STOPPING CRITERIA FOR REGRESSION TESTING IN GUI APPLICATION USING FAILURE INTENSITY AND FAILURE RELIABILITY

Miss Chalita Somsorn

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science Program in Computer Science and Information
Technology
Department of Mathematics and Computer Science
Faculty of Science
Chulalongkorn University
Academic Year 2015

เกณฑ์การหยุดสำหรับการทดสอบแบบถดถอยในโปรแกรมส่วนต่อประสานกราฟฟิกกับผู้ใช้ความเข้มของความขัดข้องและความเชื่อถือได้ของความขัดข้อง

นางสาวชาลิตา สมสร

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2558

Thesis Title	STOPPING CRITERIA FOR REGRESSION TESTING IN GUI APPLICATION USING FAILURE INTENSITY AND FAILURE RELIABILITY
By	Miss Chalita Somsorn
Field of Study	Computer Science and Information Technology
Thesis Advisor	Associate Professor Peraphon Sophatsathit, Ph.D.

Accepted by the Faculty of Science, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

____Dean of the Faculty of Science

(Associate Professor Polkit Sangvanich, Ph.D.)

THESIS COMMITTEE

____Chairman

(Professor Chidchanok Lursinsap, Ph.D.)

____Thesis Advisor

(Associate Professor Peraphon Sophatsathit, Ph.D.)

____External Examiner

(Assistant Professor Kriengkrai Porkaew, Ph.D.)

ชาลิตา สมสร : เกณฑ์การหยุดสำหรับการทดสอบแบบถดถอยในโปรแกรมส่วนต่อประสานกราฟฟิกกับผู้ใช้ความเข้มของความขัดข้องและความเชื่อถือได้ของความขัดข้อง (STOPPING CRITERIA FOR REGRESSION TESTING IN GUI APPLICATION USING FAILURE INTENSITY AND FAILURE RELIABILITY) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: รศ. ดร. พีระพนธ์ โสพัศสถิตย์, 68 หน้า.

งานวิจัยนี้เสนอเกณฑ์การหยุดทดสอบแบบถดถอยในโปรแกรมส่วนต่อประสานกราฟิกกับผู้ใช้เพื่อประเมินระยะเวลาที่เหมาะสมในการหยุดเพื่อไม่ให้สิ้นเปลืองงบประมาณสำหรับการทดสอบมากเกินไป ซึ่งเป็นสิ่งจำเป็นสำหรับการแก้ไขปรับปรุงซอฟต์แวร์ที่สามารถนำไปใช้งานได้ในเวลาอันสั้น แต่ยังคงมีคุณภาพ ปัญหาหนึ่งที่เป็นอุปสรรคต่อการบรรลุตามเป้าหมายขึ้นอยู่กับลำดับของกรณีทดสอบที่ใส่เป็นข้อมูลนำเข้า ลำดับนี้มีผลต่อจำนวนของความขัดข้องที่พบ ดังนั้นงานวิจัยนี้เสนอตัวแบบการหยุดซึ่งพิจารณาปัจจัยที่มีผลกระทบความน่าเชื่อถือของซอฟต์แวร์และค่าใช้จ่ายโดยประมาณในการดำเนินการทดสอบต่อ ขั้นตอนวิธีคือจัดลำดับกรณีทดสอบออกเป็นลำดับที่แตกต่างกันเพื่อใช้เป็นข้อมูลนำเข้าสำหรับการทดสอบแบบถดถอย เมื่อพบความขัดข้อง ความขัดข้องนั้นจะถูกแก้ไขทันทีก่อนที่จะดำเนินการทดสอบต่อ การทดสอบนี้หยุดเมื่อจำนวนข้อขัดข้องและค่าใช้จ่ายไม่เกินงบประมาณที่ตั้งไว้

การวัดผลสมรรถนะของเกณฑ์ที่นำเสนอครอบคลุมตัวชี้วัดสามประเภท ได้แก่ ความเข้มของความขัดข้อง ค่าใช้จ่ายในการทดสอบและแก้ไข และ ความน่าเชื่อถือ ความเข้มของความขัดข้องเป็นฟังก์ชันของข้อผิดพลาดและอัตราที่พบ ค่าใช้จ่ายเป็นฟังก์ชันของเวลาที่ใช้แก้ไขข้อผิดพลาด ฟังก์ชันของความเชื่อถือเป็นการแจกแจงแบบไวบุลล์เข้ามาเพื่อให้สะท้อนข้อมูลทดสอบได้ดีขึ้น ตัวแบบที่นำเสนอได้ถูกทดสอบกับโปรแกรมประยุกต์ที่ใช้งานจริงสำหรับส่วนต่อประสานกราฟิกกับผู้ใช้ ซึ่งให้ผลลัพธ์เกณฑ์การหยุดที่น่าพอใจ

ภาควิชา	คณิตศาสตร์และวิทยาการคอมพิวเตอร์	ลายมือชื่อนิสิต	....
		ลายมือชื่อ อ.ที่ปรึกษาหลัก	....
สาขาวิชา	วิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ		
ปีการศึกษา	2558		

5772608523 : MAJOR COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
KEYWORDS: GUI;REGRESSION TESTING; FAILURE; RELIABILITY; TEST CASES

CHALITA SOMSORN: STOPPING CRITERIA FOR REGRESSION TESTING IN GUI APPLICATION USING FAILURE INTENSITY AND FAILURE RELIABILITY. ADVISOR: ASSOC. PROF. PERAPHON SOPHATSATHIT, Ph.D., 68 pp.

This research proposes some criteria for GUI regression testing to determine the appropriate time to stop without wasting too much testing cost. This is essential for all software upgrades that can be released in a reasonably short time, yet still guarantees the product quality. One difficulty to achieve such a target depends on the sequence of test cases being input. The order of the input test case input sequence affects the number of failures found. As such, a test-stoppage model is proposed by determining factors that affect software reliability and the expected cost of continuing test. The procedure prioritizes the order of test cases into different sequences for the regression test input. When a failure is found, it is immediately edited before the test resumes. The test terminates when the failure intensity is within the predetermined threshold and the expected cost does not exceed the allotted budget limit.

Performance of the proposed criteria encompasses three measures, namely, failure intensity, cost of testing and editing, and reliability. Failure intensity is a function of faults and fault detection rate. The costs are function of time spent on fixing errors. The reliability function incorporates Weibull distribution to better reflect the test data. The proposed model is tested using real GUI applications as test data. Performance shows satisfactory results on stopping criteria.

Department:	Mathematics and Computer Science	Student's Signature	....
		Advisor's Signature	....
Field of Study:	Computer Science and Information Technology		
Academic Year:	2015		

ACKNOWLEDGEMENTS

Though the following dissertation is an individual work but I would never have been able to finish without the help, support, guidance and efforts from my advisor, friends, and my family. Firstly, I would like to thank my advisor, Dr. Peraphon Sophatsathit for his excellent guidance, caring, patience, and providing me with an excellent atmosphere. Without your pearls of wisdom, it would have been nearly impossible to finish this dissertation. It was favored that I had the chance of being one of your students. I would like to thank my committee members Dr. Chidchanok Lursinsap and Dr. Kriengkrai Porkaew for taking the time and effort to read and examine my dissertation and for providing me with your inventive and enriching comments. My very special thanks are for my parents whom I owe everything I am today, my father and my mother, Bancha and Angelita G. Somsorn. There is no single day that you would never support and encourage me with your best wishes and your love. You both are the most wonderful things that ever occurred in my life.

CONTENTS

Page

TABLE OF FIGURES

TABLE OF TABLES

Chapter 1 Introduction

1.1. Introduction

Graphical user interface (GUI) is an important part of a software system. It makes software applications easy to use by providing a front end that receives events from users and interacting with the underlying application through messages or method calls. Compared to traditional software systems, GUI applications have wider range of user bases which increase the chance of encountering failures and repeated requirement changes. This results in frequent code modifications that may introduce new faults which lead to new failures in already tested application. Nonetheless, it is imperative that testing for their correctness be essential to ensure safety, robustness, and usability of the software. The process of testing a software system after changes has two main parts: (1) regression testing for ensuring that the modifications have not affected existing software functionalities, and (2) non-regression testing which make sure that new functionalities are implemented correctly.

The nature of GUI applications poses unique challenges for regression testing. Firstly, because GUI inputs and outputs depend on the graphical layout of components, the expected results of existing test cases may become obsolete when there are changes in input-output mapping. Secondly, in addition to technical understanding, GUI application testers is required to understand the normal mode of operation in order to produce failures that are not expected by the developer team. Lastly, detecting frequent code modifications and adapting the old test cases to them or create new ones demand efficient testing mechanisms.

From the business perspective, releasing software quickly has the benefits of an earlier market introduction. However, hurriedness of releasing may lead to insufficient testing time and inadequate software quality. The software quality depends on many factors, such as the intricacy of the requirements, the complexity of the code, the level of reliability that needs to be reached, and the target release date of the software. An exhaustive testing, while providing the best software quality, requires too much time, cost, and effort that can cause loss effect. Therefore, determining the appropriate time to stop testing is important for maximizing the profits from early software release and reducing the risks of inadequate software quality.

In this work, a new method to assess when regression testing should be stopped is proposed. By measuring estimated failure intensity and participating test cases in many sequences, test effect when failures are detected from the test cases will appear. In this study, each sequence contains many test iterations. The number of iterations depends on the number of failures. Statistics are collected, namely, failure intensity and cumulative average failure to determine the reliability of test results. Details will be discussed in the section that follows.

1.2. Problem Statement

Given a GUI application, this research focuses on the following problems:

1) Determine the trial-and-error threshold limits on the number of regression test iterations and test expenditure to ensure acceptable regression test outcome.
2) Determine the appropriate stopping criteria of the regression testing for GUI-based applications, provided that the test runs must not exceed the predefined threshold limits.

1.3. Expected Benefits

The following benefits are expected from this research:

1) Decrease the costs, especially cost of editing and testing.
2) Save testing time.
3) Decrease the risk of releasing software with poor quality.

1.4. Scope of Research

This research will limit the scope to the followings:

1) Testing will be done on GUI in JAVA language to maintain compatibility with Netbeans IDE 8.0.2.
2) The size of source file is less than 6,000 LOC.

3) The thresholds of testing are limited to:
   3.1. The proposed method will employ JAVA GUI applications, namely, JSyntaxPane and JExifViewer.
   3.2. The number of test sequences for each test iteration is 3.
   3.3. Testing cost ≤ \$600 [1].
4) Fault seeding is placed based on the technique recommended by fault distribution of bug taxonomy [1]. Select faults that have the most dispersion value as shown in Table 1.

1.5. Contributions

The main contributions of this research include:

1) Stopping criteria for GUI regression testing.
2) Formula for computing software reliability threshold.

1.6. Research Plan

Table 1 depicts the research plan and its corresponding schedule.

Table 1 Research Plan

Step	description	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	Research problem identification	■	■															
2	Literature review		■	■	■	■	■	■	■	■								
3	Establish research methodology				■	■	■	■	■	■								
4	Choose program under test					■	■	■	■	■	■							
5	Perform the experiment									■	■	■	■	■				
6	Analyze experimental results												■	■	■	■	■	
7	Prepare draft for conference paper												■	■	■	■	■	
8	Thesis write-up				■	■	■	■	■	■	■	■	■	■	■	■	■	■

1.7. Document organization

This document is organized as follows. Chapter 2 reviews some related work. The proposed methodology is described in Chapter 3. Chapter 4 shows the experiment and the results so obtained. Some concluding remarks and future work are given in Chapter 5.

Chapter 2 Literature Review

There are three issues that involve with this work: regression testing, GUI Testing, and criteria for when to stop testing.

2.1 Regression Testing

Regression testing focuses mainly on testing to ensure that modifications of the previous version of the application do not alter existing software functionalities. Normally, regression testing is done by rerunning old test cases. As the software system grows, the number of test cases increases tremendously. Of these test cases, only a fraction is relevant to the modifications. To save time and resources, test case selection must be employed to select only the test cases that are pertinent to the modifications. Many techniques have been proposed in the literature based on methods such as textual differencing, dataflow analysis, etc. A detailed list of regression test selection techniques can be found in Biswas, et al [2]. A few related techniques are discussed in the following subsections.

2.1.1 Techniques based on textual differencing

Techniques based on textual differencing in the easiest form directly compare two versions of the program under test – original and modified versions – including irrelevant differences such as comments, styles, and formatting. To avoid these extraneous differences, the code is first transformed into their respective canonical form before comparing [3, 4] to guarantee that the same syntactic and formatting guidelines are applied to the original and the modified versions. The canonical form of original version is instrumented and then executed to produce test coverage information. Code modifications are located by syntactically comparing the canonical forms of original and modified versions. Relevant test cases that exercise the altered code are identified by using test coverage information. In this research, this technique is not applicable to the GUI testing.

2.1.2 Techniques based on program dependence graphs

For object-oriented programming, the original and the modified versions of the program can be modelled by constructing Program Dependence Graph (PDG) for the application program and the derived classes [5, 6]. The advantage of using PDG is that it models control dependence and data dependence in one graph. To select the tests, information pertaining to test history in terms of PDG predicates and regions traversed in the original version is used. This PDGs information is then compared with that of the modified version to uncover the regions from which different results may occur. However, the PDG technique is not very efficient in a large system due to considerable overhead during program dependence analysis.

