Computer Engineering Department
Faculty of Engineering
Deanery of Higher Studies
The Islamic University – Gaza
Palestine

Design of a Selective Continuous Test Runner

Mohammed Riyad El Khoudary

Supervisor

Dr. Wesam Ashour

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Engineering

1434H (2013)

Dedication

To my beloved Parents

To my dear wife

To my lovely

Salma, Tasneem, and Lana

Acknowledgments

Thanks to Allah for giving me the power and well to accomplish this research. I would not have been able to finish it without Allah's grace.

I am very thankful to my father Prof. Riyad El Khoudary for being the ideal whom I have always wanted to be like, and for his endless support. Also I would like to give my thanks to my beloved mother, Bohaisa El Ghalayini, for her support and prayers. I give them the credit for making me the person I am right now.

Words of thanks to my dear wife, Amal Murad, for her prayers, patience, motivation, and continuous support. I wouldn't have done this research without her standing beside me.

I also extend my thanks to all my brothers and sisters for their motivation. Especially to my sister Dr. Samar R. El Khoudary for her valuable help in conducting and analyzing the pilot study.

I am very grateful to my supervisor, Dr. Wesam Ashour, for his vast and enormous support, valuable comments and notes. Without his guidance this work wouldn't have seen the light with this shape.

Special thanks to Mr. Kent Beck, Dr. Fausto Spoto, and Dr. David Saff for their valuable guide and comments.

Finally thanks to my dear friend Mr. Ahmed Abu Dayya for his help in producing the printed version of the thesis.

Table of Contents

List of Abbreviations

SUT	System Under Test
CUT	Class Under Test
TDD	Test Driven Development
RTS	Regression Test Selection
TCP	Test Case Prioritization
CT	Continuous Testing
CIA	C Information Abstractor
IDE	Integrated Development Environment
API	Application Programming Interfaces

List of Figures

List of Tables

تصميم مشغّل فحص انتقائي مستمر

محمد رياض الخضري

الملخص

هذا البحث يعرض و يقارن الأساليب المتبعة لتقليص التأخير الناجم عن استخدام أساليب الاختبار الحديثة التي تتم خلال فترة البرمجة. الدراسات السابقة قدمت حلول لتقليص هذا الوقت من خلال استخدام مشغلات فحص مستمرة تنتقي الفحوصات بشكل عشوائي أو شبه عشوائي. نقدم في هذا البحث مبدأ مخطط التأثير و الذي يدرس تأثير الأسطر البرمجية بعضها على بعض و يربطها بالدوال الخاصة بالاختبار بشكل يتيح امكانية انتقاء الفحوصات ذات العلاقة بالسياق البرمجي. كما نقدم أيضاً تصميم مشغّل فحوصات مستمر يقوم على مبدأ مخطط التأثير و الذي يعمل بشكل مستمر أثناء جريان عمليات البرمجة و التعديل. النتائج أثبتت أن: تطبيق مبدأ مخطط التأثير على انتقاء الفحوصات أدى المطلوب بكفاءة و دقة. أيضاً أن هذا المبدأ عمل بكفاءة مع مشاريع برمجية ضخمة. و أخيراً توظيف هذا المبدأ في مشغّل الاختبارات المقدّم أظهر كفاءة عالية في اكتشاف الخلل في الفحوصات عند التشغيل المستمر أثناء عملية البرمجة.

Design of a Selective Continuous Test Runner

Mohammed R. El Khoudary

Abstract

This study presents the design of a selective continuous test runner, which has not been done before. Previous studies present only a continuous test runner with random or semi-random test cases selection techniques. Here we present the concept of influence graph which is constructed directly by using source code and then use this influence graph to detect any influence on any test case and run the tests on the background. For that purpose three algorithms were designed; one for building the influence graph for the first time, another for enhancing the influence graph according to code modifications, and the third for marking relevant test cases for retesting. We created an Integrated Development Environment (IDE) for test purpose. The mentioned algorithms were implemented on this IDE. Experimental results show: (1) Influence graphs helped efficiently in detecting the changed test cases, (2) The proposed technique worked well with large projects, and (3) The selective continuous test runner helped in detecting logic deviations in a more effective and fast way than the regular test running schemas.

Keywords

Regression Testing, Continuous Test Running, Regression Test Selection.

Chapter 1: Introduction

Every successful product needs validation and verification to meet specific quality measures. Software products aren't an exception. They are verified and validated by using standard software testing.

Researchers worked on enhancing and upgrading each level of software testing. Some of them gave the testing done by developers a special attention, such as Test Driven Development (TDD) [1].

As these techniques prove to have advantages; they also brought disadvantages in debate. TDD gave good results experimentally. However, industrially it added extra latencies on the development and maintenance times.

The debate among researchers is about how to reduce these latencies accompanied with using any testing technique including TDD. Some of them considered continuous testing, while others contributed with methodologies and techniques to reduce the number of test cases being executed.

This chapter introduces software testing, unit testing, system under test (SUT), regression errors, and test driven development (TDD). It describes also the drawbacks and latencies accompanied by TDD, and finally states the research motivations.

1.1 Software Testing

Testing is the process of executing a program with the intent of finding errors, it is an investigation conducted to provide stakeholders with information about the quality of the product or service under test [2, 3].

The person who performs software testing is called a test engineer. He is in charge of one or more test activities, such as designing test inputs, preparing the expected test case values, running test scripts, analyzing results, and reporting these results to the concerned developers and managers [4].

Software testing activities can be held in two ways. The most commonly used one is to test after certain parts of the application are finished, other techniques combine testing activities with development activities. Test design and construction are the most time-consuming tasks in software testing. So test activities can and should be carried out along with development [4].

Tests are grouped by where they are added in the software development process. Here follows the levels of testing accompanying each software development activity: [5]

- Acceptance Testing – assess software with respect to requirements.
- System Testing – assess software with respect to architectural design.
- Integration Testing – assess software with respect to subsystem design.
- Module Testing – assess software with respect to detailed design.
- Unit Testing – assess software with respect to implementation.

Figure 1.1 shows software development activities and testing levels–the "V Model" [4].

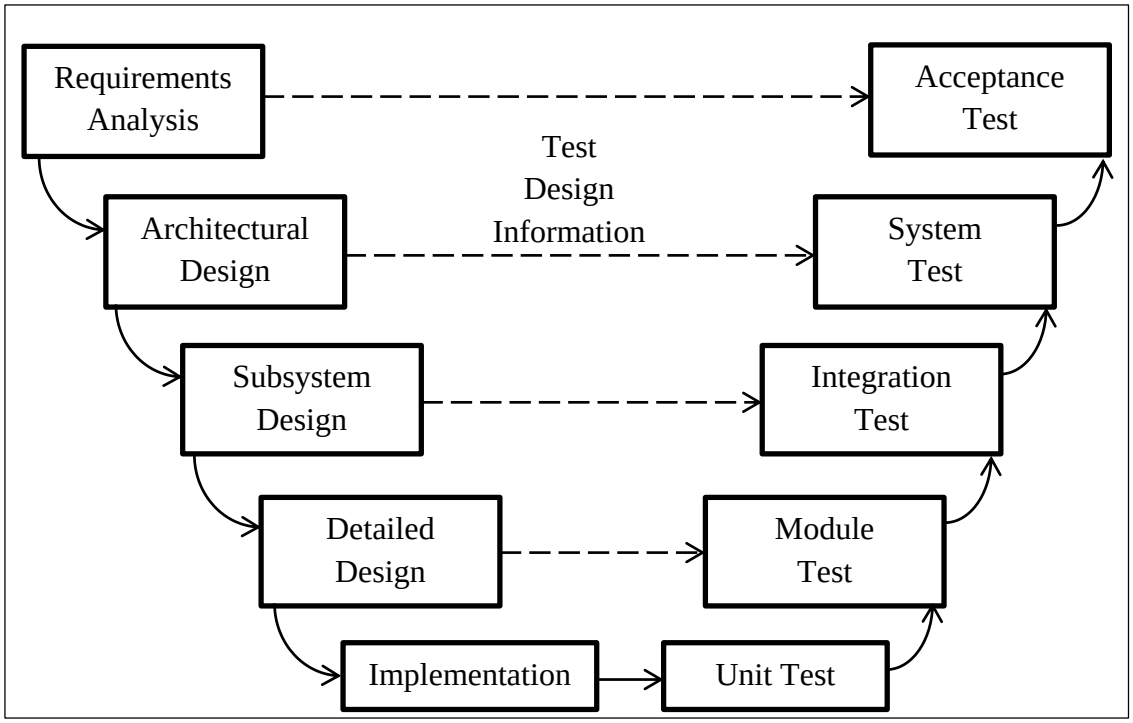

Figure 1.1 The V-Model

1.2 Unit Testing

A unit test is a piece of code wrapped inside a method written for test purpose. This method invokes pieces of code and checks the correctness of this code using some predefined assumptions. If the assumptions turn out to be right then the unit test succeeds otherwise it fails [5].

Unit testing is performed against a system under test (SUT). Some people like to call it CUT (class under test or code under test) for more specification [5].

A unit test should have the following properties:

- It should be automated and repeatable to the ease of its usage.
- It should be easy to implement, so developers will not stop implementing new units with time.
- Once it's written, it should remain for future use.
- Anyone should be able to run it.
- It should run at the push of a button.
- It should run quickly, the quicker the more commitment from developers in running tests frequently.

Figure 1.2 shows that unit tests are developed apart from production code. This has several advantages: [6]

- It helps in testing single software objects in isolation.
- It prevents cluttering up the production code with built-in unit tests.
- It reduces the final application.

A unit test tests the behavior of a specific code unit. Running a test verifies the behavior of that unit. The smaller the code units are, the more detailed and comprehensive the test suite is.

Writing unit tests individually isn't enough; the developer needs to create a comprehensive test suite to get all the benefits of unit testing. For managing test suites the term Unit Test Framework is presented. Unit test frameworks are simply software tools that help in writing, maintaining, and running unit tests [6].

1.3 Regression Errors

A regression is a feature that is used to work and now doesn't. Developers fear doing changes in code with no unit tests because this might put the application in an undefined state of stability [5].

Unit tests should run quickly. The slower unit tests are the lesser developers will run them (daily, or even weekly or monthly in some places). The problem is that, when you change code, you want to get feedback as early as possible to see if you made any mistake. The more time between running the tests, the more changes you make to the system. And the (many) more places to search for bugs when your code misbehaves. That is exactly where regression time is increased. Extreme Programming methodology

emphasizes the importance of unit test suites that are run and verified very frequently. This ensures that code can be changed rapidly without much regression errors [6, 8].

