CHAPTER 2

Test Case Prioritization Based on Risk Exposure

2.1 Introduction

Regression testing is an expensive testing process used to validate modified software and detect whether new faults have been introduced into previously tested code. A typical approach to regression testing is to save the test suites that are developed for a product, and reuse them to revalidate the product after it is modified. Regression testing is an essential part of an effective testing process for ensuring software quality. It is the process of validating modified software to provide confidence that the changed parts of the software behave as intended and that the un-changed parts of the software have not been adversely affected by the modification [1,123]. Without running a modified program on all test cases of test suite T, regression test selection (RTS) attempts to select a cost-minimized subset of test cases to determine if the modified program has the same behavior as a previous, acceptable version of the program running on T.

In all software projects, risks are taken. In one way testing in high integrity environment is easy. Every textbook, process, method and technique must be used to achieve an explicit aim: to minimize risk. In fast moving E-business projects, risk taking is inevitable. Balancing testing against risk is essential because we never have the time to test everything. It is tough to get it right. If we do not talk to the risk takers in their language we will never get the testing budget approved. Software testers must become expert in risk and they must identify
failure modes and translate these into consequences to the sponsors of the project. In this way, testers, managers and sponsors can reconcile the risks being taken to the testing time and effort. Using risks to prioritize tests means that testers can concentrate on designing effective tests to find faults and not worry about doing too little testing.

The true art of testing is to select a meaningful subset of test cases that are most likely to uncover problems, thereby reducing the total number of tests while maintaining confidence in the product’s operation.

The chapter comes up with the strategy of prioritization of the regression test suite based on the cost of risk associated with each test case. The technique strives to lower the costs without overly reducing the effectiveness by carefully prioritizing the test suite. The test cases are executed based on the priority derived from the cost of risk in the given duration of time.

2.1.1 Regression Testing

One major difference between regression testing and development testing is that during regression testing an established suite of tests will typically be available for reuse. The established suite of tests is defined as full test suite. With selective re-test (regression test) techniques, only those test cases are rerun that test the affected entities of the modified program. These test cases make up regression test suite. If the cost of selecting a reduced subset of tests to run is less than the cost of running the tests that we omit, the selective retest technique is more economical than the re-test-all technique [6].

In [7], Rothermel and Harrold described regression testing techniques as follows: “Given a program P, a modified version P’, and a set T of test cases used previously to test P,
Regression analysis and testing techniques attempt to make use of a subset of $T$ to gain sufficient confidence in the correctness of $P'$ with respect to behaviors from $P$ retained in $P'$.

Let $P$ be a procedure or program, let $P'$ be a modified version of $P$ and let $T$ be a test suite for $P$. A typical regression test proceeds as follows:

- Select $T' \subseteq T$, a set of test cases to execute on $P'$.
- Test $P'$ with $T'$. Establish $P''$'s correctness with respect to $T'$.
- If necessary, create $T'''$, a set of new functional or structural test cases for $P'$.
- Test $P'$ with $T'''$, establishing $P''$'s correctness with respect to $T'''$.
- Create $T''''$, a new test suite and test history for $P'$, from $T$, $T'$, and $T'''$.

2.1.2 Prioritization Of Test Cases

A typical regression test cycle would involve testing the entire test suite. The limited available calendar time does not allow thorough testing of all functions. Focus has to be put on those areas representing the largest risk if a fault occurred and also the resources available for testing are limited. Therefore the testing of entire test suite, which might run into thousands of test cases is practically infeasible. To overcome this challenge a risk based approach to prioritize the test cases may provide the solution. Test case prioritization is a concept to reduce the number of test cases or to order their execution, in order to increase the effectiveness of testing, based on certain rational, non-arbitrary criteria. This ordering is must by seeing the numerous possibilities of test cases and by the strict schedules and budgets. But again while dealing with the prioritization, the most important question is that how it should be done and what are the different factors, which can affect the selection of test cases. Prior to going for prioritization, first of all the aim should be decided. Based on this aim the criteria can be devised. Prioritization scheme basically addresses the two key concepts:
• What features must be tested?
• What are the consequences if some feature is not tested?

**Given:** A test suite T, a set of permutation of T, PT, a function f from PT to the real number.

**Problem:** Find $pt \in PT$ such that $(\forall pt')(pt' \in PT) \ [f(pt) \geq f(pt')]$  

(2.1)

PT represents the set of possible orderings (prioritization) of T and f is a function that when applied to an ordering, yields an evaluation of that ordering. Test suite prioritization algorithm input a test suite T and output a sequence of test cases T'.