2.2 GUI Testing

There are several works relating to GUI regression testing. White [7] proposed a method using Latin Square to perform automated regression test generation to handle GUI static and dynamic event interactions. A method based on function diagram proposed by Hui, et al. [8] could improve the efficiency of object–oriented software. Their method compared software function diagram of the previous version with the modified version to determine which test cases should be used. Memon, et al. [9] used GUI control flow graph (G-CFG) and GUI call-graph to represent the event behavior and the invoking behavior of components. The original and modified GUI representations were compared to detect obsolete test cases and modified accordingly so that these test cases could be reused. However, constructing G-CFG of the application under test could be time-consuming for large applications and therefore was not very practical in some cases. Instead of G-CFG, Falah, et al. [10] proposed Event Interaction Graph (EIG) to identify infeasible and unusable test cases. The edges of the EIG that were not covered by the usable test cases were used to generate new test cases to achieve edge coverage.

2.3 Fault seeding

Fault seeding [11, 12] is one of software testing techniques that inserts faults as a controlled variable in the program under test. It is based on planting errors with a robustly human knowledge of the programming language and nature of the system to be seeded. This technique relies on the assumption that if known and controlled number of seeded faults are inserted and measured the proportion of these faults discovered by the test process, that proportion could be used to predict the number of real (non-seeded) faults yet to be exposed. Properly used, fault insertion can give an insight as to where testing should be concentrated and how much testing should be done. For fault-seeding purposes, faults should be "representative" of naturally-occurring faults; otherwise, any results obtained from the seeded faults may to be inaccurate or biased.

2.4 Criteria for when to stop testing

The question of when to stop testing involves many factors. Some of them are related to economic reasons, such as the cost of continued testing and the expected losses due to faults that remain in the modified program. Others depend on the quality of the software system, such as fault detection rate, mean time between failures (MTBF), the complexity and difficulty of the system, and the severity of failures that may occur.

One way to determine the appropriate stoppage is by quantifying the reliability of a software system. This leads to the development of models collectively known as Software Reliability Models (SRMs). These models try to estimate system reliability by fitting a theoretical distribution to failure data and use it to design stopping criteria of testing.

The followings assumptions are used in software reliability modeling [8, 9]

1) The software system is subject to failures at random times caused by the manifestation of remaining faults in the system.
2) The total number of faults at the beginning of testing is finite and the failures caused by it are also finite.
3) The mean number of expected failures in the time interval $(t, t+\Delta t]$ is proportional to the mean number of remaining faults in the system. It is equal likely that a fault will generate more than one failure and a failure may be caused by a series of dependent faults.
4) Each time a failure occurs, the fault that caused it is perfectly removed and no new faults are introduced.

From the above assumptions , the following parameters are defined :

$\mathrm{m}(\mathrm{t})$ is the expected number of software failures at time t,

r is the failure detection rate per remaining fault,

a is the expected number of initial faults,

α is the quantified ratio of faults to failures, and

$\lambda(\mathrm{t})$ is the failure intensity function.

The expected number of failure found from the start of the test until time ***t*** can be calculated from $\alpha \times m(t)$.The number of remaining failures in assumption 3 can then be determined by subtracting the expected number of failure found at the time from the number of initial faults, yielding $a-\alpha \times m(t)$. Using *r* as the proportionality constant in assumption 3, the following relationships can be derived:

$$\frac{dm(t)}{dt} = r \times \left(a - \alpha \times m(t)\right) \tag{1}$$

which, by solving under boundary condition $\mathrm{m}(0)=0$, leads to

$$m(t) = \frac{a}{\alpha} \times \left(1 - \exp\left(-r\alpha t\right)\right) \tag{2}$$

Since $\lambda(t)$ is defined as the derivative of $m(t)$ with respect to **t**, $\lambda(t)$ becomes

$$\lambda(t) = ar \times \exp\left(-r\alpha t\right) \tag{3}$$

Software reliability can be defined as follows [13]:

$$R(\Delta t|t) = \exp\left(-\left(m(t+\Delta t) - m(t)\right)\right) \tag{4}$$

where $\Delta t \geq 0$, $t > 0$. The function $R(\Delta t \mid t)$ represents the probability that a software failure doesn't occur during the time interval $(t, t+\Delta t]$.

During testing the software, it is often assumed that fault correction process does not introduce any new faults and software reliability increases as faults are uncovered and fixed. Unfortunately, in practice, it is difficult to meet the assumptions of the above ideal case.

Lin and Huang [14] proposed using $\lambda(\mathrm{t})$ and $R(\Delta t|t)$ as stopping criteria by calculating the time needed for the software to meet failure intensity objective and acceptable reliability as follows. If the failure intensity objective is F_0, and T_1 which is the time to meet the desired failure intensity satisfying $\lambda(T_1) = F_0$ can be determined from

$$T_1 = -\frac{\ln\left(\dfrac{F_0}{ar}\right)}{r\alpha} \tag{5}$$

If the acceptable reliability R_0 is given, T_2 which is the time to meet the desired reliability satisfying $R(\Delta t \mid T_2) = 0$ can be obtained from

$$T_2 = \frac{\ln\left(\dfrac{a\left(1-\exp\left[-r\alpha\times\Delta t\right]\right)}{-\alpha\times\ln R_0}\right)}{r\alpha} \tag{6}$$

The above Equation (1) - (6) were taken mostly from reliability theory and set up to be adopted by the proposed methodology in the next chapter.

Chapter 3 Research Methodology

In this research, a model to determine a set of stopping test criteria in order to guarantee software application reliability is proposed. Several factors affecting software reliability are considered, namely, number of faults, number of failures, testing time, editing time, fault detection rate (FDR), failure intensity, testing cost, editing cost, and reliability. A concise description of each factor is given below. A fault is defined as a mistake in the software application, and a failure occurs when the application does not comply with the specifications due to a fault or combination of faults. Testing time is the time the test team needs to execute the previously planned test cases. Editing time is the time the developer team needs to edit the software application. Failure intensity is the number of failures divided by testing time. Fault detection rate is the number of faults divided by the sum of testing time and editing time. Testing cost and editing cost are estimated from testing time and editing time using average salary given in [1].

3.1. Cost estimation

As test process may continue when all test executions are closed to satisfying the predetermined conditions, the expense escalates. One way to stop the infinitesimal on-going test is setting a limit for test costs. This limit is not known in advance. A probable solution is by estimating the expected cost incurred during such indefinite repetitions. The estimation can be performed based on various parameters used in most of the related work. The equation proceeds as follows.

$$Expected\ Cost = \left(C_{testing} \times T_t\right) + \left(C_{editing} \times T_e\right) \quad (7)$$

where T_t is the expected testing time estimated from the failure intensity function $\lambda\left(\mathrm{t}\right)$ of equation (3) and the failure intensity objective F_0, which is set to 0.01 in this study, *$C_{testing}$ and $C_{editing}$ are cost of testing and editing, respectively, and T_e* is the expected editing time estimated from expected number of remaining faults divided by the editing speed of the previous iteration. Finding T_t such that $\lambda\left(\mathrm{T}_\mathrm{t} + \mathrm{T}_\mathrm{tp}\right) \le \mathrm{F}_0$ yields

$$T_t = \left(-\frac{\ln\left(\frac{F_0}{ar}\right)}{r\alpha} \right) - T_{tp} \,. \tag{8}$$

where $\mathrm{T_{tp}}$ is the summation of actual testing time of the previous iterations, and $\mathrm{T_e}$ can be computed by the following equation:

$$T_e = \frac{\#remaining\ faults}{v_{previous}} \tag{9}$$

3.2. Reliability Models

The reliability function [13] is modified to use stretched exponential function known as the complementary cumulative Weibull distribution [15]. Because of its versatile ability to take on the characteristics of other distributions, Weibull distribution has become one of the most widely used distributions in reliability engineering. The distribution characteristics depend on the value of the parameters. Here, the 2-parameter Weibull being used are the shape parameter β and the scale parameter η. Thus, the modified reliability function becomes

$$R(\Delta t|t) = \exp\left(-\eta\left(m(t+\Delta t)^{\beta} - m(t)^{\beta}\right)\right) \tag{10}$$

where >0 and $\eta > 0$. In this study, the proper value obtained from preliminary experiment are $\beta = 0.75$ and $\eta = 0.1$.

3.3. Applying fault seeding

Fault seeding technique was carefully distributed in the regression test process based on Bug taxonomy [16]. The advantage is that seeder could be directed to seed some precise kind of faults, and would be able to classify faults once seeded and checked for any gap in the coverage. Second, a seeder could generate the same kind of errors not as an automated task, but considering different context in which same type of errors could lead to different results; and finally, a seeder could assure the selection of all kind of errors by classifying them and weeding out the excesses, then granting error representativeness.

Seeded faults are injected into production software as follows.

1) Run all test case and collect coverage data.
2) Sort the classes in the production software in decreasing order of coverage percentage.
3) Choose 5 classes with the most coverage percentage.
4) Add seeded fault which have distribution from bug taxonomy [16] into the chosen classes by scattering the fault from ratio of coverage percentage and size of class (in LOC).

3.4. The proposed stopping criteria

The stopping criteria are set up as follows:

1) If failure intensity $\lambda\left(\mathrm{t}\right)$ is less than or equal to cumulative average failure intensity in current iteration, then consider the total cost of editing and testing as follows:
    a) If the cumulative costs in current iteration plus the expected cost of the next iteration are less than or equal to threshold cost, determine reliability *R(t)* as the stopping criterion
        i) If $\mathrm{R}\left(\Delta \mathrm{t} \mid \mathrm{t}\right)$ is greater than or equal to R_1, stop;
        ii) If $\mathrm{R}\left(\Delta \mathrm{t} \mid \mathrm{t}\right)$ is less than R_1, continue testing; or
    b) If the cumulative costs in current iteration plus the expected cost of the next iteration are greater than threshold cost, determine reliability *R(t)* as the stopping criterion
        i) If $\mathrm{R}\left(\Delta \mathrm{t} \mid \mathrm{t}\right)$ is greater than or equal to R_2, stop;
        ii) If $\mathrm{R}\left(\Delta \mathrm{t} \mid \mathrm{t}\right)$ is less than R_2, continue testing.
2) If failure intensity $\lambda\left(\mathrm{t}\right)$ is greater than cumulative average failure intensity in current iteration,
    a) If the cumulative cost is less than or equal to threshold cost, continue testing; or
    b) If the cumulative cost is greater than threshold cost, determine reliability *R(t)* as the stopping criterion
        i) If $\mathrm{R}\left(\Delta \mathrm{t} \mid \mathrm{t}\right)$ is greater than or equal to R_2, stop;
        ii) If $\mathrm{R}\left(\Delta \mathrm{t} \mid \mathrm{t}\right)$ is less than R_2, continue testing.

The threshold values for reliability are computed from Eq.10 using the expected number of initial faults in the program under test. Suppose there are f_0 faults in the production software, R_1 and R_2 can be defined as follows:

$$R_1 = \exp\left(-\eta\left(0.03 \times f_0 + 0.0004 \times LOC\right)^{\beta}\right) \tag{11}$$

and

$$R_2 = \exp\left(-\eta\left(0.06 \times f_0 + 0.00075 \times LOC\right)^{\beta}\right) \tag{12}$$

The constants used in the above equations were determined from production software in a preliminary test.

3.5. Research Methodology

The research methodology is shown in Figure 1. The process starts from production software. It is used in a preliminary test to decide the threshold limit of initial total cost and software reliability. Test code is added to make it an enhanced version. Seeded faults are injected which will be tested by selected data set and test cases. The selection process considers how each GUI function of the software works. A set of test cases is then created based on the guidelines in [16] to comply with the software function. Since execution sequence of the test cases affects the occurrence of faults and failures, all test cases will be organized into many sequences of tests in different orders. Some test case prioritization techniques from [17, 18] are employed.

1) Prioritize in order of coverage of statements: measure statement coverage in a program under test by instrumenting the program. Test cases are prioritized by sorting the total number of coverage statements in decreasing order.
2) Prioritize in order of coverage of branches: Same as 1 above, but use the number of decisions (branches) in the program that are exercised by each test case.
3) Prioritize in order of coverage of functions: Same as 1 above, but use the number of functions that are executed by each test case.

These 3 prioritization techniques were chosen of their performance in terms of average percentage faults detected (APFD), while not introducing too much complexity and overhead in the prioritization process. Detailed comparisons of various test case prioritization techniques can be found in [17, 18].

The actual regression test proceeds as follows. Starting with the first sequence, the first test case is executed. If a fault occurs, the corresponding faulty code is fixed. The second test case is then executed. This process repeats until all test cases in the first sequence are exhausted. The first regression test iteration is said to finish. Meanwhile, test statistics are collected to analyze if the test stopping criteria are met and the entire process terminates. Otherwise, the test continues on next iteration.