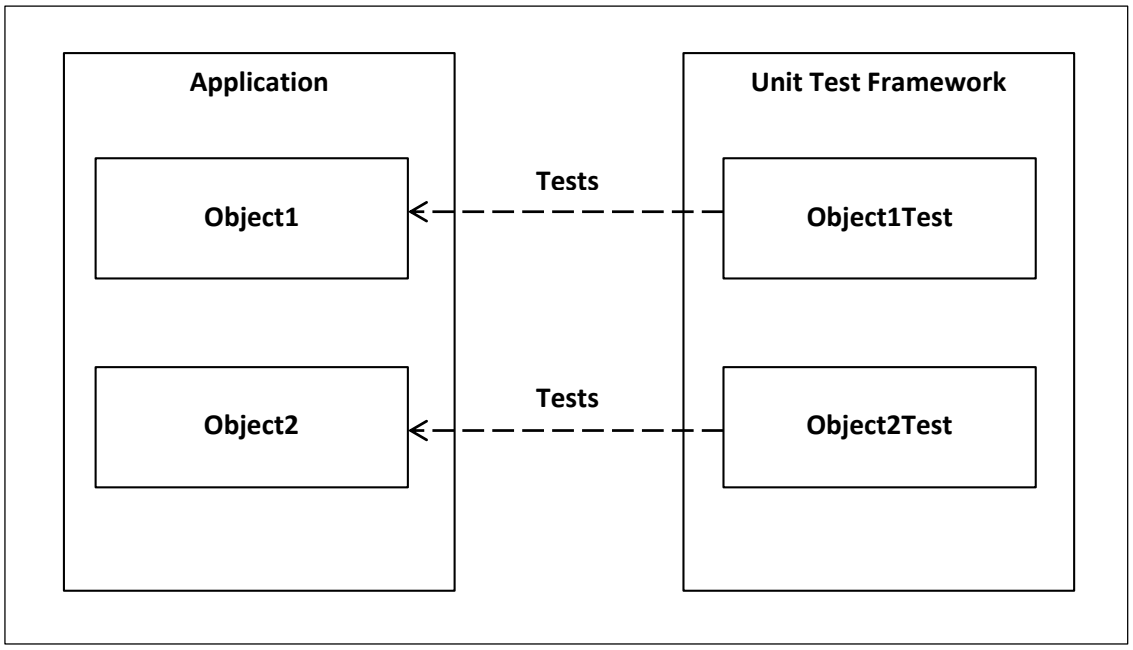

Figure 1.2 Production application and unit test framework [6]

1.4 Test Driven Development (TDD)

Test-driven development (TDD) is a software testing/development methodology that relies on the repetition of a very short and fast development cycle. It states that software development starts directly from software requirements [8].

The cycle is described as the TDD Mantra. It is defined in three steps:

1. **Red**: Write a little test that doesn't work, and perhaps doesn't even compile at first.
2. **Green**: Make the test work quickly, committing whatever development mistakes necessary in the process.
3. **Refactor**: Eliminate all of the duplication created in merely getting the test to work and make the code follow the firm standards [1].

The benefits of TDD are:

1. The study in [9] showed that programmers who write more tests are more productive.
2. Using TDD from the beginning reduces invoking the debugger, since code is tested implicitly during development [10].

3. Test coverage is significantly improved [11].
4. TDD can lead to a more modularized, flexible, and extensible code [8].
5. Although more code will be written, the overall amount of code for TDD will be shorter [12].

In the other hand there are shortcomings in TDD:

1. TDD is difficult to use in situations where full functional tests are required to determine success or failure [8].
2. TDD should be supported by the entire firm/organization to succeed [13].
3. The developer is the tester. So mistakes in production code can also exist in unit tests.
4. Unit tests are added to the maintenance overhead. Not updating unit tests increases the regression time [14].
5. To create a comprehensive test suite the methodology should start from the beginning of the development time. Starting TDD on an already started project isn't that easy.
6. The bigger the test suite becomes; the lesser the developers will run it. So regression errors might exist more frequently.

1.5 Research Motivation

The majority of work done in unit testing and TDD was for measuring applicability and profitability [16, 17, 18] or introducing the concepts to other platforms or environments like web, mobile, web services, …etc. [19, 20].

Other studies discussed reducing wasted time in regression tests by reducing developer ignorance time by introducing the concept of Continuous Test Runners [22]. Others [23] introduced a solution to the wasted time in running the complete test suite by selecting a smaller subset of test cases.

1.5.1 Research Problems

The following points describe the research problems:

- Most of the empirical studies stated that industry avoids using TDD because of the latencies added to the development time.
- As the test suite gets bigger; developers execute it less often, and that increases the ignorance time when a bug occurs.

- As the ignorance time increases regression time increases as well, and so regression errors become harder to find and trace.
- Test running time and regression time are essential components in the latencies caused by TDD.
- Also with time developers add test cases less often and so the coverage of the test cases over a program becomes incomplete.
- Researchers presented techniques for reducing regression time by continuous test running [22] and by regression test selection like [23] and others.
- However, these approaches either didn't maintain running a regression test selection or didn't design their technique for online use.

In the following we shall state the research objectives briefly and describe them in research contributions.

1.5.2 Research Objectives

- Present three novel algorithms for building and maintaining an online regression test selection (RTS) technique.
- Present the design of a continuous test runner based on the presented RTS technique.
- Implement the three algorithms and test them in a real software project to test the accuracy and speed of the proposed RTS technique.
- Conduct a pilot study about the impact of using a selective continuous test runner on the regression time.

1.5.3 Research Contributions

Our research will contribute with an RTS technique that can be used efficiently online during development. This technique will be able to synchronize with the current source code progressively in the background without the developer noticing at all. Also this technique will not be platform or programming language dependent, the algorithms can be implemented for any programming language.

Also we will contribute with the design of an efficient continues test runner. That's eligible of running our RTS or any other RTS. Additionally it will have a wide set of tunable parameters for best performance and outcome. Finally this continuous test runner will have a phase of test case prioritization as well.

1.6 Thesis Structure

The rest of the report is organized as follows:

- Chapter 2 reviews related work, states the close approaches, the strength points, and the drawbacks of each approach.
- Chapter 3 presents the novel concept of the influence graph and the three algorithms for maintaining it, showing how would such a graph handle code changes and how it would select the appropriate test cases to be executed.
- Chapter 4 presents the continuous test runner design and structure and the way it interacts with the proposed selection technique.
- Chapter 5 presents the experimental IDE that was designed and implemented for the purpose of testing the proposed techniques.
- Chapter 6 presents the experiments and the experimental results.
- Finally, we draw the conclusion and state the future work.

Chapter 2: Background and Related Work

Regression Testing is a highly challenging problem because of the large size of test cases in large systems. And because of its impact on the development of a software product. So, the concept of Regression Test Selection (RTS), that helps in reducing the time needed to run a test suite by selecting only relevant tests for retesting, was introduced by several papers starting from the earliest approach TEST TUBE [24], ECHELON [25], as well as [25, 26, 27, 28, 29]. A new study [23] introduced an approach with new considerations regarding performance. Also many papers such as [30, 31, 32] made studies and discussed the effect of (RTS) on test suite execution speed.

Another important factor that helps in reducing retesting wasted time is the concept of continuous testing (CT); which means running tests in the background during development time. CT itself was introduced in [32].

2.1 Test Driven Development (TDD)

2.1.1 Definition and Example

TDD drives the development process by unit tests. As mentioned in Chapter 1 you start by writing a unit test that ultimately fails, then write a code that barely makes the test passes, and finally refactor that code to make it follow the committed standards.

TDD helps in producing "Clean code that works", in Ron Jeffries' pithy phrase. Such a code is beneficial for a number of reasons like: being predictive, improving the lives of the users of the software, letting teammates count on each other.

Many factors drive us away from writing clean code that works. The style of TDD pushes us back to writing the clean code that works through:

- Writing new code only if an automated test has failed.
- Eliminate duplication [7].
- Also TDD imposes a specific behavior on its users.
- Tests should run to provide feedback between decisions.
- Tests for a specific module should be written by the same developer who develops this module.
- Development environment must provide rapid response to small changes.

To make testing easy; components should be highly cohesive, and loosely coupled. Many studies stated that TDD:

- Helps in making software highly cohesive and loosely coupled [33, 34].
- Provides improved test coverage [33, 35].
- Enables implementation scope to be more explicit [33, 35].
- May lead to enhanced job specification and confidence.

As mentioned in Chapter 1 TDD is based on the TDD mantra or the Red / Green / Refactor development cycle. Such a way in development helps in dramatically reducing the defect density of code and makes the subject of work crystal clear to all involved developers [7].

Let's take an example TDD development cycle. Assume we have to create a calculator that adds numbers. Beginning with red phase we should write a test that fails.

```
public void testAdd() {
      Calculator myCalculator = new Calculator();
      myCalculator.number1 = 5;
      myCalculator.number2 = 5;
      myCalculator.add();
      assertEquals(10, myCalculator.result);
}
```

Figure 2.1 Method testAdd written in the red phase of TDD

Now this code will not even compile! And it is OK with the red phase. There are deadly mistakes in that code, such as using class members without accessors. Also the calculator class doesn't exist.

Now gradually we begin to write any code that makes this test work. We will begin by creating the calculator class (Figure 2.1).

```
class Calculator {
}
```

Figure 2.2 Class Calculator creation in the red phase of TDD

Next, we add the class members as shown in Figure 2.3 and finally the method add as shown in Figure 2.4.

```
class Calculator {
	int number1;
	int number2;
	int result;
}
```

Figure 2.3 Calculator class after adding local variables

```
class Calculator {
	int number1;
	int number2;
	int result;

	public void add() {
	}
}
```

Figure 2.4 Calculator class after adding the add method

Now the code compiles! But the test also fails as shown in Figure 2.5.

