous test case growth in an unstructured quasi-subjective manner. Additionally, the problem is compounded with additional growth in test cases by ongoing product evolution, new clients, solving for production test escapes and dealing with customer feedback. With each new release, the regression

cycle needs additional time and/or additional resources to complete. Project managers, business executives, marketing and customers never like this answer. The problem continually grows harder as we add new feature functionality across our customer base. Constant innovation creates new test cases that add to the regression base and that demand longer test cycles to release.

Like many software companies, we incur technical debt, make mistakes and consequently have test escapes (aka defects) that require immediate production fixes. Analysis identifies where the test escape occurred and best practice test refactoring results in new test cases to avoid repetition of the problem.

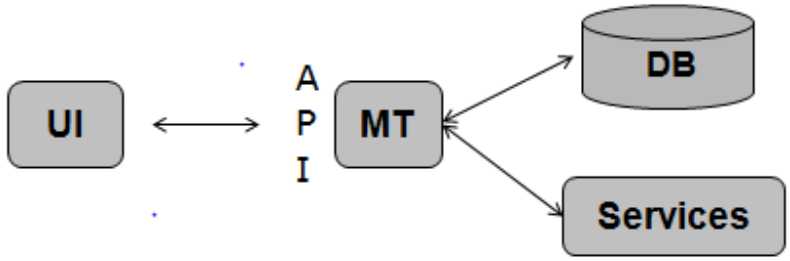
In short, our regression test set grew substantially over time and we found ourselves in a situation where our Quality Assurance Test Management process was neither sustainable nor scalable. Our QA department simply did not have the resources to keep up with the current pace of innovation and resulting test case growth. The increasing maintenance burden represented an ever growing set of test cases requiring ever more time and resources to complete; our problem was becoming unmanageable. After much deliberation, we decided to attack the regression testing problem and instrument our test cases using code coverage techniques. We define the resulting test case set derived from this analysis as the "Minimum Regression Set" (MRS) which is a work product that easily maps to use cases and requirements in a way that all stakeholders can understand.

The MRS philosophy is minimalistic in scope and is very much in sync with the current "Lean Startup" thinking. Reis argues in his book "*The Lean Startup*" (Reis, 2011) that all startups need to get a Minimum Viable Product (MVP) to market as fast as possible to iterate and learn from the market place rather than strive for perfection. The MRS is our QA product equivalent and, like Reis's philosophy, we want to get our MVP into use as fast as possible since time is money.

4 The Environment

Our client base primarily exists of large Telco carriers who, for the most part, choose to operate a fairly normal waterfall methodology. We operate internally in a very iterative mode with a core focus on automated unit tests that drive API & code coverage in order to meet our clients' needs and aggressive timelines. Our development team embraced the Test Driven Development (TDD) methodology for Java (Koskela, 2008) and has an extensive set of Unit tests that drive our Middle Tier (MT) API's:

Figure 1: Three Tiered Architecture



UI = User Interface Layer [Many different applications; mainly Java based]

MT = Middle Tier Layer [all Java]

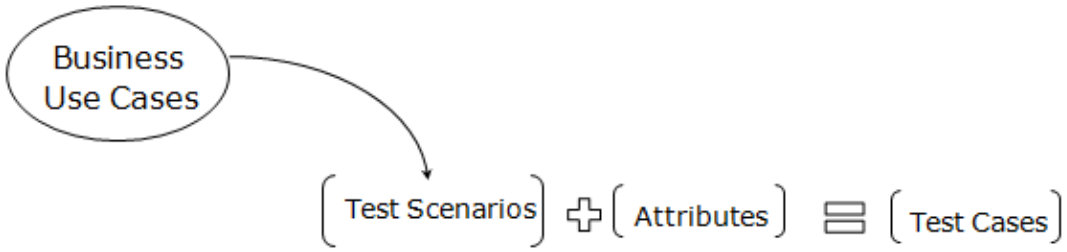
DB = DataBase

As a direct result of this methodology and running unit tests, an instrumented Middle Tier is available for all feature complete QA Releases. The code coverage tool chosen was Cobertura which provides a rich data set. Cobertura is an open source application that instruments the Java bytecode after compilation and works well with our standard eclipse IDE.