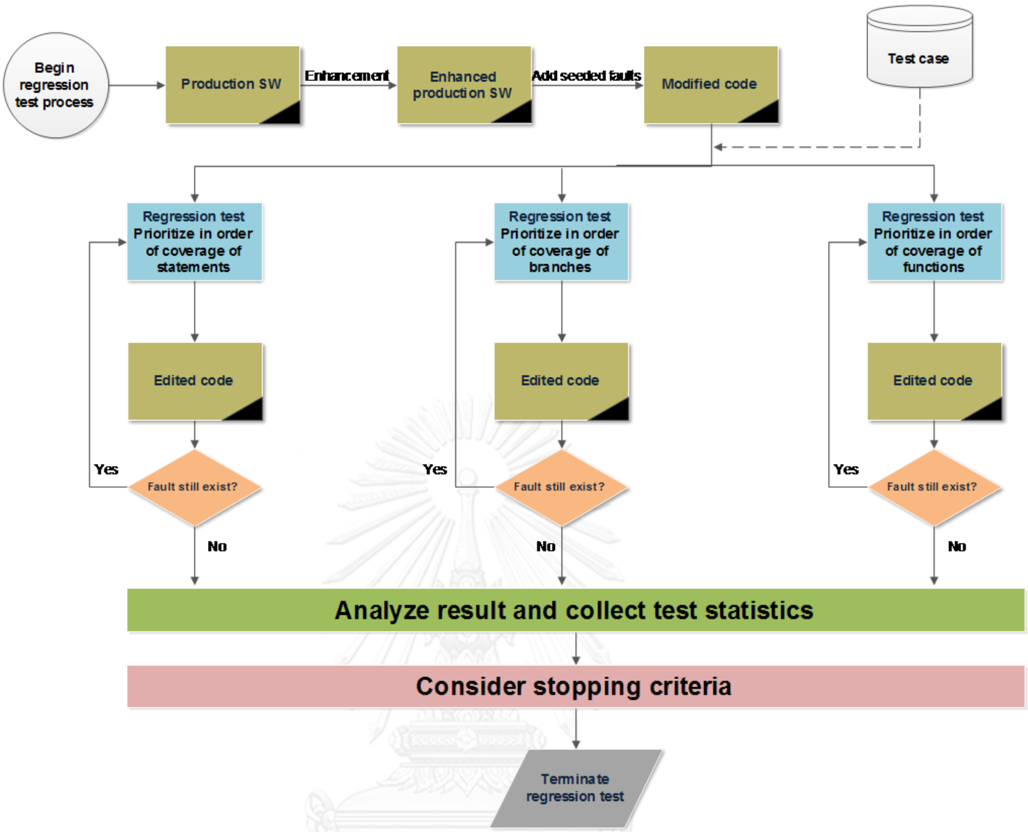


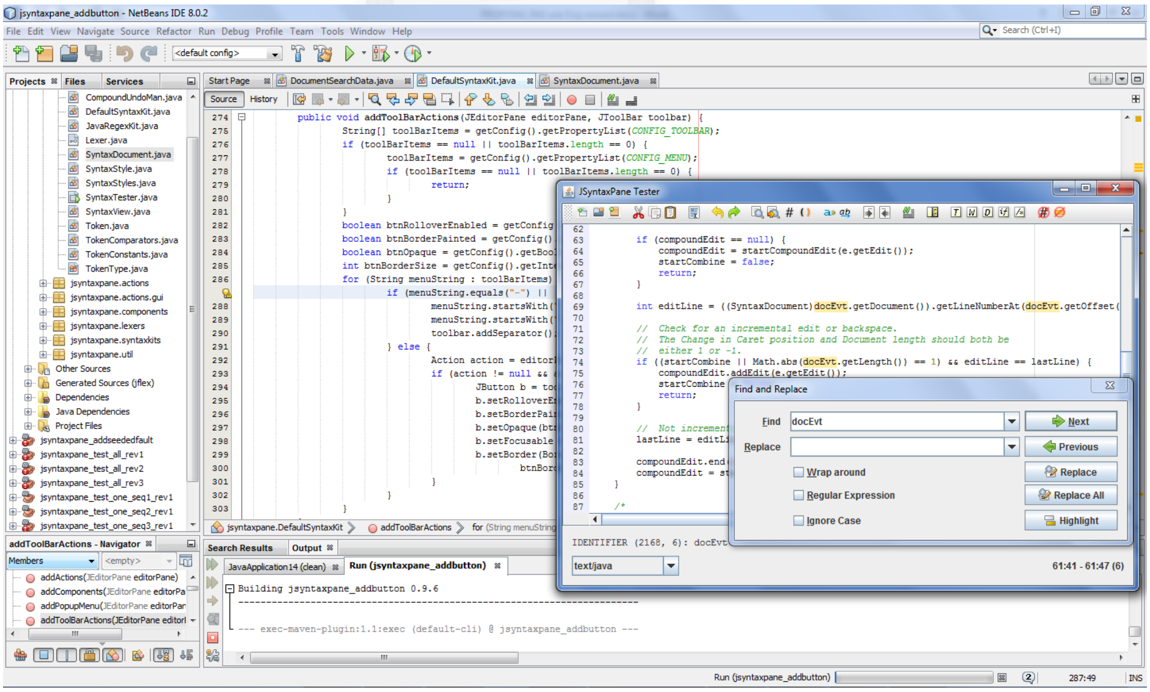
Figure 1: Research methodology

Chapter 4 Experiments and Results

The proposed method was tested with two open-source GUI applications named JSyntaxPane [9] and JExifViewer [19]. JSyntaxpane was set up to run randomized test sequence, while JExifviewer employed prioritized the test sequences in order to measure how different input test sequences affected the outcome. The set up will be described subsequently. Fault seeding was performed to initialize the test process and the regression test began as described earlier. The test toolset and their environment were NetBeans IDE 8.0.2 [9] running on Windows7 64-bit operating system with Intel(R) Core(TM) i7-3520M CPU and 8.00 GB RAM. Code coverage was measured using JaCoCo [20] plugin for NetBeans, which is a free code coverage library for Java.

4.1. JSyntaxPane

JSyntaxPane is a sub-class of Java jEditorPane with added support for syntax highlighting of 22 file types. Each file type has its own lexical analyzer to serve different functionalities. Additional functionalities can also be added. This application consists of 99 classes of size approximately 3,550 lines of code.



```
        public void addToolBarActions(JEditorPane editorPane, JToolBar toolbar) {
                String[] toolBarItems = getConfig().getPropertyList(CONFIG_TOOLBAR);
                if (toolBarItems == null || toolBarItems.length == 0) {
                        toolBarItems = getConfig().getPropertyList(CONFIG_MENU);
                        if (toolBarItems == null || toolBarItems.length == 0) {
                                return;
                        }
                }
                boolean btnRolloverEnabled = getConfig
                boolean btnBorderPainted = getConfig()
                boolean btnOpaque = getConfig().getBool
                int btnBorderSize = getConfig().getInte
                for (String menuString : toolBarItems)
                        if (menuString.equals("-") ||
                                menuString.startsWith(
                                menuString.startsWith(
                                toolbar.addSeparator()
                        } else {
                                Action action = editor
                                if (action != null && 
                                        JButton b = to
                                        b.setRolloverE
                                        b.setBorderPai
                                        b.setOpaque(bt
                                        b.setFocusable
                                        b.setBorder(Bo
                                                btnBor
                                }
                        }
                }
```

```

        if (compoundEdit == null) {
            compoundEdit = startCompoundEdit(e.getEdit());
            startCombine = false;
            return;
        }

        int editLine = ((SyntaxDocument)docEvt.getDocument()).getLineNumberAt(docEvt.getOffset(

        //  Check for an incremental edit or backspace.
        //  The Change in Caret position and Document length should both be
        //  either 1 or -1.
        if ((startCombine || Math.abs(docEvt.getLength()) == 1) && editLine == lastLine) {
            compoundEdit.addEdit(e.getEdit());
            startCombine
            return;
        }

        //  Not increment
        lastLine = editLi

        compoundEdit.end
        compoundEdit = st
    }

    /*
```

Find: docEvt

Replace:

☐ Wrap around

☐ Regular Expression

☐ Ignore Case

```
IDENTIFIER (2168, 6): docEvt
```

text/java

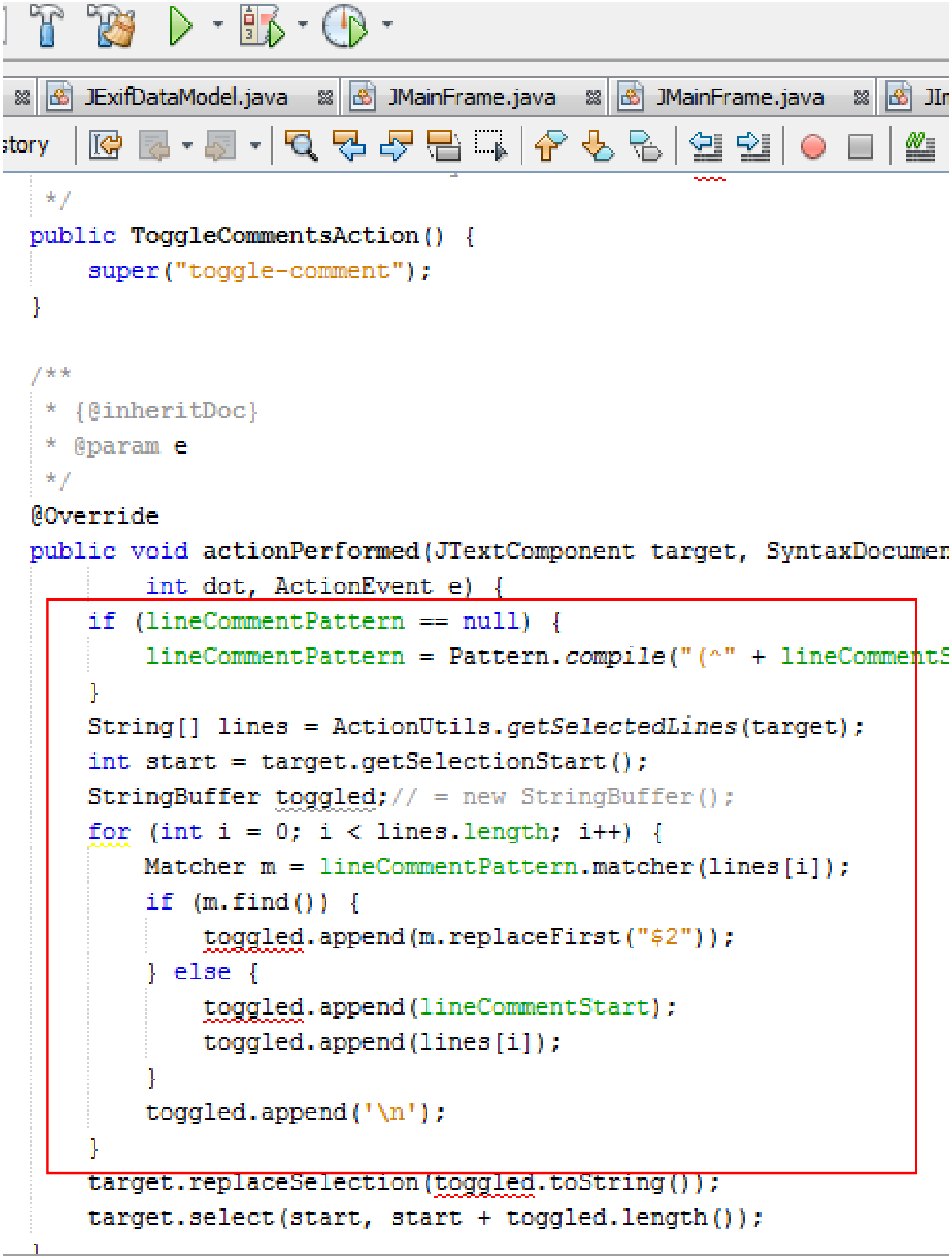
61:41 - 61:47 (6)

```
Building jsyntaxpane_addbutton 0.9.6
------------------------------------------------------------------------

--- exec-maven-plugin:1.1:exec (default-cli) @ jsyntaxpane_addbutton ---
```

Figure 2: Example of JSyntaxPane invoked by NetBeans

The version of JSyntaxPane used in the experiment contained two types of faults, namely, initial faults and seeded faults. An initial fault is an unintended fault that exists in the application before enhancement. Bug reports provided in the application project page and selected test cases were employed to uncover the initial faults. The following sample statements contain some of the initial faults shown in Figure 3.