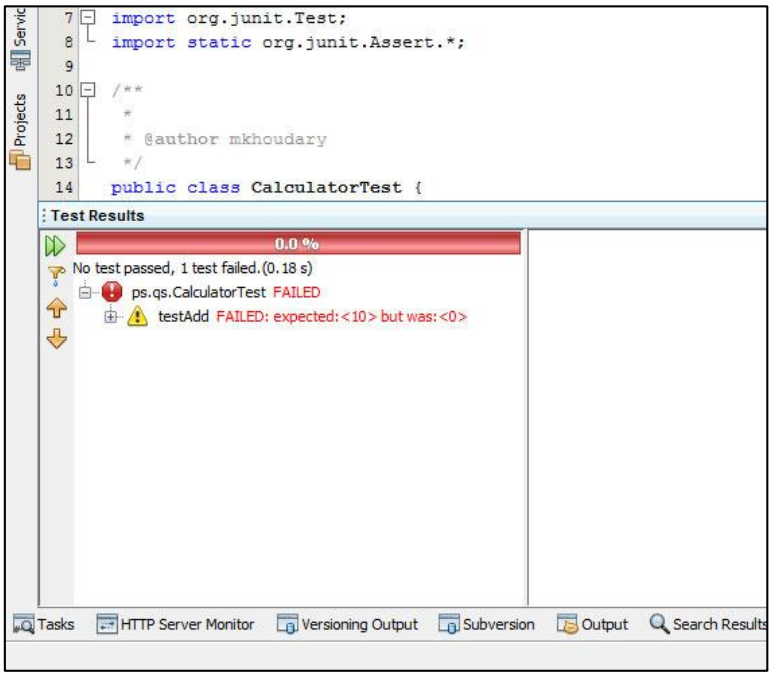

Figure 2.5 Test Case Failure

Now we just make it work. In [7] the author wrote an example with a very gradual development such as faking the result and hard coding it to 10… etc.. but in our case this

would be enough. Now you can write the code that **works** which is the **green** phase in our case.

```
class Calculator {
	int number1;
	int number2;
	int result;

	public void add() {
		result = number1 + number2;
	}
}
```

Figure 2.6 Calculator class that works

And finally; we should now go on refactoring the written code in a way that follows the standards (Figure 2.7). Code refactoring can also break the code. So it is done gradually as well as shown in Figure 2.8 and Figure 2.9.

```
class Calculator {
	private int number1;
	private int number2;
	private int result;
	public void setNumber1(int number1) {
		this.number1 = number1;
	}
	public void setNumber1(int number2) {
		this.number2 = number2;
	}
	public int getResult() {
		return result;
	}
	public void add() {
		result = number1 + number2;
	}
}
```

Figure 2.7 Refactored Calculator class

```
public void testAdd() {
      Calculator myCalculator = new Calculator();
      myCalculator.setNumber1(5);
      myCalculator.setNumber2(5);
      myCalculator.add();
      assertEquals(10, myCalculator.getResult());
}
```

Figure 2.8 Refactored testAdd method

This is a complete development cycle of TDD. Now we will not return to this functionality, ADD, unless its test has failed.

And of course it will not fail except:

- The code of the add method or the class has changed.
- The test case conditions have changed.

These changes in classes or test classes have produced such failure because they didn't follow the standard of modification red/green/refactor. So mixing up development methodologies will not actually make TDD effective enough.

2.1.2 Evaluating the Impact of Test-Driven Development

TDD was evaluated by a study that compared the impact of TDD and non-TDD approaches on quality and overall development time under industrial settings. They showed that TDD served as auto documentation to the code when the code was maintained or used, also the block coverage of unit test level was 79-88%. On the other hand the code developed using TDD increased 2.6-4.2 times when compared to non-TDD developed code. Also the development time increased by 15-35% [37].

In [38] authors found that TDD improves task estimation, process tracking, and more importantly the defect rate was significantly decreased. They also found that people using TDD are more efficient in fixing their code defects.

Another industrial case study in [39] held at IBM also stated TDD showed significant defect rate reduction to the rate of 50%. TDD also saved developers from late integration problems. So the overall development time wasn't affected that much because the time wasted in TDD corresponds the time wasted debugging and fixing in non-TDD case.

Although TDD shows great signs of improvement. However, productivity and development time stayed in debate between researchers.

In semi-industrial settings many experiments were made like in [39, 40, 41, 42, 43] and the quality effects were not as obvious in this context as in the studies done under industrial settings. However, it was obvious that TDD helped in achieving more test coverage [41, 42]. On the other hand, it was criticized that it would produce a false sense of security, because the developer who develops the module is the same testing engineer who writes the test cases, and that may produce more errors on the acceptance test level [43]. Last outcome from semi-industrial experiments is that TDD may not be suitable for all application domains [44].

Finally, in academic settings, many studies [44, 45, 46, 47, 48, 49, 50] experimented TDD on classrooms or on students. Their findings were partly contradictory; as many indicated that TDD may improve software quality significantly. However, in [48] the code quality was noticed to be decreasing while in [45, 49] was increasing. It was also observed that TDD may facilitate implementation and improve developer confidence.

Researchers in [34] introduced a comparative case study on the Impact of TDD on Program Design and Test Coverage. They first went through most of the related studies and summarized their findings. Then they held up an experiment that consists of 3 projects with 5800 to 7000 lines of code. One of them was developed using TDD while others were tested using traditional Iterative test-last. Their findings were that TDD doesn't always produce highly cohesive code, but test coverage was significantly improved.

Other latencies accompanied by using TDD come from the wasted time in maintaining and running the test suite. A solution to this was proposed using continuous testing (CT) [22]. Another solution was in using Regression Test Selection (RTS) and Test Case Prioritization (TCP) techniques [23].

2.2 Regression Test Selection (RTS)

2.2.1 Definition

Regression Test Selection (RTS) techniques attempt to reduce the cost of regression by running a subset of the test suite that might detect defects in code. [52]

An RTS technique is required to be safe. Safety means that the selection process wouldn’t eliminate a test that covers a defected code [52].

Also an RTS technique should have the following features: [53]

- **Inclusiveness** : the extent to which a technique selects tests that reveal faults in a piece of code after being modified – 100% inclusive techniques are considered safe.
- **Precision** : the extent to which a technique omits tests that can never reveal faults in a piece of code after being modified.
- **Efficiency** : measures the space and time requirements for a technique.
- **Generality :** the ability for a technique to work on a wide range of situations/programming languages/testing technologies.

In next section we will revise the related work in RTS and measure the approach on the previous features.

2.2.2 RTS Related Work

An approach called TESTTUBE was introduced in [24]. It is a system for regression test selection. It is claimed to reduce 50% or more of the test cases that need to be retested.

Their approach combines static and dynamic analysis. The system is used with programs written in C.

The system partitions the SUT into entities and these entities are the key elements in the RTS process. Any change to an entity fires the corresponding group of test cases. As can be seen from the partitioning process; such technique can only be used with sequential programming languages as it would be irrelevant in Object Oriented to partition on a procedural basis. Generality is lost in TESTTUBE because its assumptions are based on sequential programming language paradigm and lower level languages. Additionally it needs a number of platform dependent tools that might not be available for all platforms, like app engine for C language, C Information Abstractor (CIA), and more.

It links the generated entities to the corresponding test cases by performing a dynamic analysis to monitor the code flow. This affects negatively the technique efficiency; as all test cases must be rerun to record test cases coverage. Also the change detection scheme considers any entity change. Even if a new line or a console output message statement was added.

TESTTUBE needs to have an old version of the SUT to work. It uses the old version coverage database and the new version to produce a subset of tests that had their dependent entities changed.

And as external tools and dynamic analysis are used to record the code coverage they have no coverage for any invisible or private entity which also affects Precision and Inclusiveness as changes in such blocks will not be monitored or detected.

They evaluated their work on projects and they record a reduction in the test cases with a rate of coverage, but as we mentioned earlier they lost generality and they negatively affected on efficiency, precision, and inclusiveness especially if we want to port such an approach to work with TDD and Object Oriented Programming paradigm.

Another approach in [52] that also needs dynamic analysis to build up control flow graphs. The graphs contain code lines as its nodes. Nodes relations are extracted from execution flow data gathered while performing dynamic analysis. With dynamic analysis private code blocks aren't visible and so inclusiveness and precision are negatively affected.

Their granularity is method-level. Any change in a method is considered a change in code regardless to its actual effect on the related test cases.

Control flow graphs and execution traces need to be stored in disk for the current version of the code. When a new version is produced; new graphs need to be generated and a traversing algorithm runs in parallel between the old and the new control flow graphs. Any change between the two graphs is considered. All test cases affected by this part of code are rerun. This negatively affects the efficiency because generating traces and control flow graphs is required each time for this RTS to work. Tests are considered for rerun even if their code blocks were changed by an empty line, console output message, logging message, or an expression that has no actual effect on the covered tests.

The experimentation results showed that this technique is safe because they discovered all the defects exactly like the retest-all regular technique. Also the technique showed more efficiency when applied to a larger and more complex programs than smaller programs.

2.3 Test Case Prioritization (TCP)

2.3.1 Definition

The goal of TCP is to schedule test cases in an order that increases the rate of fault detection. A test suite that's scheduled correctly can save a lot of time in the worst case than a randomly scheduled test suite. [23]

TCP is another technique for increasing the quality of software testing and also a good way to fasten fault detection as mentioned earlier. Several papers discussed solutions based on TCP like in [29, 53, 54]. These solutions allow testers to order their test cases according to a specific criteria. So the ordered set of test cases is rerun [31]

Unlike RTS; TCP techniques guarantee safety because they do not eliminate any test case. So for such techniques inclusiveness and precision are guaranteed. On the other hand, efficiency remains in question.

TCP can always be used in conjunction with RTS to yield better results. Instead of applying TCP on the whole test suite; it can be applied on the subset selected by an RTS technique [31].

2.3.2 TCP Related Work

In [25] the authors present a TCP system named ECHELON. It compares binary executable files instead of source code.

They stated that binary comparison yields better results in TCP. They also have finer granularity level than the previously mentioned approaches. They work on basic block level instead of method level.

Something that might be in concern is that they use their own infrastructure tools in help to ECHELON and all these tools are specific to Microsoft programming languages. For example they use their code change detection system VOLCAN to detect change in binary level. Here generality is definitely lost, because managed code programming languages like C# and Java generate byte code and not binary files.

As the previous approaches; ECHELON compares two version of the system, and then produces a sequenced list of prioritized tests for rerun.

It's clear that ECHELON will not work well with techniques like TDD, because this would be a problem in efficiency as TDD cycles are fast and repetitive and developers will not take the overhead of generating binary files then creating a newer version to do the comparison process.

They evaluated their tool on various programming projects with various sizes; they evaluated the performance and precision issues. However, the technique in this paper lacks support for byte code based programming languages and also the support for newer techniques like TDD.

The final approach is TEST-RANK [23] and it is an RTS and TCP technique. It depends on dynamic program analysis, static program analysis, and natural language processing altogether.

Their granularity level is method-level as they said this will not affect precision. This might not be correct, as it adds test cases to the RTS even if the change wasn't relevant to the test cases and that of course affects efficiency.

They create a database after doing static analysis on the code to collect some relevant information for their use, like methods beginning and ending line numbers. They also do dynamic analysis, using aspectJ – an aspect oriented library for java, and they collect relevant information for their use. They propose specific metrics they collect during dynamic analysis. These metrics are used in ranking test cases for prioritization.

One important thing to note is that this database is collected offline. It is rebuilt at night when no development activities exist. That means this database becomes obsolete as development goes on. So after database rebuilding the system outcomes will be precise and inclusive, but as the time runs precision and inclusiveness will begin to decrease. For example fixing a problem will not take effect until the database is rebuilt. The requirement of aspectJ decreases generality as well because support for Aspect Oriented Programming is required.

They do affinity analysis based on Natural Language Processing to source code, so they analyze source code for code similarities between methods and test cases using external tools. They start with similarities in method names with test case method names and they also do fragment matching to detect such similarities.