5 Test Scenarios

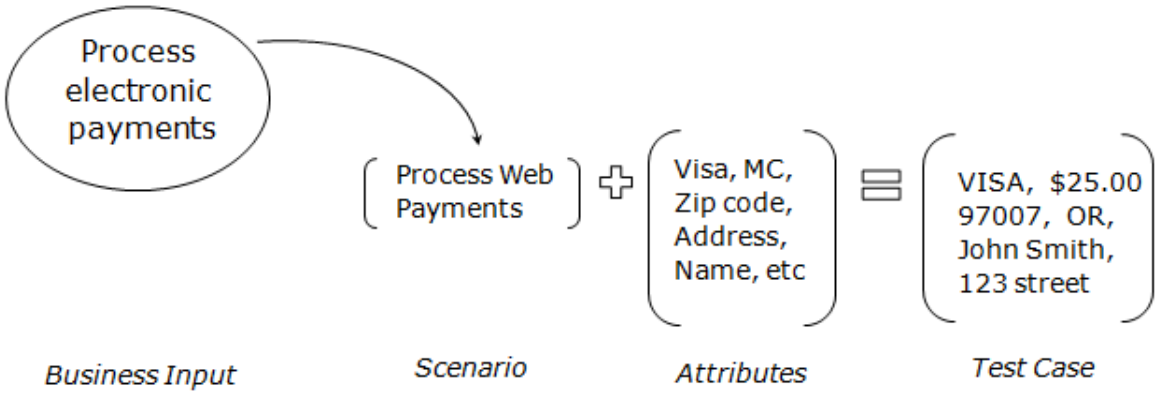
Given our clients have a tendency to describe needed changes more in terms of business use cases, marketing ideas or product delivery strategies rather than traditional software requirements, we need to use these inputs as our starting point. We take Client definition, in whatever form it arrives, and use this to describe "Test Scenarios" that are focused and mapped to client inputs. We purposely segregate out the test case data and refer to these elements as "Attributes". This methodology allows us to iterate on the Test Scenarios at a high level while maintaining their causal relationship to the client input and keep the number of test artifacts we need to manage and review to a reasonable set. Our process works like this:

Figure 2: Scenario Definition Model



For example, as an electronic payments company we process credit card transactions on many different UI channels from every state for multiple different payment devices and amounts. Using our analysis methodology above an example would be:

Figure 3: Example



Once we have a proposed set of Test Scenarios and Attributes, we can review these with all interested parties and stakeholders (both internal & external to our company). Reviewing and generating Scenarios & Attributes in this subjective manner with clients and key stakeholders is somewhat inefficient and certainly creates lots of redundancy due the non-technical nature of the analysis. However, collaborative review up front when the project request is still fresh in everyone's mind creates joint ownership of all the needed Scenarios from a subjective point of view.

A typical review process for one of our web products will create 700-900 unique Scenarios. Now the question remains are all defined Scenarios truly needed?