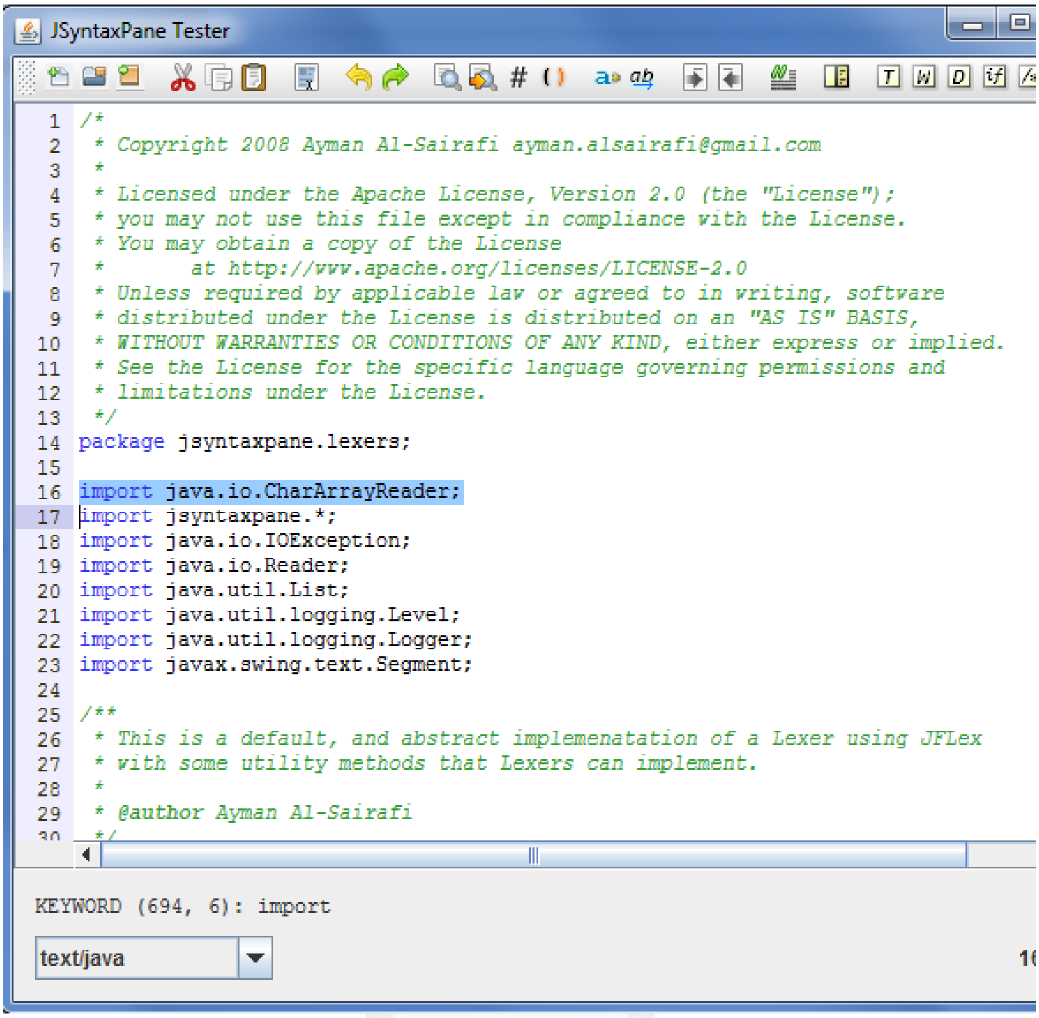


```
 */
public ToggleCommentsAction() {
    super("toggle-comment");
}

/**
 * {@inheritDoc}
 * @param e
 */
@Override
public void actionPerformed(JTextComponent target, SyntaxDocumen
        int dot, ActionEvent e) {
    if (lineCommentPattern == null) {
        lineCommentPattern = Pattern.compile("(^" + lineCommentS
    }
    String[] lines = ActionUtils.getSelectedLines(target);
    int start = target.getSelectionStart();
    StringBuffer toggled;// = new StringBuffer();
    for (int i = 0; i < lines.length; i++) {
        Matcher m = lineCommentPattern.matcher(lines[i]);
        if (m.find()) {
            toggled.append(m.replaceFirst("$2"));
        } else {
            toggled.append(lineCommentStart);
            toggled.append(lines[i]);
        }
        toggled.append('\n');
    }
    target.replaceSelection(toggled.toString());
    target.select(start, start + toggled.length());
}
```

Figure 3 Example code of initial fault

As shown in this figure, this production code will not perform toggle comment function as it is supposed to after running through NetBeans. This is shown in Figure 4 where the highlighted code on line 16 is not commented out due to the initial fault.



```
/*
 * Copyright 2008 Ayman Al-Sairafi ayman.alsairafi@gmail.com
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License
 *        at http://www.apache.org/licenses/LICENSE-2.0
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package jsyntaxpane.lexers;

import java.io.CharArrayReader;
import jsyntaxpane.*;
import java.io.IOException;
import java.io.Reader;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.text.Segment;

/**
 * This is a default, and abstract implemenatation of a Lexer using JFLex
 * with some utility methods that Lexers can implement.
 *
 * @author Ayman Al-Sairafi
 */
```

```
KEYWORD (694, 6): import
```

Figure 4 GUI of JSyntaxpane after running through NetBeans

Seeded faults were added during test execution according to the average fault distribution of the software systems provided in [1].

Seeded faults are injected into production software as follows.

1. Run all test cases and collect coverage data. Using JaCoCo coverage to record percentage of data which being covered in each test case and each class of the production software.
2. Sort the classes in the production software from largest to smallest in decreasing order of coverage percentage to obtain the sequence of classes.
3. Choose 5 classes with the most coverage percentages.
4. Add the seeded faults according to the distribution from bug taxonomy [16] into the chosen classes by scattering the faults based on ratio of coverage percentage and size of class (in LOC).

Here are sample seeded faults being injected into the test code which shown in Figure 5 .

```
/**
 * Sets the pattern from a string and flags
 * @param pat String of pattern
 * @param regex true if the pattern should be a regexp
 * @param ignoreCase true to ignore case
 * @throws java.util.regex.PatternSyntaxException
 */
public void setPattern(String pat, boolean regex, boolean ignoreCase)
        throws PatternSyntaxException {
        if (pat != null && pat.length() > 0) {
                int flag = (regex) ? 0 : Pattern.LITERAL;
                flag |= (ignoreCase) ? Pattern.CASE_INSENSITIVE : 0;
                setPattern(Pattern.compile(pat, flag));
        } else {
                setPattern(null);
        }
}
```

```
 */
public void setPattern(Pattern pattern) {
        this.pattern = pattern;
}

/**
 * Sets the pattern from a string and flags
 * @param pat String of pattern
 * @param regex true if the pattern should be a regexp
 * @param ignoreCase true to ignore case
 * @throws java.util.regex.PatternSyntaxException
 */
blic void setPattern(String pat, boolean regex, boolean ignoreCase)
        throws PatternSyntaxException {
        if (pat != null && pat.length() > 0) {
                int flag = (regex) ? 0 : Pattern.LITERAL;
                flag &= (ignoreCase) ? Pattern.CASE_INSENSITIVE : 0;
                setPattern(Pattern.compile(pat, flag));
        } else {
                setPattern(null);
        }
}
```

Figure 5 Example of fault seeding (logic or boolean fault)

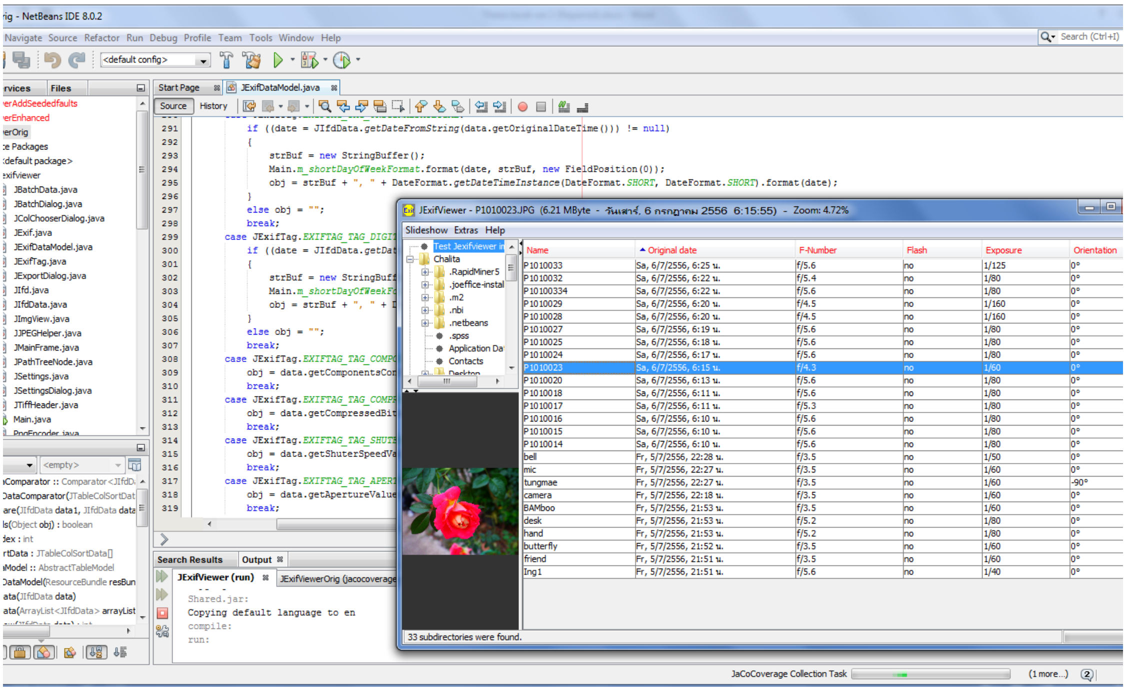
The type of seeded faults which are shown in this figure is either logic or boolean faults. This fault in turn causes failure which appears in find and replace function#4 (test case No.9 in JSyntaxpane of the appendix). There were 21 and 19 lines of code that contained initial faults and seeded faults, respectively. The total 40 faults produced 37 failures in the application. Table 2 summarizes the types of faults in the experiment.

Table 2: Faults distribution for JSyntaxPane

Type of faults	#lines	
	Initial faults	Seeded faults
FUNCTIONALITY AS IMPLEMENTED		
Feature misunderstood, wrong	9	
Feature interactions	4	
Missing feature	8	
STRUCTURAL BUGS		
Control logic and predicates		2
Loops and iterations		1
Arithmetic expressions		2
Logic or Boolean, not control		1
Initialization		1
Other processing		6
DATA		
Other data definition, structure, declaration bugs		1
Value		2
Wrong object accessed		1
Other access and handling		2
Total	21	19
	Total #lines	40

4.2. JExifViewer

JExifViewer is a Java program for displaying and comparing Exif information stored in JPEG files created by digital cameras. This program also has an image viewer which can rotate and/or flip, zoom in/out the selected image, and other basic file operations such as rename, copy, move, and delete images. This application consists of 210 classes of size approximately 5,256 lines of code.



```
                    if ((date = JIfdData.getDateFromString(data.getOriginalDateTime())) != null)
                    {
                        strBuf = new StringBuffer();
                        Main.m_shortDayOfWeekFormat.format(date, strBuf, new FieldPosition(0));
                        obj = strBuf + ", " + DateFormat.getDateTimeInstance(DateFormat.SHORT, DateFormat.SHORT).format(date);
                    }
                    else obj = "";
                    break;
                case JExifTag.EXIFTAG_TAG_DIGIT
                    if ((date = JIfdData.getDat
                    {
                        strBuf = new StringBuff
                        Main.m_shortDayOfWeekFo
                        obj = strBuf + ", " + D
                    }
                    else obj = "";
                    break;
                case JExifTag.EXIFTAG_TAG_COMPO
                    obj = data.getComponentsCon
                    break;
                case JExifTag.EXIFTAG_TAG_COMPR
                    obj = data.getCompressedBit
                    break;
                case JExifTag.EXIFTAG_TAG_SHUTE
                    obj = data.getShuterSpeedVa
                    break;
                case JExifTag.EXIFTAG_TAG_APERT
                    obj = data.getApertureValue
                    break;
```

```
Shared.jar:
Copying default language to en
compile:
run:
```

Name	Original date	F-Number	Flash	Exposure	Orientation
P1010033	Sa, 6/7/2556, 6:25 น.	f/5.6	no	1/125	0°
P1010032	Sa, 6/7/2556, 6:22 น.	f/5.4	no	1/80	0°
P10100334	Sa, 6/7/2556, 6:22 น.	f/5.6	no	1/80	0°
P1010029	Sa, 6/7/2556, 6:20 น.	f/4.5	no	1/160	0°
P1010028	Sa, 6/7/2556, 6:20 น.	f/4.5	no	1/160	0°
P1010027	Sa, 6/7/2556, 6:19 น.	f/5.6	no	1/80	0°
P1010025	Sa, 6/7/2556, 6:18 น.	f/5.6	no	1/80	0°
P1010024	Sa, 6/7/2556, 6:17 น.	f/5.6	no	1/80	0°
P1010023	Sa, 6/7/2556, 6:15 น.	f/4.3	no	1/60	0°
P1010020	Sa, 6/7/2556, 6:13 น.	f/5.6	no	1/80	0°
P1010018	Sa, 6/7/2556, 6:11 น.	f/5.6	no	1/80	0°
P1010017	Sa, 6/7/2556, 6:11 น.	f/5.3	no	1/80	0°
P1010016	Sa, 6/7/2556, 6:10 น.	f/5.6	no	1/80	0°
P1010015	Sa, 6/7/2556, 6:10 น.	f/5.6	no	1/80	0°
P1010014	Sa, 6/7/2556, 6:10 น.	f/5.6	no	1/80	0°
bell	Fr, 5/7/2556, 22:28 น.	f/3.5	no	1/50	0°
mic	Fr, 5/7/2556, 22:27 น.	f/3.5	no	1/60	0°
tungmae	Fr, 5/7/2556, 22:27 น.	f/3.5	no	1/60	-90°
camera	Fr, 5/7/2556, 22:18 น.	f/3.5	no	1/60	0°
BAMboo	Fr, 5/7/2556, 21:53 น.	f/3.5	no	1/60	0°
desk	Fr, 5/7/2556, 21:53 น.	f/5.2	no	1/80	0°
hand	Fr, 5/7/2556, 21:53 น.	f/5.2	no	1/80	0°
butterfly	Fr, 5/7/2556, 21:52 น.	f/3.5	no	1/60	0°
friend	Fr, 5/7/2556, 21:51 น.	f/3.5	no	1/60	0°
Ing1	Fr, 5/7/2556, 21:51 น.	f/5.6	no	1/40	0°

Figure 6: Example of JExifViewer invoked by NetBeans

For JExifViewer, there were total of 16 faults which caused 9 failures during execution. Table 3 summarizes each types of faults in the experiment.

