Assignment 2

1.Test Planning, Execution and Management

interpretations of the reports from 1.7 and 1.8

1. Test cases for eCards applications added into test cases section along with test tags(Design-> test cases) in Tarantula. Test case usually cosists Title,objective, priority, test data and steps for dummy actions and results.
2. Added test cases for Unit testing and System testing. Later, These test cases are categorized into Unit test cases and System test cases in Sets (Design->Sets) in Tarantula.
3. In Requirement section, edited each requirement and linked each test case to the corrosponding requirement by dragging test cases from Cases section to requirement.
4. In Test Objects section, Creted all test objects for the requirements mentioned Requirements section.
5. In Executions section, Creted execution of test cases by linking Test object and Test Cases. Intially it shows 0% when linking test object to the execution.
6. In Test-> test section, from the left pane, Selected each execution and performed testing. The observations what I have seen while testing the application are updated with pass and fail options.
7. Now the executions are showing 100% status.
8. In Report section, generated requirement coverage for testing and exported into PDF. You can see the requirement coverage below.

Requirement Coverage

Current Requirements

	Id	Priority	Cases	Steps	Executions	Last Passed	Last Tested	Raw Passed Coverage
TOTAL			11	18				100%
Displaying all the cards pertaining to the customer as a list	MobApp_2	Normal	1	1				100%
Displaying the multiple cards of the same category,in which each shows the details of the card	MobApp_4	Normal	3	5				100%
Generating QR Code for making transaction	MobApp_5	Normal	2	4				100%
Grouping Debit,Credit and Forex cards and other cards.	MobApp_3	Normal	2	2				100%
log out from the application	MobApp_6	Normal	1	1				100%
User able to login to the system using customer id and password	MobApp_1	Normal	2	5				100%

9. Under Test Result status, selected each test object and generated Test Result status report and exported into PDF.
10. Test-> Case execution list, selected all Test objects and generated Case Execution List

Case Execution List

Project: mummidra-assignment1;Test Objects: Login page Mock object, customer details mock object, Card details mock object, QR code mock object, logout mock object

Test Object	Execution Name	Test Case Name	Execution Date	Duration (min)	Executed by	Result [Priority]	Comments	Defects	Tags
Login page Mock object	Application login	validating login functionality	2015-09-15 10:47	1.07	mummidra	PASSED [H]			MobApp_1
Login page Mock object	Application login	Check customer id and password lengths	2015-09-15 10:42	0.28	mummidra	PASSED [N]			MobApp_1
Login page Mock object	Application login	Valid Customer id	2015-09-15 10:41	0.32	mummidra	PASSED [N]			MobApp_1
Login page Mock object	Application login	Valid Password	2015-09-15 10:47	0.28	mummidra	PASSED [N]			MobApp_1
customer details mock object	Customer details validation	Check the grouping of cards	2015-09-15 10:42	0.78	mummidra	PASSED [N]			MobApp_3
customer details mock object	Customer details validation	Check whether the user details displayed	2015-09-15 10:40	0.40	mummidra	PASSED [N]			MobApp_3
customer details	Customer details	Check whether tap on grouping	2015-09-	0.40	mummidra	PASSED			MobApp_3

Test cases needs to have tags to identify and link them with the requirements. Set of test cases can be grouped using Sets. Test Object is an object to test a unit or system of the application. Test execution takes each Test object to run the test cases. Test cases are linked to test object and tested with execution.
Based on the observations we found in testing real application, The results are updated in the test execution by mentioning pass or fail. Tester can add comment on it.

After Performing test execution, the reports from the requirement shows the coverage of all requirements. Case execution list is the one that consolidates the test objects and report status in one report.

Structural code coverage tool;

Code Coverage allows you to identify what parts of your source code were executed during a test. Xcode supports code coverage testing with GCOV for iOS applications. When set up for GCOV use, Xcode generates .gcno files when compiling your app and .gcda data files when your app exits. The following steps show you how to set up your project in Xcode to generate these GCOV files.

After compiling the files with configuration of GCOV buildm it generates both gcno and gcda file like this

```
AppDelegate.d                    MyApplication_dependency_info.dat main.d
AppDelegate.dia                  ViewController.d                  main.dia
AppDelegate.gcda                 ViewController.dia                main.gcda
AppDelegate.gcno                 ViewController.gcda               main.gcno
AppDelegate.o                    ViewController.gcno               main.o
```

```
$ cd <path/to/directory with .gcda and .gcno files>
$ xcrun gcov </path/to>/MyApplication/MyApplication/ViewController.m
Lines executed:69.23% of 13
```

The result shows the no. of lines executed from the file.

The set up allows the system to execute each line and shows the coverage of the file.

2. **Read and report on two articles**

Summary

Evaluating Static Analysis Defect Warnings On Production Software

The paper discuss the experimental evaluation of the accuracy and value of warnings reported by static analysis tools(FindBugs in java). Provided the environment for running the static analysis tool(Sun;s java 6 JRE,Sun's Glassfish JEE server).

Many tools reflecting true defects in the code but those are trivial. Only smart tools might detect infeasible warnings. If the developer believes the defect could result in significant misbehavior The defects warnings are regarded as false positive mentioned in the paper by Wagner et al.

when discussing false positives from static analysis, the clarification is needed on the significance of the defects generated from the static analysis tools. The automatic tools and defect detection tools don't try to identify the parts uncovered by specification. Though inconsistent and nonsensical may not be defects, these are detected by many defect detection tools. These kind of defects are exacerbated in memory safe languages like java.