6 Test Calibration

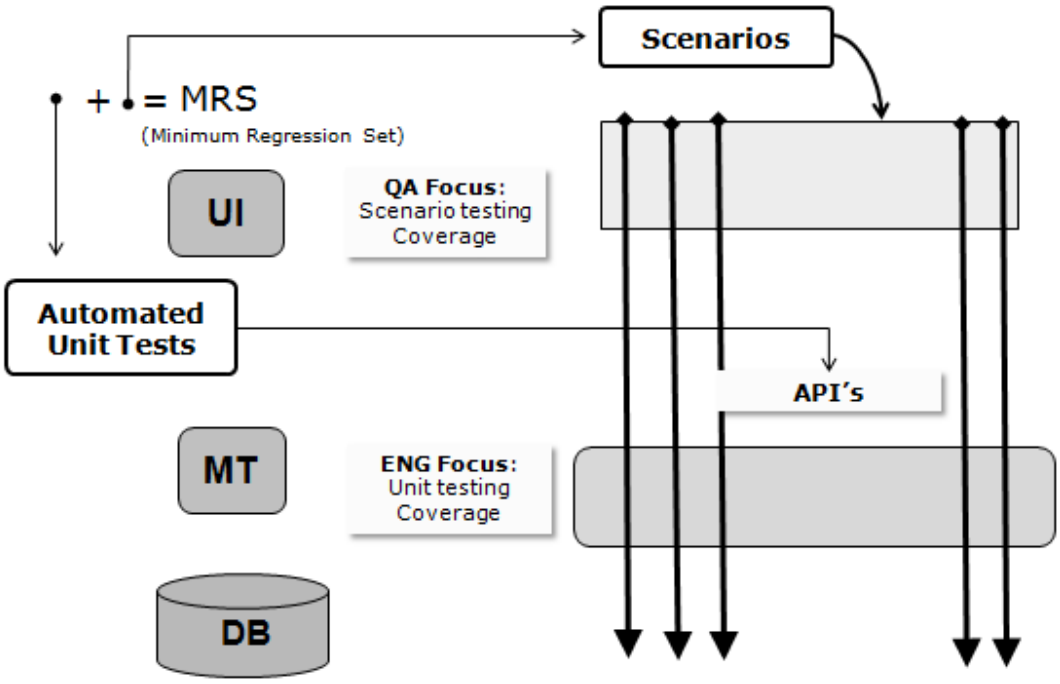
Now that we have collected a large set of Scenarios that we intellectually believe represent the needed set of regression test cases to cover the feature set we need to calibrate the list. "Test Calibration" is simply the term we use to create the MRS using code coverage analysis. Our goal in this effort is to classify all the defined Scenarios into three categories:

- Category 1: The MRS. A single Scenario that exercises a unique path of code though the system, is repeatable and can be measured as unique with Cobertura.
- Category 2: A Scenario that does not add code path uniqueness, therefore is not part of the MRS, but adds a significant & unique data set based on one of the defined attributes.

- Category 3: A Scenario that has neither code path uniqueness nor adds a unique Attribute to the data set. For most part this category represents the emotional category of Scenarios and can usually be traced to previous production test escapes, emotional feature sets and the illusion of importance.

Defining the MRS is actually a simple process where the QA Engineer takes the instrumented Cobertura build which the engineering team built to run their unit tests and replaces the deployed MT-JAR file in the system under test. Then it is a simple manner of individually running each Scenario and validating that code coverage increases with the execution of the scenario. If code coverage numbers do not change, then the Scenario is a candidate for Category 2 or 3. The methodology we use to determine the difference between these two categories is outside the scope of this paper. However, it involves a very similar process where we graph unique attributes and their data sets by Scenario to illustrate uniqueness and thusly define Categories 2 and 3. Extending the 3-Tiered Architecture diagram from figure 1 to illustrate this process looks like this:

Figure 4: Test Calibration



Cobertura provides detailed and simple to understand code coverage data. As you can see from this example (report example from the cobertura home page on sourceforge) the tool dynamically tracks coverage and it is a simple matter to see if your coverage numbers increase as you complete a Scenario.

Figure 5: Cobertura Code Coverage Example

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	55	75%	1625/2179	64%	472/738	2.319
net.sourceforge.cobertura.ant	11	52%	170/330	43%	40/94	1.848
net.sourceforge.cobertura.check	3	0%	0/150	0%	0/76	2.429
net.sourceforge.cobertura.coveragedata	13	N/A	N/A	N/A	N/A	2.277
net.sourceforge.cobertura.instrument	10	90%	460/510	75%	123/164	1.854
net.sourceforge.cobertura.merge	1	86%	30/35	88%	14/16	5.5
net.sourceforge.cobertura.reporting	3	87%	116/134	80%	43/54	2.882
net.sourceforge.cobertura.reporting.html	4	91%	475/523	77%	156/202	4.444
net.sourceforge.cobertura.reporting.html.files	1	87%	39/45	62%	5/8	4.5
net.sourceforge.cobertura.reporting.xml	1	100%	155/155	95%	21/22	1.524
net.sourceforge.cobertura.util	9	60%	175/291	69%	70/102	2.892
someotherpackage	1	83%	5/6	N/A	N/A	1.2

Report generated by Cobertura 1.9 on 6/9/07 12:37 AM.