Table 3: Faults distribution for JExifViewer

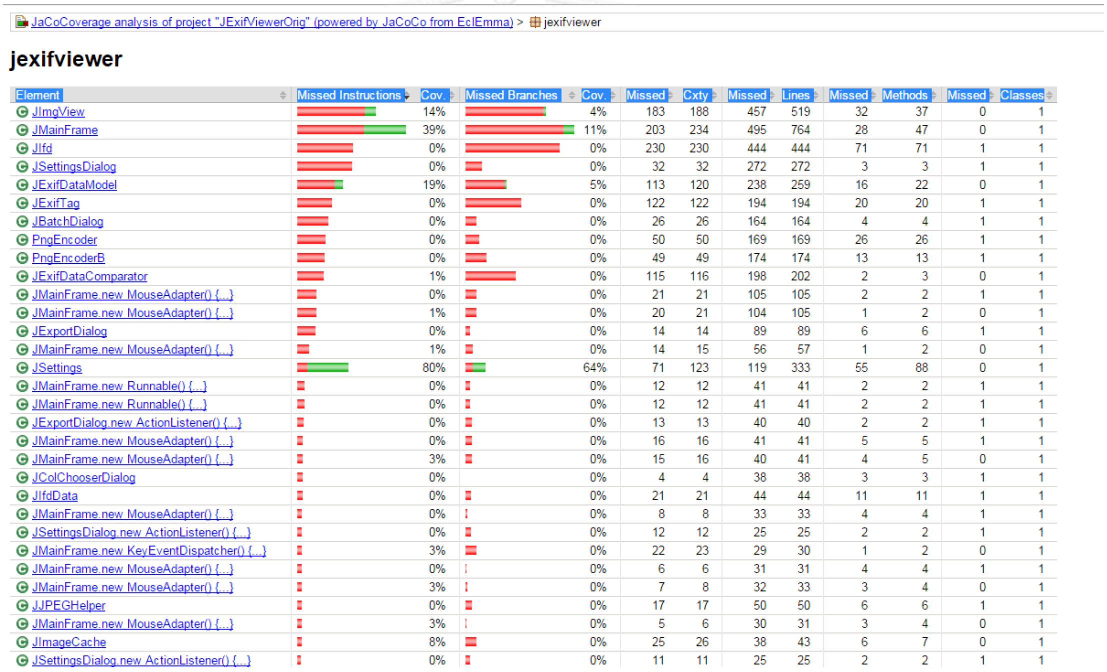
Type of faults	**#lines**	
	Initial faults	**Seeded faults**
FUNCTIONALITY AS IMPLEMENTED		
Missing feature		7
STRUCTURAL BUGS		
Control logic and predicates		5
Arithmetic expressions	1	1
DATA		
Value		2
Total	1	15
	Total #lines	16

Coverage criteria were measured using JaCoCo[20]. When running Jexifviewer through NetBeans IDE by the function 'run with JaCoCo coverage', JaCoCo would instrument the code in Jexifviewer to measure several coverage criteria, namely, instructions, branches, cycromatic complexity, lines of code, methods, and classes. The coverage criteria used for each prioritization technique are shown in Table 4

Table 4 JaCoCo coverage data used for test case prioritization technique

Techniques	Data column
Prioritize in order of coverage of statements:	Instructions
Prioritize in order of coverage of branches:	Branches
Prioritize in order of coverage of functions:	Methods

The results are shown as HTML files in Figure 7.



JaCoCoverage analysis of project "JExifViewerOrig" (powered by JaCoCo from EclEmma) > jexifviewer

jexifviewer

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
JImgView		14%		4%	183	188	457	519	32	37	0	1
JMainFrame		39%		11%	203	234	495	764	28	47	0	1
JIfd		0%		0%	230	230	444	444	71	71	1	1
JSettingsDialog		0%		0%	32	32	272	272	3	3	1	1
JExifDataModel		19%		5%	113	120	238	259	16	22	0	1
JExifTag		0%		0%	122	122	194	194	20	20	1	1
JBatchDialog		0%		0%	26	26	164	164	4	4	1	1
PngEncoder		0%		0%	50	50	169	169	26	26	1	1
PngEncoderB		0%		0%	49	49	174	174	13	13	1	1
JExifDataComparator		1%		0%	115	116	198	202	2	3	0	1
JMainFrame.new MouseAdapter() {...}		0%		0%	21	21	105	105	2	2	1	1
JMainFrame.new MouseAdapter() {...}		1%		0%	20	21	104	105	1	2	0	1
JExportDialog		0%		0%	14	14	89	89	6	6	1	1
JMainFrame.new MouseAdapter() {...}		1%		0%	14	15	56	57	1	2	0	1
JSettings		80%		64%	71	123	119	333	55	88	0	1
JMainFrame.new Runnable() {...}		0%		0%	12	12	41	41	2	2	1	1
JMainFrame.new Runnable() {...}		0%		0%	12	12	41	41	2	2	1	1
JExportDialog.new ActionListener() {...}		0%		0%	13	13	40	40	2	2	1	1
JMainFrame.new MouseAdapter() {...}		0%		0%	16	16	41	41	5	5	1	1
JMainFrame.new MouseAdapter() {...}		3%		0%	15	16	40	41	4	5	0	1
JColChooserDialog		0%		0%	4	4	38	38	3	3	1	1
JIfdData		0%		0%	21	21	44	44	11	11	1	1
JMainFrame.new MouseAdapter() {...}		0%		0%	8	8	33	33	4	4	1	1
JSettingsDialog.new ActionListener() {...}		0%		0%	12	12	25	25	2	2	1	1
JMainFrame.new KeyEventDispatcher() {...}		3%		0%	22	23	29	30	1	2	0	1
JMainFrame.new MouseAdapter() {...}		0%		0%	6	6	31	31	4	4	1	1
JMainFrame.new MouseAdapter() {...}		3%		0%	7	8	32	33	3	4	0	1
JJPEGHelper		0%		0%	17	17	50	50	6	6	1	1
JMainFrame.new MouseAdapter() {...}		3%		0%	5	6	30	31	3	4	0	1
JImageCache		8%		0%	25	26	38	43	6	7	0	1
JSettingsDialog.new ActionListener() {...}		0%		0%	11	11	25	25	2	2	1	1

Figure 7 Example of JaCoCo coverage

Test case prioritization proceeded as shown in Figure 8. Each test case is executed in Netbeans one at a time. The coverage results obtained from JaCoCo are then recorded. When all test case are exhausted, the test sequence for each prioritization technique is determined by ordering the corresponding coverage data of each test case according to the criteria in Table 4

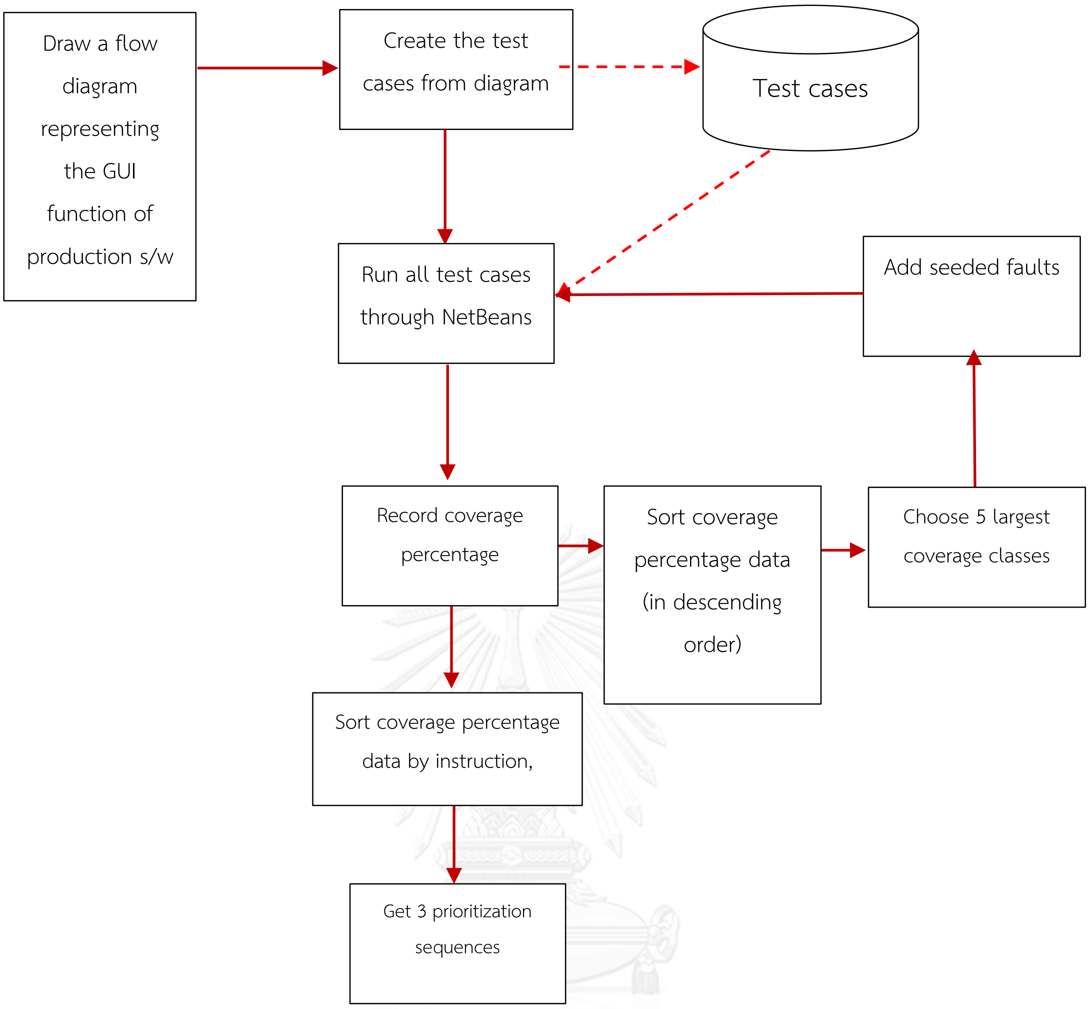


Figure 8 Establishing prioritization sequence

From Figure 8, a flow diagram representing the GUI function of production software is drawn as shown in Figure *9*. Then, test cases are created and run through NetBeans with JaCoCo coverage. Record the coverage percentage data of each class and sort in descending order to find the 5 largest coverage percentage classes. Add seeded faults into these clasess. Run all test cases and record the coverage percentage data of these 5 classes for each test case as shown in Table 5. Sort the covergae percentage data by branches, instructions, and methods. Run all test cases based on the prioritization sequences as shown in Table 5.