Static analysis for defect detection are used in performing code review for a newly written module and looking for defects in a large existing code base. The paper mainly concerns on the context of detecting defects in a large existing code base. The result of running FindBugs against Sun's JDK, Sun's Glassfish J2EE Server and portion of Google java code base is reported.

FindBug is written in Java and an open source static analysis tool. It can report nearly 300 different bug patterns. It does not perform inter procedural context sensitive analysis. Each bug pattern is grouped into Correctness or bad practice or performance or internationalization categories. And pattern is assigned a priority(High, Medium and Low). Heuristic unique

detection patterns determine the priorities. A low false positive rate for issues with high and medium priority correctness.

FindBug can be run from command line and results can be saved to XML in order to store them in database. Instance hash mechanism is used to allow a detect to be matched with previous reporting of that warnings. Findbugs can correctly diagnoses deliberate errors and masked errors.

Often users check objects null like .equals(null). These kind of nonsensical code can't be ignored. Fixing the bugs that does not significantly impact the behavior of the application should not introduce other. Fixing such bugs should consider easier to understand and maintain and less likely to break in the face of future modifications or uses. The most warnings from FindBugs are obvious once we look at a few line of code and fix is straight forward and obvious.

The issues are marked as impossible are node that we believe could not be exhibited by the code, trivial issues that happened but would have minimal adverse impact. Trying to classify the defect warnings into false positives and true positives over simplifies the issue. Inconsistent and bogus mistakes may have no adverse impact of software functionality. Removing such bugs make software easier to maintain. Static analysis tools reports true but trivial issues because in general they don't know what the code is supposed to do. Substantial portion of the issues found do have functional impact and reviewing warning to discern which have apparent functional impact is not a problem. The existence of low impact defects is not a barrier to adoption of static analysis technologies.

Report shown that FindBugs reports a substantial number of defects on production software. FindBugs reports more than 250 different bug patterns and some of them classified into correctness bugs. There are defects reported, developers are willing to address those bugs. FindBug is being improved over years. It shows that using static analysis tools are good for improving the accuracy of the software.

Concepts Learned

Learned about FindBug static analysis tool which is open source and written in Java.
Many Static analysis tools identifies defects which are trivial but some static analysis tool like FindBug produces the defects which needs the immediate attention of the developer. Classifying the bugs into false positives and true positives over simplifies the issues in order to report by the static analysis tools.

Experimental Designs
FindBug efficiency in reporting bugs through the practical execution of tool on Sun's Java6 JRE ,Sun's Glassfish JEE server explained how the analyzers correctly diagnoses deliberate errors and masked errors and reporting more than 250 different bug patterns invalidates claim that Static analyzers produce more trivial bugs than serious defects.

Experience

XCode,a tool for developing IOS and OSX applications, also has in build static analyzer tool. We practiced the static analyzer in unit testing phase to identify the obvious and critical issues. The usage tool helped us to reduce the bug leaks to system testing phase.

Summary

Coverage is not strongly correlated with test suite effectiveness

The discussion about the test-suit size, coverage, effectiveness and how these are correlated to one another are covered in this paper. The effectiveness of the test suits even though it has many test cases, whether ignoring some of the test cases or take them into consideration are mentioned. the automatically generated test cases and manually generated test case do not generalize because the manually written test cases have high correlation and the automatically written test cases have low correlation. And there was a lot of studies saying that the "effectiveness is related to coverage". To measure the effectiveness, there are some methodologies

To selecting subject Programs, the programs from the various domains are drawn by following some criteria.
Apache POI, Closure Compiler, HSQLDB, JFreeChart, Joda Time. Based on size of the program (minimum of 10000 SLOC),program having large number of test cases and able to use Ant and JUnit. Generating faulty scenarios, generating test Suits, measuring coverage,and measuring effectiveness has been covered. Creation of number of mutants and injected them into the original source code and applied a large number of test suits on these faulty codes. If these test suits were able to kill the mutants, then those are called the non-equivalent mutants and if they were not able to kill them, then those were called equivalent mutants. The calculation of the percentage of equivalent and non-equivalent mutants of all selected programs.
For each subject program, generated a fixed number of test suits basing upon the master suit of that particular program. the number of statements covered, number of decision statements covered and number of modified statements covered by the test suits generated in the previous step there are two types of measurements.

Raw Effective Measurement: The number of mutants the test suit detected by the number of non-equivalent mutants that were generated by that program.

Normalized Effective Measurement: The number of mutants the test suit detected by the number of non-equivalent mutants it covered.

The methodologies stated implied that the high levels of coverage by a test suit need not make it effective.

Experimental Designs

Taking five types subject programs from various domains which are based on java language and each of the program has over 10000 SLOC. But I think we can get much more broad results if we use different type of languages. Just like in the previous experiments conducted there were different languages involved. Because some programs might not have some inbuilt functions and some programs might not have the variables declarations and there possibility of error occurrence might be different for them. Therefore the number of mutations need to be included and the number of test cases generated might different for them.

Critiques:

Correlating my experience based on the paper:
Testing team used to perform automation on application by code from the development team and used to write automation scripts to test the application. Generating automation script using java script in xcode and giving dummy values as inputs seems like testing followed mutants to test the code.