While our stated goals for the MRS are 100% API and 100% code coverage, the reality is that 100% code coverage is not realistic. There are numerous reasons why a particular Class / Line or Branch can be

ignored safely and as long as the Dev Lead and QA Lead agree we mark these as exceptions on our internal Wiki where coverage and MRS data are kept.

Cobertura provides lots of coverage detail. The ability to drive down from a package level summary to the Java Class and then finally the actual source allows for complete inspection of behavior of a Scenario in the MT layer. Furthermore, the Cyclomatic complexity gives an easily understood scale (bigger the number – the more complex the code) and identifies areas where code reviews should start.

The MRS creation activity yields several interesting results. Generally after execution of approximately a third of the defined Scenarios, the code coverage needle will stop incrementing but we will have nowhere near 100% coverage. This is the moment where we realize that the Scenarios analysis done as an intellectual exercise has missed a number of valid cases. Typically what is missed and overlooked are the error handling routines, obscure use cases and available functionality that was not obvious at review. When running with code coverage enabled, these potential test escapes are very obvious.

7 User Interface White Space

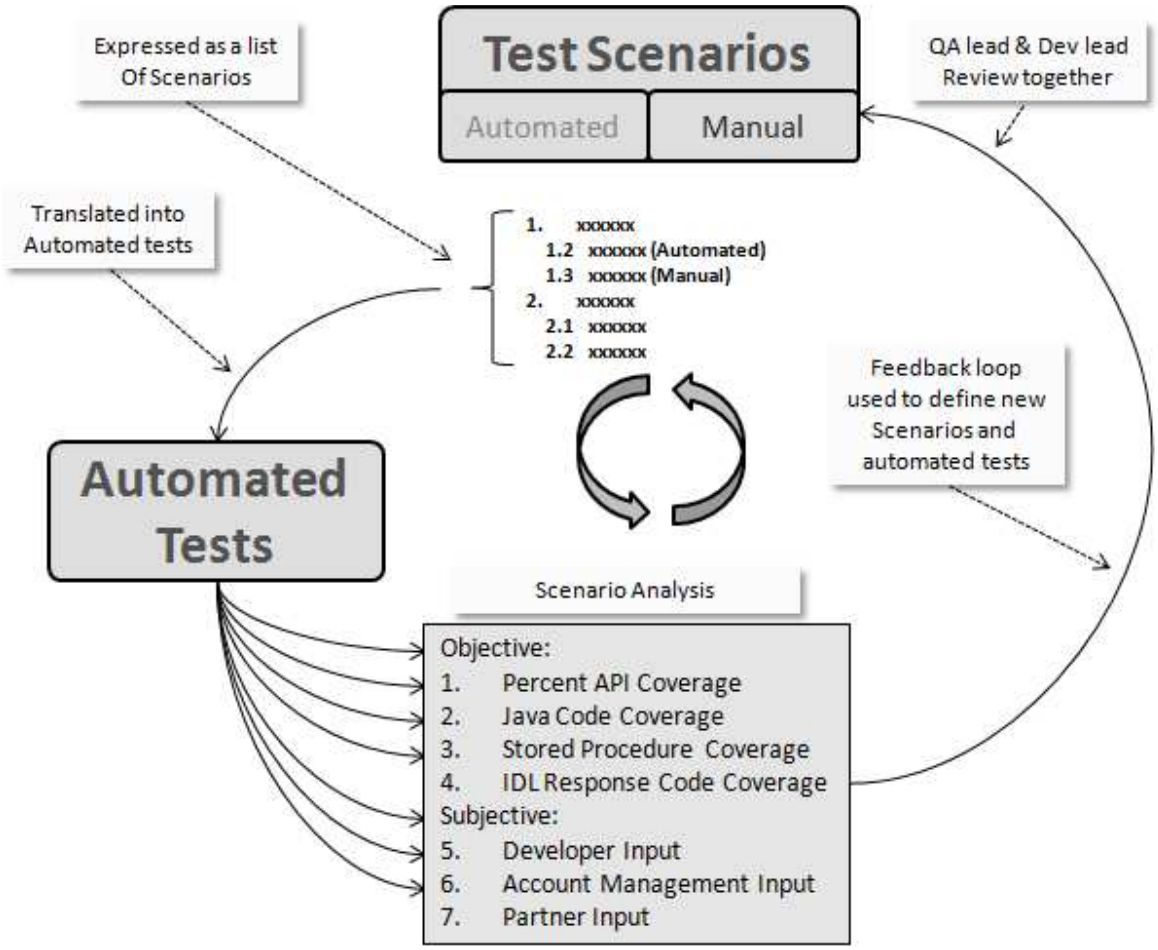
The methodology outlined so far where we define Scenarios and measure code coverage in the middle tier simply implies code coverage in the UI layer. We have made a conscious decision not to directly measure and instrument the UI layers in our system because of the rapid rate of change these interfaces experience. The economic reality is the return on investment of this activity is not warranted due to the time and effort required to create an MRS and frequency of changes.