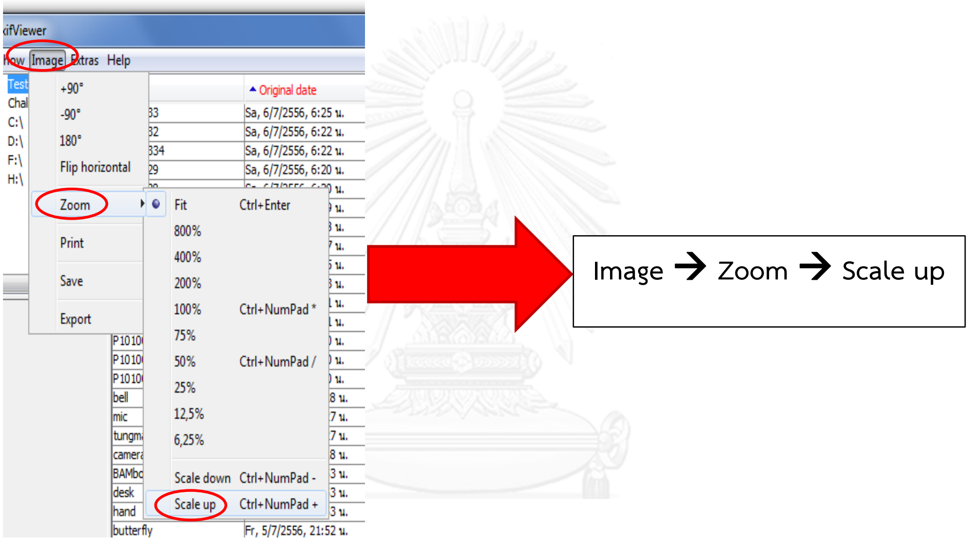


Figure 9 Example of drawing tree diagram

Table 5 Coverage statistic of JExifviewer test cases

	Total							Total							Total						
	29,776	1,584	1,210	1,922	2,692	3,727		2,544	140	190	318	274	370		804	22	20	71	37	47	
	Missed Coverage Instruction							Missed Coverage Branch							Missed Coverage Method						
Sq	All	JExifDataModel	JExifTag	JIfd	JImgView	JMainFrame	TotalEnhancedI	All	JExifDataModel	JExifTag	JIfd	JImgView	JMainFrame	TotalEnhancedB	All	JExifDataModel	JExifTag	JIfd	JImgView	JMainFrame	TotalEnhancedM
1	23,495	1,096	782	1,467	2,310	2,132	7,787	2,245	120	149	273	264	315	1,121	587	9	11	60	30	23	133
2	22,864	908	762	1,202	2,310	2,046	7,228	2,168	102	145	240	264	303	1,054	569	11	9	43	30	22	115
3	22,696	1,110	782	1,457	1,935	1,903	7,187	2,185	121	149	272	234	299	1,075	558	11	11	59	24	20	125
4	21,678	886	762	1,189	1,927	1,682	6,446	2,080	99	145	239	230	274	987	527	10	9	42	24	17	102
5	21,641	886	762	1,187	1,934	1,665	6,434	2,075	101	145	237	232	271	986	527	11	9	42	24	17	103
6	22,285	1,088	782	1,454	1,934	1,684	6,942	2,157	118	149	272	232	276	1,047	547	10	11	59	24	16	120
7	21,613	891	762	1,189	1,924	1,693	6,459	2,068	100	145	239	230	273	987	523	10	9	42	23	17	101
8	22,445	1,110	782	1,454	1,934	1,826	7,106	2,172	121	149	272	232	290	1,064	550	11	11	59	24	18	123
9	22,537	1,110	782	1,454	1,934	1,865	7,145	2,182	121	149	272	232	294	1,068	556	11	11	59	24	19	124
10	22,262	1,110	782	1,454	1,882	1,644	6,872	2,157	121	149	272	224	293	1,059	546	11	11	59	20	19	120
11	24,453	1,250	1,210	1,922	2,310	2,094	8,786	2,353	133	190	318	264	311	1,216	614	14	20	71	30	23	158
12	24,795	1,250	1,210	1,922	2,310	2,130	8,822	2,378	133	190	318	264	314	1,219	626	14	20	71	30	23	158
13	25,240	1,281	1,210	1,922	2,317	2,284	9,014	2,408	133	190	318	264	331	1,236	649	16	20	71	32	28	167
14	21,820	1,110	782	1,454	1,799	1,880	7,025	2,147	121	149	272	219	297	1,058	529	11	11	59	19	20	120
15	21,110	908	762	1,189	1,798	1,794	6,451	2,061	102	145	239	219	285	990	505	11	9	42	19	19	100
16	21,801	1,110	782	1,454	1,784	1,880	7,010	2,145	121	149	272	217	297	1,056	529	11	11	59	19	20	120
17	21,823	1,110	782	1,454	1,801	1,880	7,027	2,148	121	149	272	220	297	1,059	529	11	11	59	19	20	120
18	21,806	1,110	782	1,454	1,835	1,880	7,061	2,154	121	149	272	226	297	1,065	528	11	11	59	19	20	120
19	21,972	1,110	782	1,454	1,922	1,880	7,148	2,161	121	149	272	231	297	1,070	533	11	11	59	21	20	122

Table 6: Test sequences for JExifViewer

TestCaseNo.	Prioritize in order of coverage of statements	Prioritize in order of coverage of branches	Prioritize in order of coverage of functions
1	5	5	15
2	4	4	7
3	15	7	4
4	7	15	5
5	10	6	2
6	6	2	6
7	16	16	16
8	14	14	14
9	17	10	10
10	18	17	17
11	8	8	18
12	9	18	19
13	19	9	8
14	3	19	9
15	2	3	3
16	1	1	1
17	11	11	11
18	12	12	12
19	13	13	13

4.3. Results of

The results of JSyntaxPane are shown in Table 7. The **expected testing time**, **expected editing time**, and **expected cost** of each iteration are computed from previous iteration using equation (5). The cost is estimated in dollars ($) using average salary given in [8]. The variables **#rem, #faults**, and **#fails** denote number of remaining faults at the beginning of each iteration, number of faults that have been corrected, and number of failures that have occurred in each iteration, respectively. $\boldsymbol{\alpha}$ is the ratio of cumulative number of faults in each sequence to cumulative number of failures in that sequence. $\mathbf{r}$ is the failure detection rate per remaining faults. **FDR** is fault detection rate which is the number of faults per minute. **Failure intensity** is the number of failures per minute of testing time. $\boldsymbol{\lambda(t)}$ is the expected failure intensity calculated from equation (3). $\boldsymbol{\lambda(t)}$ **avg** is the average of $\lambda(t)$ from the start of each sequence. $\boldsymbol{m(t)}$ and $\boldsymbol{m(t+\Delta t)}$ is the expected number of failures used to calculate the **reliability** $R(t)$ by means of equation (8), where Δt is set to one year time period.

It can be seen that the expected testing time and expected editing time tend to overestimate the actual testing time and actual editing time. At any rate, both the expected and actual time tend to go in the same direction. The α calculated in each iteration is used to estimate the actual α**,** which turned out to be 1.081. Meanwhile, $\lambda(t)$ gives a projection of how future failure intensity will behave. As the number of faults decreases in each iteration, the reliability increases. Note that the final value of reliability in each sequence is not equal to one another. This is because the sequence of test cases affects the number of test iterations, the number of uncovered faults, and failures in each iteration, all of which affect the value of reliability.

Substituting the number of initial faults and lines of code into Eq. 11 and 12 yields $R_1 = 0.84361$ and $R_2 = 0.75827$.

In the first sequence, $\lambda(t)$ of the first iteration was equal to $\lambda(t)$ avg. The summation of actual cost in first iteration and expected cost of second iteration was greater than threshold cost which was set as \$600. The value of reliability was greater than R_2, so the first sequence could be stopped in this iteration.

In the second sequence, the test continued until the third iteration in which no failure occurred and $\lambda(t)$ was less than $\lambda(t)$ avg. The expected testing time, editing time, and expected cost could not be calculated due to division by 0. So the summation of actual cost of the first iteration to third iteration did not exceed \$600. Since the reliability was greater than R_1, the second sequence could be stopped at its third iteration.

In the third sequence, the test continued in the second iteration where $\lambda(t)$ was less than $\lambda(t)$ avg , the expected cost was less than threshold cost and the reliability value was greater than R_1, the sequence stopped in the second iteration.

Table 7: Experimental results of JSyntaxPane

Test seq.	Rev.	Expected testing time (min)	Expected editing time (min)	Expected cost ($)	Actual testing time (min)	Actual editing time (min)	Actual cost ($)	#rem. faults	#faults	#fails	α	r	FDR	Failure intensity	$\lambda(t)$	$\lambda(t)$ avg	m(t)	$m(t+\Delta t)$	Reliability
1	1	N/A	N/A	N/A	23.94	465.82	387.29	40	20	9	2.22	0.0094	0.041	0.376	0.228	0.228	7.08	18.00	0.64
	2	140.45	465.82	461.73	44.14	174.23	167.34	20	12	4	2.46	0.0045	0.055	0.091	0.055	0.086	3.16	8.13	0.78
	3	115.47	116.15	166.53	23.91	79.84	79.03	8	6	1	2.71	0.0052	0.058	0.042	0.03	0.059	0.85	2.95	0.87
	4	3.15	26.61	23.27	21.23	0.00	13.56	2	0	0	2.71	0	0	0	0	0.048	0.00	0.00	1.00
	5	N/A	N/A	N/A															
2	1	N/A	N/A	N/A	39.76	370.02	320.90	40	26	11	2.36	0.0069	0.063	0.277	0.144	0.144	8.09	16.92	0.70
	2	138.88	199.24	247.84	27.44	79.84	81.29	14	6	2	2.46	0.0052	0.056	0.073	0.051	0.087	1.69	5.69	0.80
	3	111.33	106.45	156.14	25.95	45.34	52.78	8	0	0	2.46	0	0	0	0	0.063	0.00	0.00	1.00
	4	N/A	N/A	N/A															
3	1	N/A	N/A	N/A	28.34	226.48	198.97	40	27	9	3.00	0.0079	0.106	0.318	0.162	0.162	6.54	13.33	0.75
	2	98.00	109.05	149.69	20.91	90.10	85.31	13	7	5	2.43	0.0184	0.063	0.239	0.094	0.121	3.25	5.35	0.90
	3	53.76	77.23	96.02	21.43	0.00	13.69	6	0	0	2.43	0	0	0	0	0.085	0.00	0.00	1.00
	4	N/A	N/A	N/A															

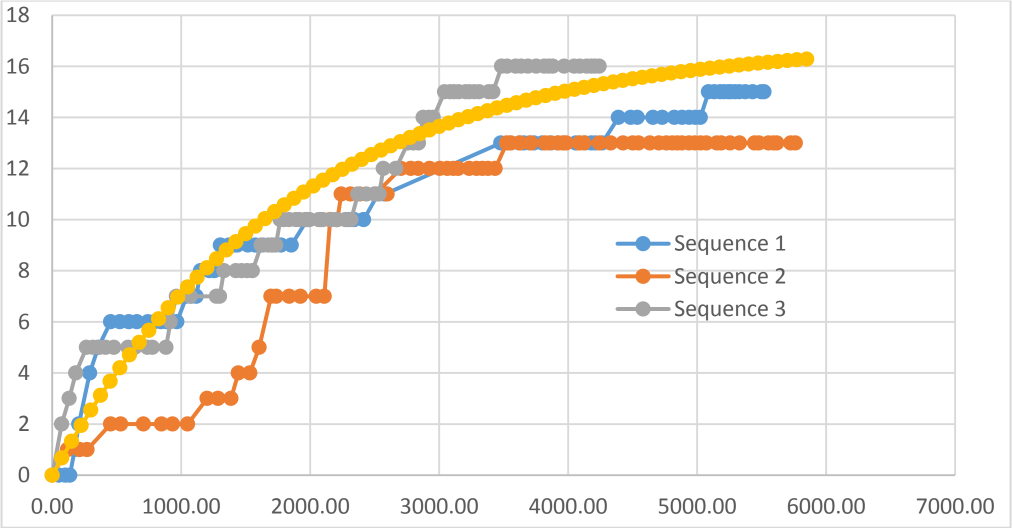


Figure 10 Graph of number of failures in each sequence and m(t) of Jsyntaxpane

Figure 10 plots the number of failures found for each test sequence and the expected number of failures predicted by ***m(t)***. It can be seen that the order of the test cases affected the rate at which failures were found. ***m(t)*** gives the theoretical projection of the number of failures found. The results from Sequence1 and Sequence 3 are fairly close to ***m(t)***, whereas Sequence 2 is not as closed. It shows that m(t) performs quite well for 2 out of 3 randomized sequences.

Table 8 shows the results of JExifViewer. The column name uses the same convention as Table 7. The results followed the same trends as JSyntaxPane. In this case, α turned out to be 1.78 and $R_1 = 0.8382$ and $R_2 = 0.7536$. The input test sequence prioritization was performed according to test coverage technique in Table 4.

In the first sequence (prioritize in order of coverage of instructions), $\lambda(t)$ of the first iteration was equal to $\lambda(t)$ avg and the summation of actual cost in first iteration and the expected cost of second iteration did not exceed the threshold cost of $600. Since the value of reliability was greater than R_1, the first sequence could be stopped in this iteration.

In the second sequence (prioritize in order of coverage of branches), the test continued until the second iteration in which no failure occurred and $\lambda(t)$ was less than $\lambda(t)$ avg. The expected testing time, editing time, and expected cost could not be calculated due to division by 0. So the summation of actual cost of the first iteration did not exceed $600. The value of reliability was greater than R_1. So the second sequence could be stopped at its third iteration.

In the third sequence (prioritize in order of coverage of functions), the test continued on the second iteration where $\lambda(t)$ was less than $\lambda(t)$ avg. The expected cost was less than the threshold cost and reliability was greater than R_1. So it could be stopped in the second iteration.

Compared with randomization in JSyntaxPane, test case prioritization in JExifViewer helped lower the cost of testing and editing. Because similar functions tended to have a nearer coverage, they were more likely to be edited uninterruptedly. If editing other functions affected previous test case, it wouldn’t be found until the next iteration.

Table 8: Experimental results of JExifViewer

Test Sq	Rev	Expected testing time	Expected editing time	Expected cost	Actual testing time(min)	Actual editing time(min)	Actual cost($)	#remaining faults(LOC)	#faults(LOC)	#failures	Alpha	r	r avg	FDR(LOC)	Failure intensity	m(t)	m(t+Dt)	Lamda	Lamda average	Reliability
1	1	N/A	N/A	N/A	9.42	100.30	86.12	16	8	7	1.14	0.0465	0.046	0.073	0.743	5.51	14.0	0.4	0.45	0.69
	2	68.091	100.30	123.60	4.83	58.42	49.74	8	5	2	1.44	0.0517	0.039	0.079	0.413	1.68	5.54	0.2	0.28	0.80
	3	36.699	35.05	51.44	4.12	0.00	2.63	3	0	0	1.44	0	0.030	0	0	0.00	0.00	0	0.21	1
	4	#DIV/0!	#DIV/0!	#DIV/0																
2	1	N/A	N/A	N/A	4.666	86.07	71.71	16	13	6	2.17	0.0804	0.080	0.143	1.285	4.10	7.38	0.5	0.57	0.85
	2	18.279	19.86	27.539	4.583	0	2.92	3	0	0	2.17	0	0.040	0	0	0	0	0	0.28	1
	3	#DIV/0!	#DIV/0!	#DIV/0!																
3	1	N/A	N/A	N/A	5.116	71.29	60.20	16	10	8	1.25	0.0977	0.097	0.131	1.563	5.94	12.8	0.8	0.83	0.74
	2	33.330	42.77	55.452	3.916	23.03	20.89	6	1	1	1.22	0.0426	0.062	0.037	0.255	0.90	4.91	0.2	0.50	0.78
	3	58.789	115.15	129.51	4.05	0	2.587	5	0	0	1.22	0	0.043	0	0	0	0	0	0.34	1
	4	#DIV/0!	#DIV/0!	#DIV/0!																