About natural language processing scoring we do not see it that effective, especially with developers with non-standard or non-relating naming habits. The matching process will end up wasting time without finding worthy matches.

And finally they linearly combine the results of metrics and natural language processing score, to give a rank to each method/test pair. This serves as an advice on what

test case to rerun when modifying a specific class and the rank serves in prioritizing these cases.

TEST-RANK needs to run program on a virtual environment with test values then Aspect Oriented Programming is used to trace method calls. After that it collects data for its database. Such an approach cannot work online in a CT (Continuous Test Runner). Although authors stated the importance of CT on software testing.

In summary TEST-RANK uses the database and metrics built offline to suggest test cases to a developer to run when changing in a code block (method). All newly added/modified test cases will fall out of this as they need to be synchronized at night. Also method granularity level will not differentiate between code lines relevant to test cases and written for other purposes such as debugging and user interface manipulation.

About the metrics; authors said that these metrics/predictors help in ranking test cases. Test cases with a rank of zero will not be considered. They create a prioritized subset of test cases for rerun.

Here follows a list of TEST-RANK predictors:

- **Execution Count** gives higher score for methods being executed more in a test case. This metric is irrelevant as the whole test case could depend on a method that calls a logging method in a for loop, so the logging method takes a higher rank than the real method.
- **Call Count** counts distinct call locations in a specific test case. This also seems irrelevant because a logging method could be called from various locations. And as mentioned before, logging methods doesn't affect execution at all.
- **Stack Depth Sum** gives higher score to closer methods to test case. While this predictor seems logical and gives a meaningful indication; in certain designs the target call would take less priority because it's behind a mediator, facade, observer... etc.
- **Value Propagation** measures the intersection between test/method pairs depending on similarity between passed variables. it predicts the link between a test case and a method through calculating the intersection between values in a test case and passed values in a method. Why would we need such a predictor since we can know for sure that this test case invokes this method. Stack depth sum predictor gives close results.

- **Inverse Coverage** measures the importance of a method *m* to a test case *t* when the test case only contains a call to *m*. This predictor seems significant. However, in large test cases where many relevant methods are being will be all equally important and this measure will not work.

There are also predictors collected from the natural language processing phase:

- Measuring similarities between two given segments of code. They use a tool called WordNet lexicon and a tool called CodePsychologist to assign each (test, method) pair a score between 0 to 1 based on textual similarity between word groups. So any difference in the developer style of coding would rule out this predictor as TEST-RANK actually locates test cases that look after methods.

They combine this predictor with a coverage filter. So if the method isn't covered they will not do natural language searching on it. Since they already know that this test case is related to these methods; why would they need to do such a processing phase?!

The affinity usage seems fine. But since we're going to collect a database why not collecting the actual invocations and why not knowing the exact affinity from the calls this test case makes?

For testing their approach they use Jumble Mutation Operators. But since their granularity is on method-level they just detect the change on a method without checking if it is relevant to the test case or not.

2.4 Continuous Testing (CT)

Continuous Testing was introduced in [22]. It uses real-time integration with the development environment to asynchronously run tests that are always applied to the current version of a program.

In [22] researchers used CT as another way for reducing wasted time during testing. They base their study on a model of developer behavior during development time. They showed that the greater amount of wasted time during testing is due to the slowness of running the whole test suite.

They also showed that the more ignorance time between running the test suite and developing; the more regression errors. And with regression time increase regression tests become harder to find.

They stated that the faster an error is detected the easier it is fixed for the following reasons:

1. More code changes must be considered to find the changes that directly pertain to the error.
2. The developer is more likely to have forgotten the context and reason for these changes, making the error harder to understand and correct.
3. The developer may have spent more time building new code on the faulty code, which may also needs to be changed.

They also stated that they can make use of the CPU free cycles during development in running test cases from the test suite. By reporting the error produced by the failure of a test case they could notify the developer about something he/she broke in the code during development.

They built a CT and they prioritized test cases using one of the following techniques:

- **Suite order:** Tests are run in the order they appear in the test suite, which is typically ordered for human comprehensibility, such as collecting related tests together, or is ordered arbitrarily.
- **Round-robin:** Like the suite order, but after every detected change, it restarts testing at the test after the most recently completed one. This is relevant only to continuous testing, not synchronous testing.
- **Random:** Tests are run in random order, but without repetition.
- **Recent errors:** Tests that have failed most recently are ordered first.
- **Frequent errors:** Tests that have failed most often (have failed during the greatest number of previous runs) are ordered first.
- **Quickest test:** Tests are ordered in increasing runtime; tests that complete the fastest are ordered first.
- **Failing test:** The quickest test that fails, if any, is ordered first.

These prioritization techniques are not efficient enough when working with huge projects and hundreds of test cases. Or when working with different modules. For example suite order, round robin, and random are completely unpredicted and are less likely to be effective. Imagine that the test cases of the last module exists at the end of the test suite. The CT needs to run all the test cases to reach the tests for this specific module.

Also recent errors, frequent errors, quickest test, and failing test aren't always the best or the good techniques to follow. For these techniques moving to a new module wouldn't run it's test cases until the CT runs all the frequent, quickest, failing … etc.

Their experiment was to develop two applications one on Perl and the other on Java and to study measurements like Test-Wait, Regret, Wasted Time, and Improvement among these experiments to prove the improvement of CT.

The experimental results showed that CT decreased wasted time by 92-98% which is considered superb.

Anyway the number of test cases in the test suite in this study isn't clear. Although their proof about the benefits of CT are obvious and clear; the prioritization techniques they mentioned aren't that effective with the larger test suite size.

Chapter 3: The Concept of Influence Graph

As we've seen in Chapter 2; various approaches to RTS techniques relied mainly on dynamic analysis accompanied by static analysis or natural language processing. We've shown that dynamic analysis isn't necessarily beneficial especially if we're intending to use such techniques with a CT.

In this chapter we present our approach to RTS, the influence graph, which will be the online database used for RTS.

3.1 Definition

An influence graph, from its name, is a graph that reveals the influence of code lines and code blocks on other "special" code lines based on programmatic dependency. So nodes inside such graph are simply line numbers for a specific source code.

For example let's assume our "special" code that we need to measure influence to is the return line of the method add from following code shown in Figure 3.1.

```
1  public int add() {
2        int number1 = 10;
3        int result = number1 + 5;
4        return result;
5  }
```

Figure 3.1 One method influence code fragment

From what we can observe; we can say that line 2 **influences** line 3 and line 3 **influences** line 4, so line 4 is **influenced by** line 3 and line 3 is **influenced by** line 2.

The influence relation came from the variables in a line that influence the result or outcome of another line, so line 2 influences line 3 because of **number1**, and line 3 influences line 4 because of **result**.

So derived from the above we notice that line 2 **directly** influences line 3 also line 2 **indirectly** influences line 4; meaning any change occurs to line 2 reflects changes to all directly and indirectly influenced lines.

Also influence relationship can be between a method and a line. From Figure 3.2 we can see that line 4 is directly influenced by two components; first one is code line 3 and the second is the method **normalize**.

This method could be in the same class or in another class. When a change occurs to the method we know that this change will affect line 4 in the method divide.

```
1  public float divide() {
2        float number1 = 10;
3        float result = number1 / 5;
4        return normalize(result);
5  }
6
7  public float normalize(float number) {
8        return Math.round(number);
9  }
```

Figure 3.2 Two methods influence code fragment

And as we said earlier for our influence relationship; the influence is measured inside the method context; meaning the influence relationship of line 4 only contains these three influencing components. To get the influence relationships of the **normalize** method we need to go deeper an extra level. This depth parameter is put for the purpose of giving the user the ability to tune performance and accuracy as we will see in later sections.

3.2 Visualizing Influence Graph

Figure 3.1 shows a simple influence graph for code fragment 1. We can see the influence relationship between lines in the method **add**. Now from this relationship a change in line 2 would reflect a change **indirectly** on line 4 and so the return value of the method **add** might change.

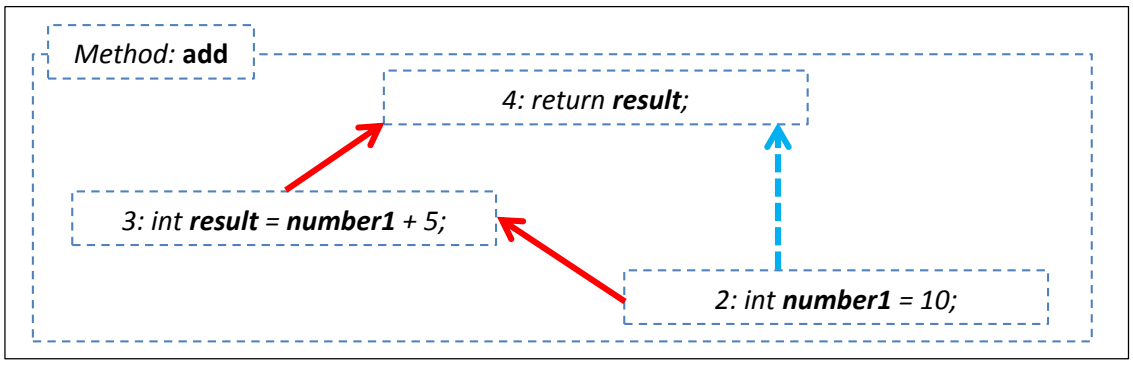

Figure 3.3 Influence Graph for code fragment shown in Figure 3.1

Now for the code fragment shown in Figure 3.2 we can see the influence graph in Figure 3.4, we can observe that the method "**normalize"** influences line 4 in the method "**divide**". If the return value of "**normalize**" changes, the return value of the method "**add**" might change as well.

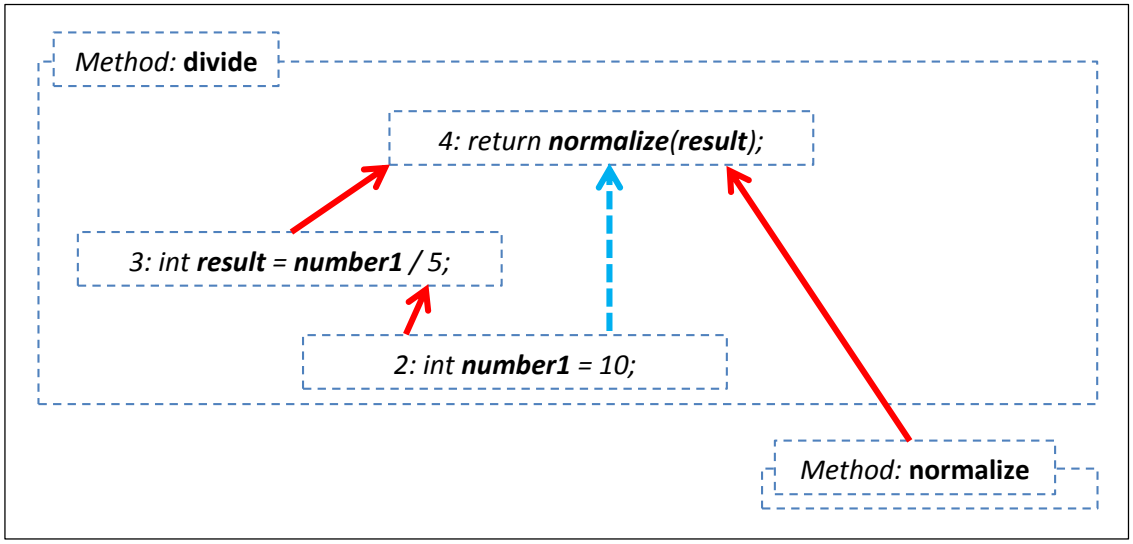

Figure 3.4 Influence Graph for code fragment shown in Figure 3.2

3.3 Depth of Influence Graph

The depth of an influence graph presents the number of levels the influence will draw starting from the "special" code. For example for a graph with depth = 1; method calls will not be analyzed. In case of Figure 3.2 the method **normalize** will not be considered.