There are many goals of prioritization. They are:

• Testers may wish to increase the rate of fault detection of a test suite - that is, the likelihood of revealing faults earlier in a run of regression tests using that test suite.
• Testers may wish to increase the coverage of coverable code in the system under test at a faster rate, allowing a code coverage criterion to be met earlier in the test process.
• Testers may wish to increase their confidence in the reliability of the system under test at a faster rate.
• Testers may wish to increase the rate at which high-risk faults are detected by a test suite, thus locating such faults earlier in the testing process.
• Testers may wish to increase the likelihood of revealing faults related to specific code changes earlier in the regression testing process.
Test case prioritization can be used either in the initial testing of software or in the regression testing of software. One difference between these two applications is that, in the case of regression testing, prioritization techniques can use information gathered in previous runs of existing test cases to help prioritize the test cases for subsequent runs.

It is useful to distinguish two varieties of test case prioritization: general test case prioritization and version-specific test case prioritization. In general test case prioritization, given program P and test suite T, we prioritize the test cases in T with the intent of finding an ordering of test cases that will be useful over a succession of subsequent modified versions of P. Thus, general test case prioritization can be performed following the release of some version of the program during off-peak hours, and the cost of performing the prioritization is amortized over the subsequent releases. It is hoped that the resulting prioritized suite will be more successful than the original suite at meeting the goal of the prioritization, on average over those subsequent releases.

In contrast, in version-specific test case prioritization, given program P and test suite T, we prioritize the test cases in T with the intent of finding an ordering that will be useful on a specific version P' of P. Version-specific prioritization is performed after a set of changes have been made to P and prior to regression testing P'. Because this prioritization is accomplished after P' is available, care must be taken to keep the cost of performing the prioritization from excessively delaying the very regression testing activities it is intended to facilitate. The prioritized test suite may be more effective at meeting the goal of the prioritization for P' in particular than would a test suite resulting from general test case prioritization, but may be less effective on average over a succession of subsequent releases.
Typically -- though not necessarily -- general test case prioritization does not use information about specific modified versions of P, whereas version-specific prioritization does use such information. Of course, it is possible for general test case prioritization techniques to incorporate information about expected modifications to improve the average performance of prioritized test suites over a succession of program versions, and it is possible to use prioritization techniques that ignore the modified program as version-specific techniques.

It is also possible to integrate test case prioritization with regression test selection or test suite minimization techniques - for example, by prioritizing a test suite selected by a regression test selection algorithm, or by prioritizing the minimal test suite returned by a test suite minimization algorithm.

2.1.3 Prioritization Techniques

Given any prioritization goal, various prioritization techniques may be applied to a test suite with the aim of meeting that goal. For example, in an attempt to increase the rate of fault detection of test suites, we might prioritize test cases in terms of the extent to which they execute modules that, measured historically, have tended to fail. Alternatively, we might prioritize test cases in terms of their increasing cost-per-coverage of code components, or in terms of their increasing cost-per-coverage of features listed in a requirements specification.

Researchers have worked in arriving at various such techniques. There are 14 such techniques, which are classified into three groups [20]: Control Group, Statement Level Group and Function Level Group. Table 2.1 details all these 14 techniques.

<table>
<thead>
<tr>
<th>#</th>
<th>Group</th>
<th>Name of the Technique</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Control Technique</td>
<td>Random Ordering</td>
<td>Randomized ordering</td>
</tr>
<tr>
<td>Statement Level Technique</td>
<td>Optimal Ordering</td>
<td>Ordered to optimize rate of fault detection</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------</td>
<td>-------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Total Statement Coverage Prioritization</td>
<td>Prioritize on coverage of statements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Additional Statement Coverage Prioritization</td>
<td>Prioritize on coverage of statements not yet covered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total fault-exposing potential prioritization</td>
<td>Prioritize on probability of exposing faults</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Additional Fault exposing potential prioritization</td>
<td>Prioritize on probability of faults adjusted to consider previous test cases.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total function coverage prioritization</td>
<td>Prioritize on coverage of functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Additional Function coverage prioritization</td>
<td>Prioritize on coverage of functions not yet covered.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total fault exposing potential (function level) prioritization</td>
<td>Prioritize on probability of exposing faults</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Additional fault exposing potential (function level) prioritization</td>
<td>Prioritize on probability of faults adjusted to consider previous test cases.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total fault index prioritization</td>
<td>Prioritize on probability of faults existence.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2.1: List of Test Case Prioritization Techniques

<table>
<thead>
<tr>
<th>12</th>
<th>Additional fault index prioritization</th>
<th>Prioritize on probability of faults existence, adjusted to consider previous test cases.</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Total fault index with fault exposing potential coverage prioritization</td>
<td>Prioritize on combined probability of faults existence and fault exposure.</td>
</tr>
<tr>
<td>14</td>
<td>Additional fault index with fault exposing potential coverage prioritization</td>
<td>Prioritize on combined probability of faults existence and fault exposure, adjusted on previous coverage.</td>
</tr>
</tbody>
</table>

#### 2.1.4 Simple Risk Model

Risk is an Event that has some probability of happening, and that if it occurs will result in some loss. Risk Based Testing is to do heavier testing of those parts of the application that may bring higher risks. The main guide for selecting tests cases is risk i.e. how much risk is associated with each test case? Risk tells the importance attached with a test case but how to measure the risk? To provide this solution we must perform risk analysis to identify the most critical areas both to the vendor and to the customer. Risk analysis is started at the start of the project and it is continuously updated throughout the test execution.