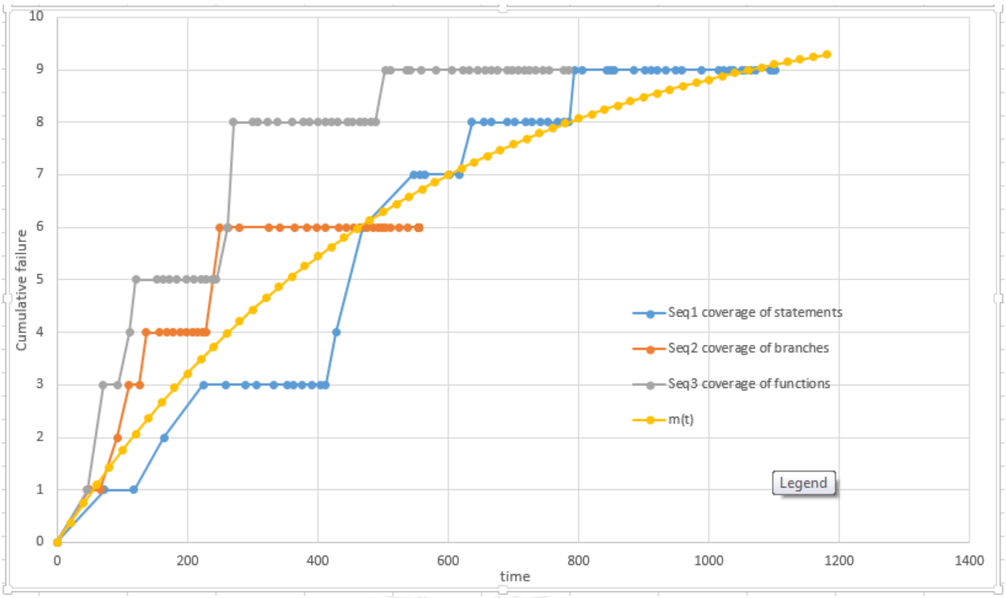


Figure 11 Graph of number of failure in each sequence and m(t) of JExifviewer

Figure 11 shows the trend of failure discovery of JExifviewer using each prioritization technique. All techniques behave in the same trend but keep difference failure detection rate (***r***). From the graph, the expected software failure at time ***t*** or ***m(t)*** was able to predict the number of failures found rather well, especially from sequence 1 which was very close to the result. Compared with JSyntaxpane which randomized the test sequences, the use of test case prioritization in JExifviewer resulted in less testing time and editing time by almost five folds. Between the 3 prioritization arrangements, prioritization by function achieve the best result of both uncovered failure and testing and editing time.

Table 9 Comparative reliability with Weibull and without Weibull distribution

Test sequence	**Revision**	**Reliability**			
		JSyntaxPane		JExifViewer	
		w/o Weibull	w/ Weibull	w/o Weibull	w/ Weibull
1	1	1.81E-05	0.644206	0.000205	0.694755
	2	0.00697	0.783253	0.021073	0.807712
	3	0.122536	0.872341	1	1
	4	1	1		
2	1	0.000146	0.701349	0.037745	0.852629
	2	0.018292	0.802263	1	1
	3	1	1		
3	1	0.001126	0.749327	0.001058	0.743925
	2	0.122034	0.895928	0.018236	0.788964
	3	1	1	1	1

Table 9 shows reliability values of both applications when computed with and without Weibull distribution (Equation 10 and Equation 4, respectively). The former is proposed technique, whereas the latter is a comparative existing technique [14]. It can be seen from the comparison that the reliability with Weibull distribution is applicably suitable for the stopping criteria. The initial value of reliability in the first iteration w/Weibull is higher than that of the w/o Weibull.

Figure 12 summarizes graphical comparisons of both tests. From the graph, the values computed by Weibull distribution increase in a more stable rate than those without Weibull whose increment goes up drastically in the final distribution.

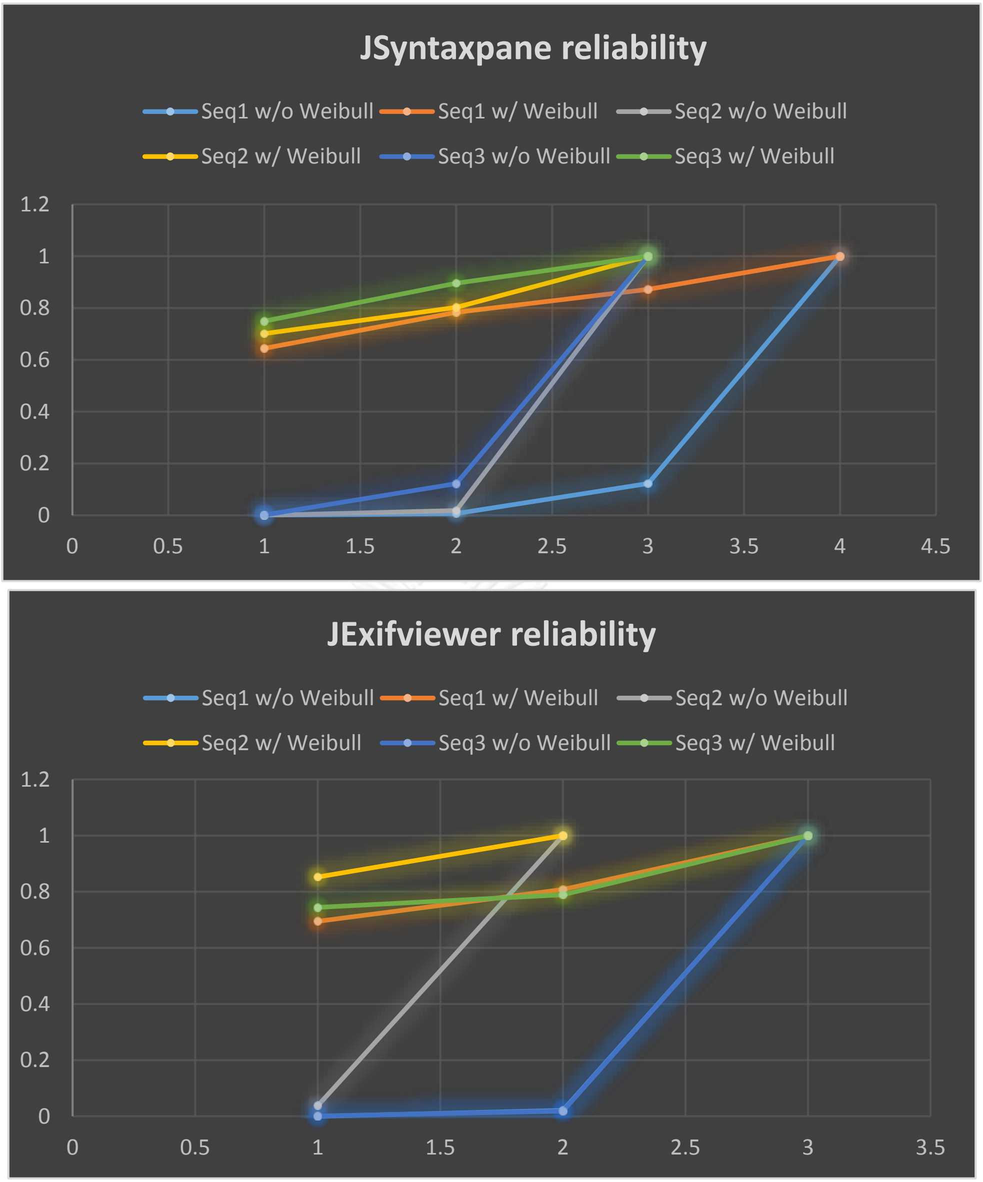


Figure 12 Graph of reliability comparison for JSyntaxpne and JExifviewer

Chapter 5 Discussion and Conclusion

5.1. Discussion

This research proposed the stopping criteria for GUI application regression testing. Software reliability model was used to determine the appropriate time to stop. An equation to estimate the cost of testing and editing was proposed by using SRM to calculate the expected testing and editing time. Weibull distribution was integrated into reliability function for flexibility purpose. Stopping criteria involved 3 factors computed from test statistics, namely, failure intensity, cost of editing and testing, and reliability. The proposed methodology was successful in controlling test process to stop earlier than it normally should buy virtue of the 3 combined factors of stopping criteria. The rationale was straightforward in that as failure intensity decreased owing to spontaneous bug fixes, reliability increased. On the contrary, if erroneous situation dragged on, test cost escalated. Upon reaching the proposed costing limit, test process terminated. In either case, the approach could practically be tailored to work in production environment. One validity measure was resulted from threshold cost figure, which was derived from non- authoritative source of salary. Nonetheless, the issue was relatively minor.

Test cases were organized into test sequences using randomization and prioritization based on 3 coverage measures, i.e., statements, branches, and functions. The 3 prioritization techniques chosen in this research are the most suitable in terms of fault detection without adding too much complexity. However, they may not be the best technique to uncover all the faults as the results depended largely on the input test sequence. This fact was apparent in the resulting experiment.

The proposed methodology was tested using 2 GUI applications. Constants and thresholds used in the equations were calculated in a preliminary test using production software. The results show that the stopping criteria are suitable for determining appropriate time to stop regression testing and can help lower both time and cost of testing and editing which is beneficial from business perspective. One many contend that GUI testing in many cases is dependent on the application set up, system requirements, domain of applicability, etc. Thus, test results could vary inconsistently which might lead to inconclusive outcome.

5.2. Conclusion

This research proposes a practical stopping criteria for GUI regression testing. The ultimate objective is to end the test process faster than running the test normally, thereby saving considerable time and costs, yet still preserving test outcome reliability. The approach exploits 3 factors of test process, while organizes the input test cases in two different scenarios. Test coverage is set up to measure the impact of input sequence. And findings turn out satisfactorily.

Future works include optimizing regression test techniques to achieve minimal cost and finding more efficient test sequence generation.

REFERENCES

1. *Software Engineer I Salary*. Salary.com.
2. Biswas, S., et al., *Regression test selection techniques: A survey.* Informatica: An International Journal of Computing and Informatics, 2011. **35**: p. 289–321.
3. Vokolos, F.I. and P.G. Frankl, *Pythia: A regression test selection tool based on textual differencing*, in *Reliability, Quality and Safety of Software-Intensive Systems*, D. Gritzalis, Editor. 1997, Springer US. p. 3-21.
4. Vokolos, F.I. and P.G. Frankl. *Empirical evaluation of the textual differencing regression testing technique*. in*, International Conference on Software Maintenance, 1998. Proceedings*. 1998.
5. Rothermel, G. and M.J. Harrold. *Selecting regression tests for object-oriented software*. in *Software Maintenance, 1994. Proceedings., International Conference on*. 1994.
6. Rothermel, G. and M.J. Harrold, *Selecting tests and identifying test coverage requirements for modified software*, in *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. 1994, ACM: Seattle, Washington, USA. p. 169-184.
7. White, L.J. *Regression testing of GUI event interactions*. in *Software Maintenance 1996, Proceedings., International Conference on*. 1996.
8. Hui, Z., et al. *GUI regression testing based on function-diagram*. in *2010 IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS)*. 2010.
9. Memon, A.M. and M.L. Soffa. *Regression Testing of GUIs*. 2003. ACM.
10. Falah, B., R. Nouasse, and Y. Laghlid, *GUI Regression Test Selection Based on Event Interaction Graph Strategy*. 2013, IJCSET.
11. Grigorjev, F., N. Lascano, and J.L. Staude. *A fault seeding experience*. in *Simposio Argentino de Ingenieria de Software (ASSE 2003)*. 2003. Citeseer.

12. Harrold, M.J., A.J. Offutt, and K. Tewary, *An approach to fault modeling and fault seeding using the program dependence graph.* Journal of Systems and Software, 1997. **36**(3): p. 273-295.

13. Xie, M., *Software Reliability Modelling*. 1991: World Scientific. 232.

14. Lin, C.-T. and C.-Y. Huang. *Software Release Time Management: How to Use Reliability Growth Models to Make Better Decisions*. in *2006 IEEE International Conference on Management of Innovation and Technology*. 2006.

15. Ahmad, N., et al., *The exponentiated Weibull software reliability growth model with various testing-efforts and optimal release policy.* International Journal of Quality & Reliability Management, 2008. **25**: p. 211-235.

16. Beizer, B. *bug taxonomy - Otto Vinter*.

17. Elbaum, S., A.G. Malishevsky, and G. Rothermel, *Test case prioritization: a family of empirical studies.* Software Engineering, IEEE Transactions on, 2002. **28**(2): p. 159-182.

18. Elbaum, S., A. Malishevsky, and G. Rothermel, *Incorporating varying test costs and fault severities into test case prioritization*, in *Proceedings of the 23rd International Conference on Software Engineering*. 2001, IEEE Computer Society: Toronto, Ontario, Canada. p. 329-338.