As the depth increases; the analysis of influence will cover up more calls in the flow. A depth of zero means that the analysis will analyze influence recursively until no extra levels are found.

Such a parameter helps so much in balancing performance and precision; the more depth the more precision and the less performance. Reaching an optimal level of depth depends on the nature of the project as we will show in later sections.

3.4 Usage of Influence Graph in RTS

The more metrics an RTS technique requires; the more time and space required to gather these metrics. The more time required for metrics gathering like in [23]; the wider the gap between the RTS database and the actual situation during development.

Also many of the metrics proposed in various papers including [23] weren't that effective to the process of RTS; they managed to add natural language processing along with metrics/predictors to be able to know what test cases affected by which methods, but when looking at influence imagining test cases as the first node in an influence graph; then we can build an influence graph for a test case to show exactly what methods affect this test case and based on our search we can either take close methods if we specify less depth or reach more precise results by increasing depth to get finer granularity, in the statement level recursively.

Figure 3.5 shows typical influence graphs for a test case. We specify that the "special" code that we need to analyze for influence in the test cases are the assertion lines. Because in unit testing these assertion lines are the ones that make a test succeeds of fails. And for the more in depth method calls we specify the "special" code to be the return lines. In this assumption only lines and methods that really affect the assertion or the return lines will be included in the influence graphs.

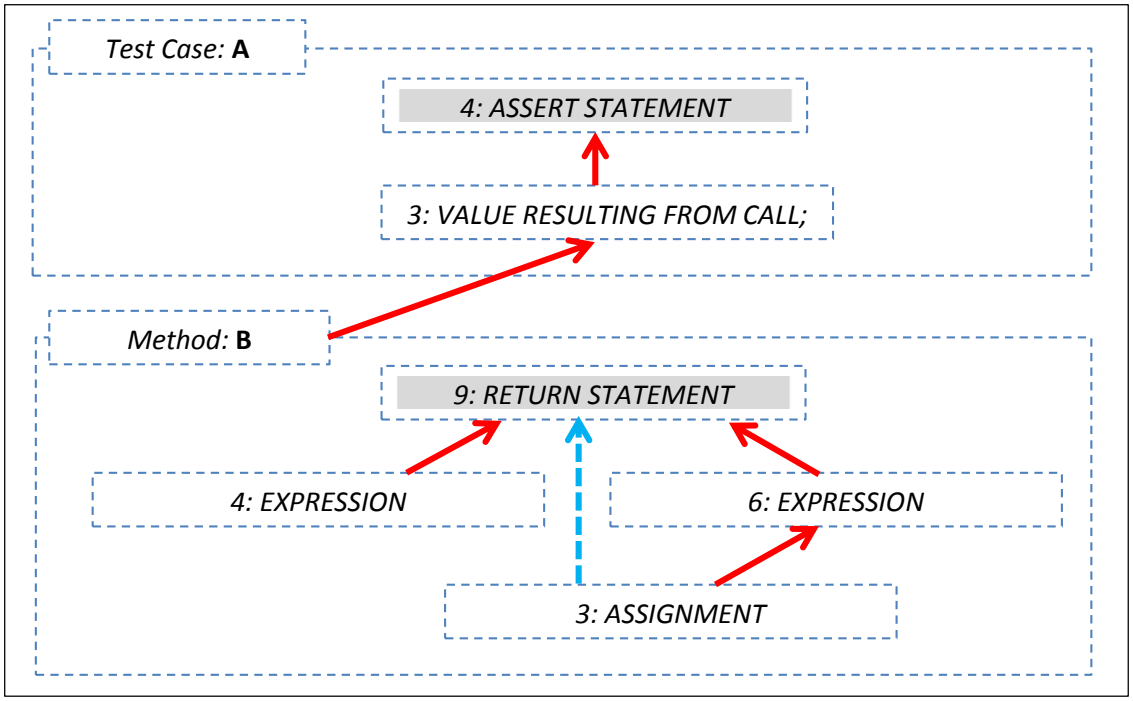

Figure 3.5 A typical influence graph for a test case

3.5 Benefits of RTS with Influence Graphs

In [52] researchers needed to regenerate the control flow graphs of their source code every time they need to compare two versions of a program, and that is done by using dynamic analysis.

Unlike this approach; the influence graph is built and enhanced progressively while the code is written and maintained. The influence graph is built by analyzing source code. For applications that didn't use influence graph RTS from the beginning, all has to be done is to run the influence graph analyzer on the project and it will start static analysis to produce all the required influence graphs for the progressive work later on.

Another benefit over approaches like [23] is the storage space required for the database of the influence graph is only graphs nodes and links between them. Additionally ranks would be required if TCP is deployed with RTS.

Another strong benefit is that only code lines that affect the assert lines or the method return code lines will be included in the influence graph; so no irrelevant change notifications will be fired when adding console printing statements, empty lines, or even complicated statements that do not contribute in the final return value or in any of the assertions. This is a novel achievement over the shown related work [22, 23, 24, 51] , the code fragment in Figure 3.6 includes irrelevant modifications from the previous code fragment in Figure 3.2.

```
1   public float divide() {
2         float number1 = 10;
3
4         float result = number1 / 5;
5         System.out.println("Number is " + number1);
6         return normalize(result);
7   }
8
9   public float normalize(float number) {
10
11        return Math.round(number);
12  }
```

Figure 3.6 Code fragment in Figure 3.2 with irrelevant modifications

The influence graph for such a code is the very same graph generated in Figure 3.4; with only differences in line numbers.

No modifications to any of the influence graph nodes means no tests to be rerun.

Also platform/programming language independence is a very essential key factor in influence graphs. They can work with sequential as well as object oriented programming languages. They can work with programming languages with virtual machines or native languages, they can work with any testing methodology including Unit Testing as described in this context, they can work efficiently with TDD as they give instant response in development time as will be shown later on.

Another advantage is that influence graphs are driven from source code, they are able to map and monitor private and hidden code blocks, unlike approaches like [23, 49].

3.5.1 Influence Graphs and Influence Indexes

Influence graphs are only generated to methods that contribute directly or indirectly to test cases. An influence graph for a method is only created once then referenced by all relevant test cases.

In order to check if a line under modification is part of an influence graph or not two ways are possible. One of them requires no extra storage but processing which is navigating through influence graphs to check for this line. Another way that requires storage but saves a lot of processing power is to build up an index, in database or in memory, that contains the lines that influence the return value or the assert node.

Influence indexes can give response about if a line affects an influence graph or not. The faster the index is the faster the response will be.

3.6 Influenced Test Cases

The term Influenced Test Cases refers to the subset of test cases that are influenced by a specific code line, meaning if a code line influences a return value of a method that affects another method that affects a group of assert statements; the test cases that contain these assert statements are marked as the influenced test cases.

Influenced Test Cases are actually the exact outcome that an RTS technique requires. And as we've seen influence graphs can derive this subset without much processing unlike the approaches [22, 51] and others that require dynamic code analysis and parallel traverse.

A very important question arises here is that why would we need an influence graph since we gather the influencing line numbers in an index?

From the index we can know that this code line affects test cases and we can retrieve the influence graph that contains such a code line. But to get the influenced test cases we need to do a very quick traverse not through influence graph nodes but through influence graphs themselves to reach this list of test cases.

Influence graphs maintain online modifications, our approach is meant for progressive and online testing. An index will not give much information on which nodes to add and which nodes to remove so we will need to rebuild the whole influence graph again to rebuild the index. Having the influence graph built and reflecting code modifications on it partially rebuilds the index quickly and with only the required modifications saving up a lot of memory and processing power.

3.7 Building Influence Graphs

As we mentioned earlier in this chapter; influence graphs are meant to be progressive. It can be online and up to date anytime it is used. We also mentioned that influence graphs are built directly from source code with static analysis, and in this context no specific static analysis is required, just several events and data need to be gathered along during the development process. Any IDE with decent abilities can provide such information.

3.7.1 Source Code Analysis versus Byte/Binary Codes Analysis

An important question arises here; why using static analysis on source code instead of static analysis on byte code or binary files?

At first we ruled out binary files in Chapter 2 when we discussed approach in [25] as it would be an overhead to generate binaries each time you need to do TCP or RTS. Also managed code programming languages like C# and Java are built over virtual machines so their binary files are mixed with the virtual machine code and there will not be an easy way to distinguish user code from virtual machine code. Finally this will not help much with technologies like TDD as it requires extra processing in building and linking files.

Using source code analysis versus byte code analysis was in debate between tools-developers and researchers like in [55, 56] and others. They specified that depending on the developers needs they can choose between source code and byte code.

About byte code; we preferred using source code for the following reasons:

1- We do not want to make our technique language specific; dealing with source code would only require collecting some data during writing code. But if we built our technique on byte code we will not be able to extend it to programming languages with no byte code such as PHP, RUBY … etc.

2- We need to build influence graphs progressively in development time, small source code changes could reflect bigger byte code changes and so changes will be harder to monitor and reflect. Unlike source code where I know exactly where the modification occurred so reflecting this modification will be exact and direct.

3- Influence graphs and influence indexes rely on code line numbers. With byte code line numbers will not be indicative and will require translation to be reflected on byte code.

4- In many cases byte code is preferable over source code especially to Figure out specific features or to draw out some information from the code like Nullness analysis in [58] and other information like unused methods or variables. For such studies byte code is really required as it could do analysis to classes with no source code. But in our case classes with no source code will not subject to any change so no need to include their interior design in any influence graph. And during development we have source code in hand so no need to generate byte code for that purpose.

5- We shall not forget that we need our technique to run online; having to create byte code every time a change occurs then analyze this byte code to detect differences will require an amount of time that will decrease efficiency.