A model can be developed to calculate the risk exposure for each function, which can be further mapped to the probability of an error and the cost (consequence or impact) of an error if it occurs in production

$$RE_f = P_f * C_f$$  \hspace{1cm} (2.2)
Where:

- $\text{RE}_f$: Risk Exposure of function $f$
- $\text{P}_f$: Probability of fault occurring in function $f$
- $\text{C}_f$: Cost if fault occurs (in production) in function $f$

2.2 Risk Based Prioritization of Test Cases

Risk based prioritization involves the execution of following four steps sequentially to arrive at prioritized regression test suite:

- Step 1: Assess cost $\text{C}_t$ for each test case
- Step 2: Derive probability $\text{P}_t$ for each test case
- Step 3: Calculate Risk Exposure $\text{RE}_t$ for each test case
- Step 4: Select test cases with top $\text{RE}_t$ as regression test

2.2.1 Assess Cost for each Test Case

There are two kinds of costs associated with each function:

Consequences of faults from customer's perspective ($\text{C}_t$ (Customer)): - It would include costs as seen by the customer i.e.

- Probability of a legal threat
- Loosing market place
- Not fulfilling government regulations etc.

Cost of a fault as seen by the vendor ($\text{C}_t$ (Vendor)) i.e.

- Probability of negative publicity
- High software maintenance cost
- Loosing business

Assuming Weight $C_t$ (Customer) and $C_t$ (Vendor) equally

$$C_t = \frac{(C_t \text{ (Customer)} + C_t \text{ (Vendor)})}{2} \quad (2.3)$$

To calculate cost we need to calculate both $C_t$ (Customer) and $C_t$ (Vendor). Both these costs are calculated as described below:

$C_t$ (Customer)

- Understand the criticality of the function from the business standpoint and it’s impact if an error occurs in production.
- Prepare a questionnaire for coming out with criticality of the function
- Score each test case based on answer for questionnaire as $C_t$ (Customer) on a 1~5 scale (1 - low, 5 - high)

$C_t$ (Vendor)

- Based on the system complexity what is the cost of fixing the bug
- The key factor in coming out with the cost to the vendor for any particular function is the coupling of that function with other modules. Coupling [19] measures the strength of connection between two modules in a program. This measure is mainly used to evaluate the quality of design. Coupling is also a predicator of programming and testing effort. During program development, coupling predicts the difficulty of divide-and-conquer process undertaken by the
design. During program maintenance, coupling predicts the difficulty of changing the program and likelihood of errors introduced by these changes. If the coupling is high the $C_t$ (Vendor) is high as the cost of fixing bugs by the vendor would be high.

- Measure $C_t$ (Vendor) on a 1~5 scale (1 - low, 5 - high)

### 2.2.2 Derive Probability of Failure

The chance of failure can be attributed to two factors: Chance/probability of error and Frequency of use. Identify all the possible reasons, which can lead to a fault in the function. Probability of occurrence of a fault can be because of the any one or combination of the reasons: New Function, Design, Size, Complexity and Interfaces Used By the system.

The indicators used to calculate the probability of fault for a particular function $P (f)$ are weighted i.e. the weight varies from 1 to 5, rating 5 is the most important indicator of a function with poor quality. Say for example weight used for different indicators are: New Function (5), Design (4), Size (3), Complexity (2) and Interfaces Used By the system (1).

Once all the reasons are identified, assign a numerical value to each of the indicators for each of the function under test, based on the likely hood of occurrence of the fault because of that reason. Say if a function XYZ is a new function in the system the likely hood of having a fault in that system can be high. If function XYZ were a small function the likely hood of fault owing to size would be very low and so on. The numerical values are assigned to each identified reason (effecting the functionality): Very High (5), High (4), Medium (3), Low (2), and Very low (1). Once the numerical values are assigned, the probability of fault occurrence is calculated for each function.
\[
WeightedAverage = \frac{W1 \cdot N1 + W2 \cdot N2 + \Lambda WN \cdot NN}{W1 + W2 + \Lambda WN}
\] (2.4)

\[
ProbabilityOfFaultOccurrence = \frac{WeightedAverageOfAllFactors}{HighestWeightedAverage}
\] (2.5)

Table 2.2 depicts the probability of fault occurrence calculation.