Once the MRS is defined for an application, one final code review is needed of the UI layer. It is the intention of this review exercise to define what we refer to as the “White Space”. The White Space is the UI code structures not exercised by any of the Scenarios since their scope is entirely contained inside the UI framework. While this activity is code review based, we add these items to the MRS anyway. We label these Scenarios this way since they live in the White Space of Figure 4. Some examples of these items that generate additional White Space Scenarios: JQuery elements, Analytic web tags and form based validation logic.

8 Scenario Lifecycle

Now that we have defined the MRS, let’s examine the overall lifecycle. The diagram below shows the highly iterative nature of our definition process. For each major release where we add new feature / functionality or where we undertake a significant re-factor to our code base, we have a pragmatic process to define or update the MRS for any particular application in our system.

Figure 6: Scenario Life Cycle



A feedback loop ensures we capture needed automated tests and that new Scenarios are captured in the "Test Scenarios".

9 Test Escapes

They happen. Despite best efforts and rigor in the analysis phase to determine what is needed for regression as the MRS, mistakes happen and defects make their way to production. When these events occur, the root cause analysis of the test escape can now be expressed as a part of our methodology and in terms of MRS. In our system, test escapes tend to occur from one of the following cases:

- Automated test failure where we have misinterpreted an expected exit condition; a coding failure in the test case that gives a false positive or fails to run properly.
- MRS definition inaccuracy where an important Scenario is missed in the "White Space".
- The review process overlooked the importance of a particular set of code and a mistake was made identifying it as an acceptable non executed code path.
- Scenario was not executed because of time & resource constraints.

In the first 3 cases, test escapes are easy to resolve by defining new Scenarios & automated tests to ensure the issue doesn't happen again. In the fourth case, this is simply the price we pay for a too aggressive schedule.

10 The Deliverable

At the end of the day, Test Calibration and the resulting definition of the MRS is about risk management, speed to market and delivering high quality software in the context of an economic framework. We want to define the smallest set of regression tests that need to be run in the shortest amount of time because running test cases costs money and delays service delivery.

This structured approach works and allows for precise discussion around the fit for release of a particular Release Candidate build. This is invaluable data for the Project Manager and the "Go/No-Go" meeting to describe precisely in business terms the readiness of a release. Figure 3 is an illustration of how each Scenario can be described in a simple line format, Cobertura code coverage details easily fit on a slide deck and the net result is a simple presentation that clearly outlines the release status. Even more powerful to this discussion is the fact key stakeholders were engaged in creating the initial draft of the Scenario list. The systematic reduction to the MRS is a powerful definition of "Just Enough" testing and facilitates a conversation about test completeness. Clearly, the regression risk of a production release at any point in the project plan is simply the percent complete of testing the MRS; a simple one line sentence that conveys product readiness.

We live in an imperfect world. Should a date occur and business drivers dictate that a production release needs to go live when QA has not finished testing then QA has a simple message for the team → MRS = 45%.

References

Koskela, L 2008 *Test Driven, Practical TDD and Acceptance TDD for Java Developers.* Manning

Reis, E 2011 *The Lean Startup.* Crown Business

Thomas, Young, Brown, Glover 2004 *Java Test Patterns*. Willey

http://cobertura.sourceforge.net/index.html (last accessed June 2012)

www.eclipse.org/ (last accessed June 2012)

http://en.wikipedia.org/wiki/Cyclomatic_complexity (last accessed June 2012)