3.7.2 Dynamic Code Analysis

Approaches [23, 24, 51] used dynamic analysis in RTS and TCP for the purpose of testing two software versions for modifications.

Dynamic analysis is language specific unless binary analysis is done like [25] and in such cases many language specific third party tools are required, also linked binary files must be generated in order to perform such analysis.

Also dynamic analysis needs to run code with test data to be able to extract the required information and metrics. So it will be very hard to synchronize information with a CT online.

For the previously mentioned reasons we didn't use any dynamic analysis in our approach.

3.7.3 Influence Graph Building Conditions

The build of influence graph goes on as long as the program compiles. When the program has compilation errors the influence building process stops modifying but continuously monitors modified lines and marks them for manipulation as soon as the program compiles again.

Also building process is triggered based on a specific event; various events may be used to trigger the building process. This may include:

1. As typing code goes on.
2. When the developer leaves the current line.
3. When the developer saves the current file.
4. When the developer leaves the current line and stalls for a time.
5. When the developer exits the current code block.

Any of the previously mentioned events can be used. However, to achieve the best interactivity without distracting the developer with rash information the options 4 or 5 could be used.

3.8 Influence Graph Creation Algorithm

We mentioned earlier that this approach is progressive, but the creation algorithm can run on legacy projects to create the required influence graphs to be able to port older projects with our approach.

The creation algorithm starts from a test case and begins creating influence graphs to the depth desired. More depth means more **inclusiveness** and **precision** but less **performance**.

3.8.1 Creation Algorithm Basic Version

The algorithm starts with a test case method and starts building its influence graph from **assert** statements and other lines influence them. After building this root influence graph the algorithm checks depth, if maximum depth still not exceeded the algorithm

proceeds to create influence graphs for each method call. These graphs are built from **return** statements and other lines influence them. This process will continue recursively until no more levels exist or maximum depth is reached.

The algorithm will take as input the method, the root influenced node type, the caller line, and the current depth. Here follows description for the inputs.

- **Method**: is the method the algorithm is intending to analyze.
- **Root Influence Node Type**: is the type of "special" code that needs to be considered for root influence. It can be **assert** for test methods and **return** for other methods.
- **Caller line**: is the line that called this method. For test cases this parameter is set to null.
- **Current depth**: is the depth of this method call. It starts with 1 for test cases.

The basic version of the algorithm is shown in Figure 3.7.

3.8.2 Influence Graphs Registry

The previous algorithm has a problem is that it will create influence graphs for such methods multiple times and will re-analyze each method each time it appears in code. This will reduce performance. A solution to that issue would be in the "Influence Graphs Registry".

This registry is in charge of registering any newly created influence graph. When an analysis process starts it checks the existence of a method influence graph in that registry. If a match is found the analysis process will simply bind to that influence graph and the analysis stops. If not the analysis will continue as regular and the generated graphs will be registered.

The increase of depth increases **inclusiveness** and **precision** but may reduce **performance**, as this is unwanted especially with an RTS technique that's intended to be used with a CT; the existence of the registry reduces this problem as methods will only be analyzed once and then used anywhere else.

So the code in Figure 3.8 will be added to the beginning of the algorithm.

```
createInfluenceGraph(method, rootNodeType, callerLine, depth) {
    newInfluenceGraph = create and register empty influence graph;
    if callerLine is not null then
        link newInfluenceGraph to callerLine;
    end if;
    list rootInfluencedNodes =  find all lines
                                with type rootNodeType
                                in method;
    for each root in rootInfluenceNodes loop
        link root to its dependent lines;
        add root to newInfluenceGraph;
    end loop;

    if depth >= maximum depth then
         return;
    end if;

    list methodCalls = find all lines
                       with method calls
                       in newInfluenceGraph;

    for each calledMethod in methodCalls loop
         createInfluenceGraph(calledMethod,
                              Return_Node,
                              callerLine,
                              depth + 1);
    end loop;
}
```

Figure 3.7 Influence Graphs creation algorithm basic version

```
search influence registry for influenceGraph for method;
if there exist influenceGraph then
      if callerLine is not null then
            link oldInfluenceGraph to callerLine;
      end if;

      return;
end if;
```

Figure 3.8 Creation algorithm addition for Influence Graphs Registry

3.8.3 Influencing Variables Dependency

The last piece of information required in an influence graph is the list of influencing variables, this list might not be important during the creation process but when an update process occurs such a variables list for all the directly and indirectly influencing variables will be required to associate a new node to the influence graph.

3.8.4 Variables Context Sensitivity

Used variables could exist in the same block or in any higher level block that share a visibility scope with the currently analyzed block. For that purpose variables are searched from the block under analysis and recursively upwards until the declaration line is reached.

Siblings to the block under analysis and its sibling ancestors aren't searched for variable declaration existence because variables existing in the siblings scopes can never be seen. Except for static variables and this is solved implicitly as shown in the next code fragment in Figure 3.9.

```
public class Constants {
	public final static String MY_DATA = "Data";
}

public class User {
	public void print() {
		System.out.println(Constants.MY_DATA);
	}
}
```

Figure 3.9 Code fragment showing static variables scoping

As we can see Constants.MY_DATA refers clearly to the location of MY_DATA, it is in the declaration section of the class Constants.

For variables in scopes higher than the method under influence analysis, these variables will be associated with influence graphs as global variables, so any change to these global variables on any method even if not in a specific influence graph will trigger retesting for all test cases associated.

3.8.5 Loops and Conditional Blocks

As mentioned earlier our approach has the finest granularity of statement-level. The variables dependencies are measured on the block level. Our influence graph deals with for loops, if/else, try/catch… etc. as blocks and any variable declared inside these blocks is only visible for these blocks and can only be considered if it changes the return value of the method.

A special case to loops and conditional statements is that variables participating in loops control and conditions are considered for influence.

```
1  public float calculate(int variable) {
2        if (variable == 10) {
3              return 11.0;
4        } else {
5              Return 12.1;
6        }
7  }
```

Figure 3.10 Code example where control variables influence the return value

In the code fragment shown in Figure 3.10, **variable** is considered to influence as it participates in the return value. Such variables are linked directly to the return line in influence.

In the code fragment shown in Figure 3.11 **variable** will not be considered for influence because it doesn't affect the return value.

```
1  public float calculate(int variable) {
2        if (variable == 10) {
3              System.out.println("Ten");
4        } else {
5              System.out.println("Not Ten");
6        }
7
8        return 12.0;
9  }
```

Figure 3.11 Code example where control variables do not influence the return value

3.8.6 Recursion

As all we need is only to detect changes that influence the return value of methods recursion is fixed at the point to recalling self for recursion, so the influence will be measured to only the current call.

3.8.7 Influence Indexes

The influence indexes are maintained and enhanced during the influence creation phase as well as the influence update phase, as we shall see later. In general, anytime a node is added or removed to/from an influence graph registry records are updated.

3.8.8 Dynamic Dispatch Problem

The dynamic dispatch problem arises when polymorphism is used, an instance of a class is declared by the interface class. The problem here is that it will not be known which implementer class will be instantiated and so the influence graphs for the specified class will not be created as it is unknown.

For classes' instantiated in the same line like this:

```
Shape shape = new Circle();
```

It is already solved by directly considering shape as **Circle** in our analysis, but for more complicated cases when using a design pattern like Factory or when specifying the implementation conditionally two solutions are possible:

1. To cover all the implementations with influence graphs and bind them all to the caller line. This doesn't seem promising as discovering all implementer classes needs extra analysis and will be in vein as only one implementation is used.
2. As test cases are associated with testing values; static analysis for these declarations can be made to evaluate which implementer is used. Although this seems promising. However, an influence graph for a method is not with one test case so for an influence graph to depend on specific testing values it would be a huge decrease in reusability and so degradation in **efficiency**.

Our proposed solution to that issue is a mixture between the two solutions as follows:

1. When a direct assignment occurs the implementer class will be considered.

2. When assignment occurs through conditional statements or factories, static analysis will run to specify the possible subset of implementers used in this context and will do influence graph analysis on them all.
3. Any change on any of the possible implementers will be considered as a change for the RTS algorithm as we shall see in latter sections.

This solution is safer because it guarantees the **inclusiveness** requirement as dropping changes that might affect test cases will make the technique unsafe.

So the update to the algorithm is that when we get **methodCalls** for further analysis; methods for all possible implementers will be included for analysis.

3.8.9 Methods with No Return Value

Another issue comes with methods with no return value. We know that influence graphs either depict the influence of code over an assert line or a return line, but for methods with no return value the whole thing differs as they will neither have assert nor return lines.

For this case a method with no return value is considered as:

1. An extension in code lines for a passed parameter if it is not a primitive type, so the influence of the lines inside the method with no return value will extend the influence of the caller method.
2. Global variables will be considered for global variables influence implicitly as discussed in Section 3.8.4.
3. Methods with no return value that take primitives as parameters and make no change to global variables will be neglected.

In this case such methods will be measured for influence and no changes will be missed.

The final version of the creation algorithm in shown in Figure 3.12.

3.9 Influence Graph Update Algorithm

The creation algorithm makes our approach equal to other approaches that depend on creating the RTS database offline at some time when no development occurs. If we run the influence graph creation algorithm for all test cases each night as in approach [23] we will be able to give accurate and precise RTS. But by the middle of the development

day or when development process is fast this database will start to age quickly and will be misleading and outdated.

The influence graph update algorithm is the one in charge of making our approach progressive. As influence graphs will morph and transform by the addition, modification. Pruning operations are made by this algorithm in effect to code changes made by developers.

```
createInfluenceGraph(method, rootNodeType, callerLine, depth) {
   search influence registry for influenceGraph for method;
   if there exist influenceGraph then
       if callerLine is not null then
           link oldInfluenceGraph to callerLine;
       end if;
       return;
   end if;
   newInfluenceGraph = create and register empty influence graph;
   if callerLine is not null then
       link newInfluenceGraph to callerLine;
   end if;
   list rootInfluencedNodes = find all lines with type rootNodeType
                              in method;
   for each root in rootInfluenceNodes loop
       link root to its dependent lines;
       add root to newInfluenceGraph;
   end loop;
   if depth >= maximum depth then
       return;
   end if;
   list methodCalls = find all lines with method calls
                      in newInfluenceGraph;
   for each calledMethod in methodCalls loop
       createInfluenceGraph(calledMethod, Return_Node,
                            callerInfluenceNode, depth + 1);
   end loop;
}
```

Figure 3.12 Influence Graph creation algorithm final version

3.9.1 Update Algorithm Basic Version

The algorithm will be executed when a modification in a code line occurs. The update will be triggered by the IDE.

First the algorithm will check if this code line is part of an influence graph or if the modified code line contains at least one of the variables that influence an influence graph. If no match found the algorithm will stop, otherwise the influence graph will be loaded, and the original line will be compared to the modified line to extract two lists:

- List of variables that were added to the new line.
- List of variables that were removed from the new line.