<table>
<thead>
<tr>
<th>Function Name To Be Tested</th>
<th>Reasons Effecting The Function</th>
<th>Probability Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>New Function (5)</td>
<td>Design (4)</td>
</tr>
<tr>
<td>F100</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>F200</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>F300</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.2: Calculation of Probability of Fault Occurrence

2.2.3 Calculate Risk Exposure

Risk exposure is calculated by combining \( C_t \) and \( P_t \) as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>( C_t )</th>
<th>( P_t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>F100</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>F200</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>F300</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fn</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>( C_t )</th>
<th>( P_t )</th>
<th>( RE_t = C_t + P_t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>F100</td>
<td>4</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>F200</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>F300</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Fn</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
2.2.4 Algorithm for Prioritization using Risk Exposure

Inputs:
F : Set of functions for P'
E : Set of Probabilities parameters
C(V) : Set of cost elements for vendor
C(C) : Set of cost elements for customers
T(P) : prioritized test suite
P : Original program P
P' : Modified program
T : original Test suite

Prioritization Algorithm

1. Calculate_risk_exposure ( F ) :
2. Translate_Function_to_testcases(F);
3. While T is not empty do
4. T(P) = T(p)||extract-test_case_with_next_highest_re(T);
5. Endwhile

Calculate_risk_exposure(F)

1. For every function f ∈ F
2. p(f) = Calculate_probability_of_risk_occurrence(f);
3. s(f) = Calculate_impact(f);
4. riskexp(f) = p(f) * s(f);
5. endfor

Calculate_probability_of_risk_occurrence(f)

1. for every e ∈ E
2. $p(\text{initial}) = \text{Calculate\_probability}(f, e)$;

3. $p1(f) = \text{weight}(e) \times p(\text{initial})$;

4. 

5. for every $e \in E$

6. $p1(f) += p(f, e)$;

7. $w += w + \text{weight}(e)$;

8. 

9. $p2(f) = p1(f) / w$;

10. $p(f) = p2(f) / \text{Highest}(p(f))$;

**Calculate impact($f$)**

1. For every $c \in C(v)$

2. $s1(f, c) = \text{Calculate\_cost}(f, c)$;

3. 

4. for every $c \in C(c)$

5. $s2(f, c) = \text{calculate\_cost}(f, c)$;

6. 

7. for every $c1 \in C(v)$ and $c2 \in C(c)$

8. $s(f) = s1(f, c1) + s2(f, c2)$;

9. 

**2.2.5 Requirements Traceability Matrix**

Prioritization schemes often provide list of features to be tested and not the test cases. The tester must determine which test cases correspond to the targeted features. Requirements traceability matrix maps requirement to test cases and provides the following information
• Which test cases exercise a specific feature
• Which requirements have no corresponding test case
• Which features still need to be tested
• Which test cases are affected by changes in the requirements

<table>
<thead>
<tr>
<th>Function</th>
<th>TC1</th>
<th>TC2</th>
<th>TC3</th>
<th>TC4</th>
<th>TC5</th>
<th>TC6</th>
<th>TC7</th>
<th>TC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function 1</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function 2</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function 3</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function 4</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Function 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>

Table 2.3: Requirements Traceability Matrix

2.2.6 Regression Test Suite Selection Method

Since the major goal of regression testing is to assure system stability, we have to rerun test cases for requirements attributes that have been modified or may be affected by the modifications. Through regression testing, we want to achieve adequate confidence in software quality. We would like to have our regression test suite achieve some coverage target as well. To get “good enough quality,” our regression test suite should achieve some coverage target. This coverage target has to be set up depending on the available time and budget.

• Identify a coverage target in terms of the percentage of the entire regression test suite based on the time and resources available for regression testing. (For example 20% of the entire test suite)
• Select the test cases with highest $RE_t$ to make 20% of the entire test suite.
<table>
<thead>
<tr>
<th>Test Case</th>
<th>Full Test Suite</th>
<th>Regression Test Suite (20%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T100</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>T200</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>T300</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Tn</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 2.4: Selected Regression Test Suite

Table 2.4 lists the selected regression test suite, where Y means the test cases is selected and N means test case is omitted.

2.3 Summary

Test case prioritization techniques (Control Group, Statement Level Group and Function Level Group) schedule test cases on the basis of code coverage at the fastest rate, or the features to be exercised in order of expected frequency of use, or faults can be detected as early as possible. These techniques are very subjective in nature and don’t actually mitigate all the risks in which can be faced if the software fails. These techniques are very costly in terms of prioritization of test cases and are generally valid for software systems written in low level and third generation languages.

For commercial software by calculating risk exposure associated with each test case, test cases can be prioritized. The proposed technique will help in coming up with objective regression test suite. The identified test suite will help in covering the high-risk exposure functions of the system under stringent time line. The approach is quite systematic and requires straightforward calculation.