19. *jexifviewer Java program for displaying and comparing Exif informations stored in JPEG files created by digital cameras. JExifViewer is an Open Source project released under the GPL.*

20. *JaCoCo Java Code Coverage Library*

APPENDIX

As mentioned before, test cases for both JSyntaxpane and JExifviewer were generated according to their GUI functionalities as well as existing bug reports from other users. The detail of all test cases used are given in Table 10 and Table 11

Table 10 Test cases for JSyntaxpane

1	**Test case name**	**Test CUT function**	**Expected result**
	Test Step/Substep	Open file under test	File is open correctly
		Select some Text	Selected text is highlighted
		Press CUT button	Selected text disappears
		Paste in other text editor	Selected text is pasted correctly
2	**Test case name**	Test COPY function	**Expected result**
	Test Step/Substep	Open file under test	File is open correctly
		Select some Text	Selected text is highlighted
		Press COPY button	
		Paste in other text editor	Selected text is pasted correctly

3	**Test case name**	Test PASTE function	**Expected result**
	Test Step/Substep	Copy some text from other editor	
		Press PASTE button	Selected text is pasted correctly
4	**Test case name**	Test SELECT ALL function	**Expected result**
	Test Step/Substep	Open file under test	File is open correctly
		Press SELECT ALL button	All texts in syntaxpane are highlighted
		Press Backspace	All texts disappear
5	**Test case name**	Test UNDO REDO function	**Expected result**
	Test Step/Substep	Do something	The action is done correctly

		Press UNDO button	The previous action is undone
		Press REDO button	The previous action is redone
6	**Test case name**	Test FIND/REPLACE function #1	**Expected result**
	Test Step/Substep	Open file under test	File is open correctly
		Press DISPLAY FIND AND REPLACE DIALOG button	FIND AND REPLACE dialog appears
		Type some text in FIND text field	The typed text appears in text field
		Press NEXT button	A matching text is hilighted in jsyntaxpane or a warning appears in case no matching text exists

		If a match is found, proceed to the next step	
		Press NEXT button	The next matching text is hilighted in jsyntaxpane
		Press PREVIOUS button	The matching text from step 4 is hilighted in jsyntaxpane
7	**Test case name**	Test FIND/REPLACE function #2	**Expected result**
	Test Step/Substep	Open file under test	File is open correctly
		Press DISPLAY FIND AND REPLACE DIALOG button	FIND AND REPLACE dialog appears
		Type some text in FIND text field	The typed text appears in text field
		Select IGNORE CASE check box	IGNORE CASE check box becomes selected
		Press NEXT button	A matching text is hilighted in jsyntaxpane

			or a warning appears in case no matching text exists
		Press NEXT button	A next matching text is hilighted in jsyntaxpane or a warning appears in case no matching text exists
		Press NEXT button	A next matching text is hilighted in jsyntaxpane or a warning appears in case no matching text exists
8	**Test case name**	Test FIND/REPLACE function #3	**Expected result**
	Test Step/Substep	Open file under test	File is open correctly

		Press DISPLAY FIND AND REPLACE DIALOG button	FIND AND REPLACE dialog appears
		Type some text in FIND text field	The typed text appears in text field
		Select REGULAR EXPRESSION check box	REGULAR EXPRESSION check box becomes selected
		Press NEXT button	A matching text is hilighted in jsyntaxpane or a warning appears in case no matching text exists
		Press NEXT button	The next matching text is hilighted in jsyntaxpane or a warning appears in case no matching text exists

9	**Test case name**	Test FIND/REPLACE function #4	**Expected result**
	Test Step/Substep	Open file under test	File is open correctly
		Press DISPLAY FIND AND REPLACE DIALOG button	FIND AND REPLACE dialog appears
		Type some text in FIND text field	The typed text appears in FIND text field
		Type another text in REPLACE text field	The typed text appears in REPLACE text field
		Press NEXT button	A matching text is hilighted in jsyntaxpane
		Press REPLACE button	The hilighted text is replaced with the text in REPLACE text field and the next matching text is hilighted in jsyntaxpane
		Press REPLACE button	The hilighted text is replaced with

			the text in REPLACE text field and the next matching text is hilighted in jsyntaxpane
10	**Test case name**	Test FIND/REPLACE function #5	**Expected result**
	Test Step/Substep	Open file under test	File is open correctly
		Press DISPLAY FIND AND REPLACE DIALOG button	FIND AND REPLACE dialog appears
		Type some text in FIND text field	The typed text appears in FIND text field
		Type another text in REPLACE text field	The typed text appears in REPLACE text field
		Press REPLACE ALL button	All matching texts are replaced with the text in REPLACE text field

11	**Test case name**	Test FIND NEXT function	**Expected result**
	Test Step/Substep	Open file under test	File is open correctly
		Press DISPLAY FIND AND REPLACE DIALOG button	FIND AND REPLACE dialog appears
		Type some text in FIND text field	The typed text appears in FIND text field
		Press NEXT button	A matching text is hilighted in jsyntaxpane
		Close FIND AND REPLACE dialog	FIND AND REPLACE dialog disappears
		Press REPEAT LAST FIND	The next matching text is hilighted in jsyntaxpane
		Press REPEAT LAST FIND	The next matching text is hilighted in jsyntaxpane
12	**Test case name**	Test GOTO LINE function	**Expected result**

	Test Step/Substep	Open file under test	File is open correctly
		Press GOTO LINE NUMBER button	GOTO LINE dialog appears
		Type some number in to the text field	
		Press GO button	The caret moves to the beginning of the entered line number or the nearest line number if the entered line number does not exist
13	**Test case name**	Test JUMP TO PAIR function (for programming language)	**Expected result**
	Test Step/Substep	Open file under test	File is open correctly
		Choose the corresponding language in the combobox	
		Click on one of the following brackets [({ <	
		Press JUMP TO PAIR button	The caret moves to the

			corresponding]) } >
14	**Test case name**	Test JUMP TO PAIR function (for markup language)	**Expected result**
	Test Step/Substep	Open file under test	File is open correctly
		Choose the corresponding language in the combobox	
		Click on a tag	
		Press JUMP TO PAIR button	The caret moves to the corresponding tag
		Press JUMP TO PAIR button	The caret moves to the corresponding tag
		Press JUMP TO PAIR button	The caret moves to the corresponding tag
15	**Test case name**	Test TOGGLE COMMENTS	**Expected result**

	Test Step/Substep	Open file under test	File is open correctly
		Choose the corresponding language in the combobox	
		Click on a line	
		Press TOGGLE COMMENTS button	The line is commented according to te rule of associated language
		Select the commented line	
		Press TOGGLE COMMENTS button again	The line is uncommented
16	**Test case name**	Test INDENT/UNINDENT	**Expected result**
	Javascript		
	Java		
	Python		
	Test Step/Substep	Open file under test	File is open correctly
		Choose the corresponding language in the combobox	
		Select some lines	
		Click INDENT button	The selected lines are indented from

			the beginning by one tab
		Select some other lines	
		Click UNINDENT button	The spaces at the beginning of each selected lines are decreased by one tab (nothing happen if that line does not begin with space)
17	**Test case name**	Test TOGGLE LINES	**Expected result**
	Javascript		
	Java		
	xhtml,xml,xpath		
	Test Step/Substep	Open file under test	File is open correctly
		Choose the corresponding language in the combobox	
		Press TOGGLE LINES button	Line numbers disappear
		Press TOGGLE LINES button	Line numbers reappear

18	**Test case name**	Test SURROUND WITH TRY CATCH	**Expected result**
	Java		
	Test Step/Substep	Open file under test	File is open correctly
		Choose the corresponding language in the combobox	
		Click on a blank line	
		Press SURROUND WITH TRY CATCH button	A try/catch block appear at that position
		Select some texts that span across multiple lines	
		Press SURROUND WITH TRY CATCH button	All lines that contain the selected texts are surrounded by a try/catch block
19	**Test case name**	Test SURROUND SELECTION WITH WHILE	**Expected result**
	Java		
	Test Step/Substep	Open file under test	File is open correctly
		Choose the corresponding language in the combobox	
		Click on a blank line	

		Press SURROUND SELECTION WITH WHILE button	A while block appear at that position
		Select some texts that span across multiple lines	
		Press SURROUND SELECTION WITH WHILE button	All lines that contain the selected texts are surrounded by a while block
20	**Test case name**	Test SURROUND WITH IF	**Expected result**
	Java		
	Test Step/Substep	Open file under test	File is open correctly
		Choose the corresponding language in the combobox	
		Select some texts that span across multiple lines	
		Press SURROUND WITH IF button	All lines that contain the selected texts are surrounded by an if block

21	**Test case name**	Test OUTPUT EXPRESSION TO SYSTEM.OUT	**Expected result**
	Java		
	Test Step/Substep	Open file under test	File is open correctly
		Choose the corresponding language in the combobox	
		Select some texts that span across multiple lines	
		Press OUTPUT EXPRESSION TO SYSTEM.OUT button	The selected texts are surrounded by System.out.println("The value of SELECTED TEXTS = " + (SELECTED TEXTS));
22	**Test case name**	Test SURROUND LINES WITH BLOCK COMMENTS	**Expected result**
	Java		
	Test Step/Substep	Open file under test	File is open correctly
		Choose the corresponding language in the combobox	
		Select some texts that span across multiple lines	

		Press SURROUND LINES WITH BLOCK COMMENTS button	All lines that contain the selected texts are surrounded by /* and */
23	**Test case name**	Test language combobox	
	Test Step/Substep	Open file under test	File is open correctly
		Choose the corresponding language in the combobox	The UI components are displayed correctly
		Choose some other language in the combobox	The UI components are displayed correctly
		Change back to the corresponding language in the combobox	The UI components are displayed correctly
24	**Test case name**	Test QUICK FIND function	
	Test Step/Substep	Open file under test	File is open correctly

		Press Ctrl+F	QUICK FIND dialog appears
		Type some text in QUICK FIND text field	The typed text appears in text field and the first matching text is hilighted in real time or a warning appears in case no matching text exists
		If a match is found, proceed to the next step	
		Press NEXT button	The next matching text is hilighted in jsyntaxpane
		Press PREVIOUS button	The matching text from step 4 is hilighted in jsyntaxpane
25	**Test case name**	Test FIND/REPLACE function (WRAP AROUND#1)	**Expected result**
	Test Step/Substep	Open file under test	File is open correctly

		Press DISPLAY FIND AND REPLACE DIALOG button	FIND AND REPLACE dialog appears
		Type some text in FIND text field	The typed text appears in text field
		Press NEXT button until the last matching text is reached	The last matching text is hilighted
		Make sure that the WRAP AROUND is not selected then press NEXT button	A warning dialog appears informing that Serch String not found
		Select WRAP AROUND check box	WRAP AROUND check box becomes selected
		Press NEXT button	The first matching text is hilighted
26	**Test case name**	Test	**Expected result**
	Test Step/Substep	Type some characters	The characters appear
		Press ENTER button	The caret moves to a

			new line below

Table 11 Test cases for JExifviewer

Sq	Test step/Substep	Expected result
1	Navigate to a directory	That folder is selected and the image files inside are shown correctly
	Click on a column name	The image files are sorted by that column attribute
2	Hover the mouse over an image file	The image tooltip information is shown according to tooltip setting
	Check the tooltip information	The information is consistent with image properties
3	Left-click on a row in the right panel	An image appears in the bottom-left panel
4	Right-click on a row in the right panel	List pop-up menu appears
	Choose Rename command	Rename dialog appears
	Type a new name in text field and click OK button	The name changes while other information remains the same

5	Right-click on a row in the right panel	List pop-up menu appears
	Choose Copy command	A directory chooser appears
	Choose directory and press OK button	The copied image appears in the chosen directory
6	Right-click on a row in the right panel	List pop-up menu appears
	Choose Move command	A directory chooser appears
	Choose directory and press OK button	The image is moved to the chosen directory
7	Right-click on a row in the right panel	List pop-up menu appears
	Choose Delete command	Delete dialog appears
	Choose Yes button	The image is removed
8	Right-click on a row in the right panel	List pop-up menu appears
	Choose Delete command	Delete dialog appears
	Choose No button	Delete dialog disappears
9	Right-click on a row in the right panel	List pop-up menu appears
	Choose Cancel command	List pop-up menu disappears
10	Select an image	An image appears in the bottom-left panel
	Double-click on the image	Full screen image is shown

	Double-click on the image again	Full screen image disappears
11	Right-click on a directory	Tree pop-up menu appears
	Choose Add shortcut command	A shortcut with the same directory name appears at the root of the directory tree
12	Right-click on a shortcut	Tree pop-up menu appears
	Choose Remove shortcut command	The shortcut disappears
13	Right-click on a directory in the top-left panel	Tree pop-up menu appears
	Choose Cancel command	Tree pop-up menu disappears
14	Right-click on the image	Image pop-up menu appears
	Choose +90 command	The image is rotated 90 degrees clockwise
15	Right-click on the image	Image pop-up menu appears
	Choose -90 command	The image is rotated 90 degrees counterclockwise
16	Right-click on the image	Image pop-up menu appears

	Choose 180 command	The image is rotated 180 degrees
17	Right-click on the image	Image pop-up menu appears
	Choose Flip horizontal command	The image is flipped horizontally
18	Middle-click on the image	The image is flipped horizontally
	Right-click on the image	Image pop-up menu appears
	Choose Original command	The image is reverted back to original
19	Right-click on the image	Image pop-up menu appears
	Choose Cancel command	Image pop-up menu disappears

VITA

Chalita Somsorn was born in Bangkok and received the B.S. in Computer Science from Chulalongkorn University in 2013. Currently, studying for a M.S. in Computer Science, Chulalongkorn University. The areas of interest are software engineering,GUI application ,regression testing and software reliability models.