If neither list contains variables then the algorithm will exit, otherwise added variables will be reconsidered in the influence graph and will be ported and linked along with their dependent code lines. New method calls will be considered and linked to their corresponding influence graphs.

The list of removed variables will be considered and for each variable a pruning process will start recursively removing nodes if the only link between the current node and the influencing node is just this variable.

Inputs to this algorithm are the Influence Graph, the original code line, and the modified code line.

- **Influence Graph**: is the graph detected to own the changed line.
- **Original Code Line**: is the code line before modification.
- **Modified Code Line**: is the code line after modification.

Influence graph update algorithm basic version is shown in Figure 3.13.

3.9.2 Influence Indexes

As we mentioned earlier, one problem that prohibits using only indexes to discover change in test cases is that with modifications indexes need to be completely torn down and rebuilt. That would mean a huge overhead that would ultimately kill this suggestion.

So for indexes to be partially rebuilt and enhanced the update algorithm should put that into consideration.

The consideration is so simple; when a variable is removed from influence:

1. Corresponding nodes are tested for other variables influence, if no influence from other variables then this node will be pruned.

2. Each node pruned is removed from indexes until we reach another influence graph.
3. In such case if this influence graph only influences the influence graph under test then the influencing graph will be put offline, so any index hit will be considered a miss after testing the influence graph status.

```
updateInfluenceGraph(influenceGraph, oldLine, newLine) {
      if influeceGraph does not include oldLine AND
         newLine variables does not influences influenceGraph then
            return;
      end if;
      list addedVariables = find variables in newLine
                            not in oldLine;
      list removedVariables = find variables in oldLine
                              not in newLine;
      if addedVariables AND removedVariables are empty then
            return;
      end if;
      for each addedVariable in addedVariables loop
            find all code lines depending on addedVariable;
            link code lines to influenceGraph;
      end loop;
      for each removedVariable in removedVariables loop
            inspect nodes that only depend on removedVariable;
            recursively remove nodes from influenceGraph;
      end loop;
      list methodCalls = find all lines with method calls
                         that were added to influenceGraph;
      for each calledMethod in methodCalls loop
            createInfluenceGraph(calledMethod,
                                 Return_Node,
                                 callerInfluenceNode);
      end loop;
}
```

Figure 3.13 Influence Graph update algorithm basic version

This way improves **performance** as if we have, for example, 10 lines that use the same variable that used to influence the return value and now it is not. Say we have 5

calls to other 5 influence graphs with 20 lines each, that means we will need to go through 110 nodes to remove from index, this is a very simple case but imagine these 5 influence graphs are influenced by other 10 influence graph each, with each influence graph having 20 lines, the result would be 1110 nodes to be inspected and removed.

So the proposed solution simply makes us iterate through the influence graph basic lines, 10 lines in our previous example, in addition to only the adjacent influencing graphs, 55 in our example, to put them offline. Meaning only 65 inspections needed instead of 1110 and if the recursion in the previous example took us one more level in depth there will be a huge increase in the number of nodes for retest in the first case while the second case will merely increase by the number of influencing graphs in each level.

So the algorithm will have the addition shown in Figure 3.14 for index modifications.

```
recursively remove nodes from influenceGraph and remove index
      entry if no other links to that node;
```

Figure 3.14 Influence Graph update algorithm addition for index manipulation

Please note that any attempt to bind an offline influence graph to an influenced node will reactivate this influence graph immediately.

3.9.3 Pruned Influence Graphs

As we mentioned in the previous sub section; pruned influence graphs are put offline while their values remain in the index. This was made for the sake of **performance** from two prospective:

1. We do not need to remove all index values while in development to preserve processing power.
2. We do not want to remove these indexes and influence graphs because developers might use them so soon. Developers may only comment code lines to test something so removing these indexes and graphs immediately would mean the need to rebuild these influence graphs and indexes again.

Anyway **efficiency** isn't only about performance; storage is also a very important actor, so these pruned influence graphs can have the following states:

1. Immediately after pruning: influence graphs and influence indexes will be left in memory but marked offline.

2. After a specific period **X**: influence graphs are removed from memory but left in disk while influence indexes remain in memory but marked offline.
3. After a specific period **Y** > **X**: Influence graphs and influence indexes are removed from disk and memory.

Periods **X** and **Y** can be specified according to developers and development needs, also can be tuned with the available resources, for example huge storage and good RAM but limited processing power will make us increase **X** and **Y**, while limited storage will make us push **X** further, finally limited storage and RAM will make us put **X** and **Y** relatively close and small. In conclusion tuning is done freely to cope with the available resources.

3.9.4 Modifications and RTS

The most appropriate place to begin regression tests selection especially since we intend to make a continuous test runner is this algorithm. Whenever modifications occur, the algorithm should traverse through influence graphs and mark the corresponding test cases as dirty for retesting. So the RTS algorithm is triggered from the update algorithm, after modifications are done. So the modification shown in Figure 3.15 should be added to the end of the update algorithm.

```
trigger Influence Graph RTS on influenceGraph;
```

Figure 3.15 Influence Graph update algorithm addition for triggering RTS algorithm

3.9.5 Dynamic Dispatches

As we illustrated the proposed solution for the dynamic dispatch problem in our case, during pruning if any implementer dropped from the coverage the bond with its influence graph will be removed and the influence graph will be put offline.

Influence Graph update algorithm final version is shown in Figure 3.16.

3.10 Influence Graph RTS Algorithm

As mentioned in Section 3.9.4 the third algorithm is the one in charge of RTS. The results of this algorithm can be taken immediately and utilized by the IDE.

The output of this algorithm can subject to TCP at a later stage, making the proposed system capable of doing RTS with or without TCP with no modifications to the core system.

```
updateInfluenceGraph(influenceGraph, oldLine, newLine) {
      if influeceGraph does not include oldLine AND
         newLine variables does not influences influenceGraph then
            return;
      end if;
      list addedVariables = find variables
                            in newLine
                            not in oldLine;
      list removedVariables = find variables
                              in oldLine
                              not in newLine;
      if addedVariables AND removedVariables are empty then
            return;
      end if;
      for each addedVariable in addedVariables loop
            find all code lines dependent on addedVariable;
            link code lines to influenceGraph;
      end loop;
      for each removedVariable in removedVariables loop
            inspect nodes that only depend on removedVariable;
            recursively remove nodes from influenceGraph AND
            remove index entry if no other links to that node.
      end loop;
      list methodCalls = find all lines with method calls
                         that were added to influenceGraph;
      for each calledMethod in methodCalls loop
            createInfluenceGraph(calledMethod,
                                 Return_Node,
                                 callerInfluenceNode);
      end loop;
      trigger Influence Graph RTS on influenceGraph;
}
```

Figure 3.16 Influence Graph update algorithm final version

The algorithm starts at the influence graph that contains or contained the modified code line, the algorithm queries what code lines this graph influences and for each influence graph a traverse to influenced graphs is made until test cases are reached, the test cases are then gathered in a subset for retesting.

For the **performance** sake the traverse will not be made on code lines inside influence graphs as it would be pointless. Instead, the traverse will start from the modified code line influence graph and will hop to the influenced graphs one after another until leaf influence graphs, which contain test cases, are reached and that's a huge decrease over traversing through code lines.

Figure 3.17 illustrates the traverse made by this algorithm.

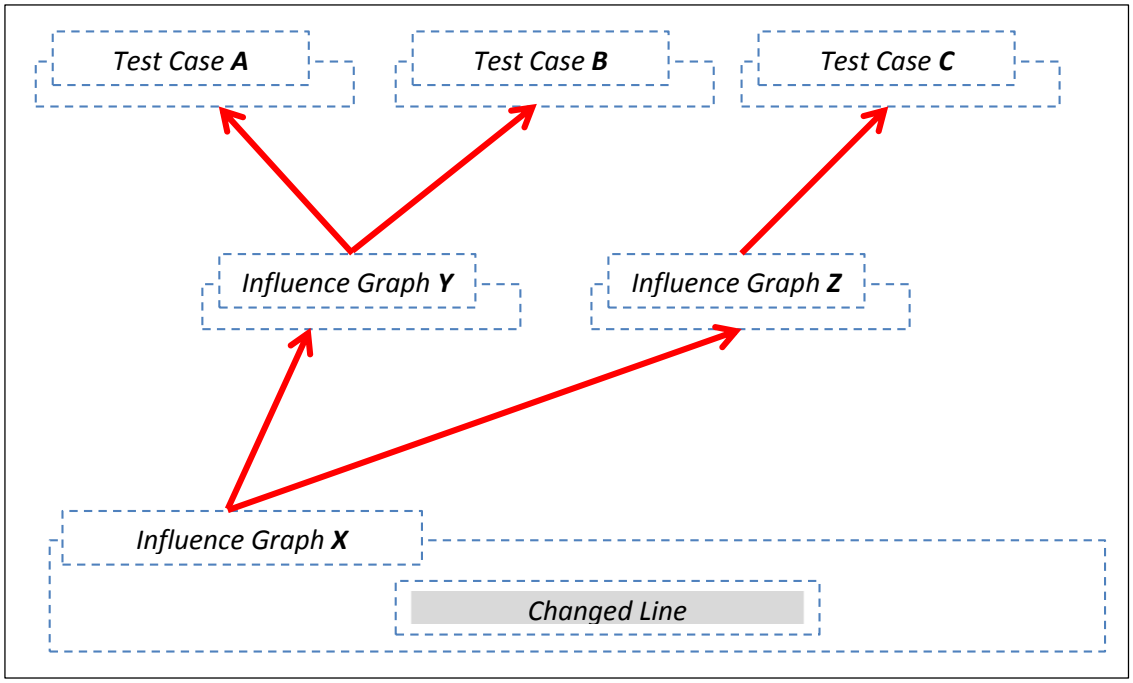

Figure 3.17 Influence Graph RTS algorithm traverse

As shown in this Figure test cases A, B, and C are influenced by the modified influence graph **X,** this graph influences **A** and **B** through the influence graph **Y** and influences **C** through the influence graph **Z**.

As we can see if we considered graph X to contain 30 lines, graph y and z to contain 50 lines each, then if we were to do regular traversing, like the one in [52] we would have traversed through 130 lines. With our algorithms we could detect the influenced test cases only by three hops for the same example.

Naturally methods count in a project will never be even as half as code lines and so traversing through methods is definitely better than traversing through lines.

The input for the algorithm is only the influence graph that had the modified code line. The final version of the RTS algorithm is shown in Figure 3.18.

```
getInfluencedTestCases(influenceGraph) {
      traverse through influence graphs that influenceGraph
      influences;
      list influencedTestCases = gather leaf test cases in a list;

      store influencedTestCases for retesting;
}
```

Figure 3.18 Influence Graph RTS algorithm

3.11 TCP and Influence Graph RTS

TCP usually utilizes metrics or predictors that help in ordering test cases, since we already have a subset of test cases; any TCP technique can be applied to that subset.

An example would be measuring the time required for each test case to complete, and then give each test case a weight depending on its time of execution. The more time required running a test, the more weight it is given. Then we can order the resulting subset of test cases ascending by this metric and we have a subset with TCP applied to it.

Another metric would be to give extra weight to the more frequently executed test cases, so after RTS the subset is ordered descending and the most frequently executed test case is run first.

We can apply any of the TCPs mentioned in [22] and more importantly; this will be done apart from RTS. Unlike [23] where researchers mixed the two techniques to do their selection so no way to use one without using the other.

Chapter 4: Proposed Continuous Test Runner and IDE Integration

The second part of our contribution is designing a continuous test runner that will host the Influence Graph based RTS technique. As we've seen in chapter 2 not every RTS capable of being run continuously as all of them need to be rebuilt offline and with the help of a dynamic analysis phase.

As concluded in [22] the continuous test runner can save up to 90% of the wasted time during development caused by doing regression tests. They showed as the developer ignorance time decreases the regression time decreases as well and bugs are fixed faster and better.

The problem in [22] was that they do not do any RTS, instead they use some random TCP and they run all the test cases using that TCP technique. So I would be programming in module A and make a bug then go to module B and make another bug when the sequence reach module "**A**" test cases so I will be notified about its faults, I will return to module "**A**" and start fixing when the CT runs test cases for module "**B**" and notify about module "**B**" faults and this is so distractive. Additionally I will not be able to know the state of module "**A**" test cases until its test cases are tested again using the CT.

An RTS can take the place of a TCP, although the TCP refines the results of the RTS, but a TCP cannot do everything an RTS offers because usually doing TCP for all test cases is slow and will not fit a CT.

Also the CT in [22] is for Eclipse and wasn't updated since 2003 unlike the proposed approach which is platform/programming language independent. The design of the CT wasn't clear through their paper. Finally we tried to contact the authors to get the source code of their CT but unfortunately the code was missing and all the hosting repositories expired.

There are many commercial Continuous Test Runners like [59,60] which we cannot compare or study because they are closed source. There's an open source project called Infinitest [61] that unfortunately has no document on the scientific way of how it runs. The only provided document is a one page how to use document, anyway it doesn't seem to prioritize or select any of the test cases, instead they depend on a filtering system made manually by the developer.

Any Continuous Test Runner is required to do the following tasks:

1. Do tests continuously in the background.
2. Display test results to the developer.
3. Do not disturb or irritate the developer with the test results.

Also as the CT is working on the background; it should be clear that it needs to be so wise with the developer machine resources, and the more slow down caused by the CT the less the developer will keep this CT in action.

In Section 4.1 we discuss our continuous test runner new design.

4.1 Proposed Continuous Test Runner Design

The structure of the continuous test runner is shown in Figure 4.1.

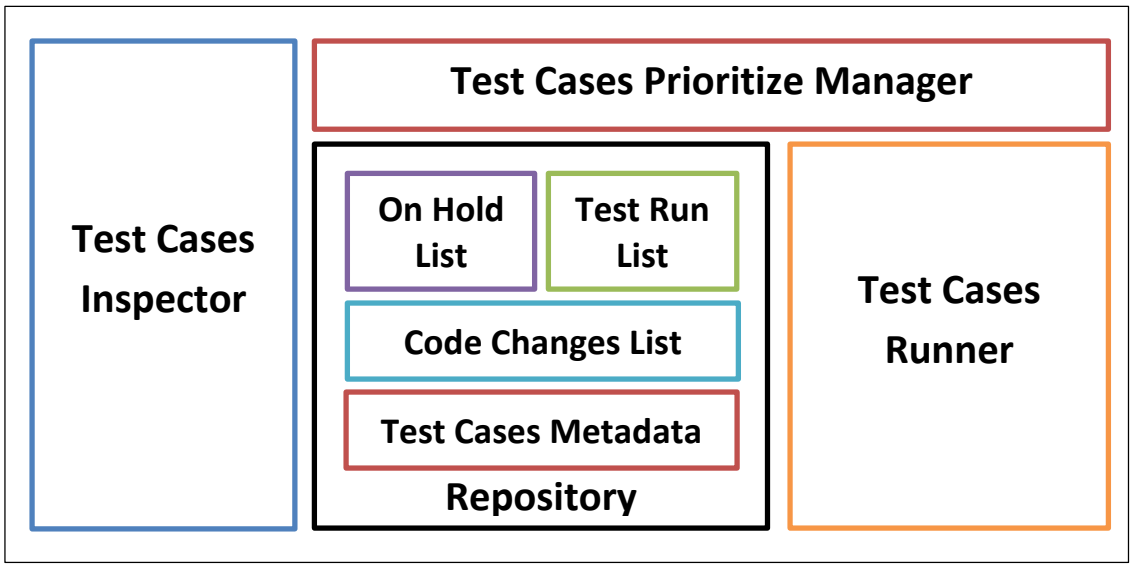

Figure 4.1 Proposed Continuous Test Runner Block Diagram

Here follows a brief description about every module:

1. **Test Cases Inspector:** the main module that makes the test runner a **continuous** one. It is in charge of specifying which test case to suspect and which to run.
2. **Repository:** the storage module that contains three lists; one for test cases on hold, another for test cases to run, and the last one for suspended code modifications.
3. **Test Cases Prioritize Manager:** this module is in charge of prioritizing test cases in the Test Run List inside the repository, it then gives these test cases to the **Test Cases Runner** module and gets feedback that helps in future prioritizations.
4. **Test Cases Runner:** this module is in charge of running test cases then posting notifications to the subscribed consumers.

In the following subsections each module will be discussed in separate.

4.1.1 Test Cases Inspector

Figure 4.2 shows a state diagram for the Test Cases Inspector.

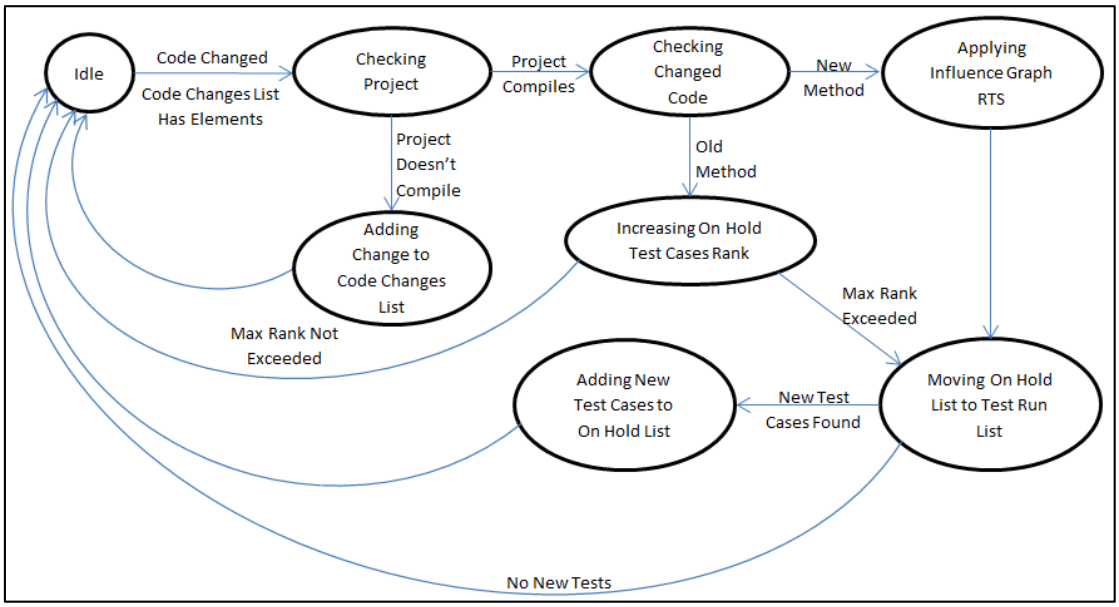

Figure 4.2 State diagram for the test cases inspector

As can be seen in Figure 4.2 the inspector begins in an idle state which means no code modifications exist. Code modification can be measured using various schemes:

1. When the developer leaves the current modified line.
2. When the developer compiles the current class.
3. When the developer closes the current file or move to another file.
4. Or any other event that the developer would do during development.

An optimized step is to check the method using the Influence RTS once, if all the changes are in the same method then the method test cases will be added only once for retesting.

During method modification, the corresponding test cases are put on hold. The ranks of these test cases increase. This is made to not to overwhelm the background process with multiple runs to the test cases and also not to distract the developer with false positives because he/she is still developing in the same method.

If the project has a compilation error due to a code change. the code changes are pushed into the code change list. If no compilation errors then the code is checked for

modifications. The inspector stays into the idle state in case of either no changes or compilation errors. And it gets out of this state by either new modifications or when code changes exist in the code changes list and the project compiles.

If the code compiles again; all the code changes are popped from the code changes list and all the corresponding test cases are gathered for retesting.

The ranks of the test cases increase also when the developer is still modifying in the same method. Too many modifications would increase the developer ignorance time with errors. So for projects with methods known to be large it is suggested to keep the maximum rank low.

It wouldn't matter if the code change is coming from actual code writing by the developer or the developer pressing CTRL + Z for undoing some modifications, because the IDE should provide the inspector with the change event and the changed code lines.

To make a customizable and powerful design; we put specific parameters for controlling the continuous test runner to optimize its operation depending on the developer needs and conditions. Table 4.1 shows parameters for the inspector.

Table 4.1 Test rank inspector customization parameters

Parameter	Description
TEST_CASE_MAX_RANK	Maximum rank for a test case to move from the On Hold List to the Test Run List.
MAX_CODE_CHANGES_LIST	Maximum size of the Code Changes List, after reaching the maximum queue, the inspector begins popping the oldest changes to keep size.
TEST_CASES_AUTO_UNHOLD	Being true; test cases are moved from on hold list automatically to test run list when the developer modifies another method other than the one that produced the on hold test cases. False value disables this option and so moving test cases will only depend on rank increase.

4.1.2 Repository

This module contains shared resources between various modules of the continuous test runner; it basically consists of three lists managed by the other modules.

The first list is called **On Hold List** and it contains test cases marked for retesting but not yet confirmed. Test cases being on hold have ranks to prevent starvation, with every modification the ranks of these test cases increase and when this rank reaches the TEST_CASE_MAX_RANK parameter; these test cases are moved from the **On Hold List** to the **Test Run List**. Also test cases could be moved from **Om Hold** to **Test Run** lists when the developer moves to modify another method, this option can be disabled by the developer using the parameter TEST_CASES_AUTO_UNHOLD so test cases would only be moved to **Test Run List** when their rank reaches a certain number.

The second list is the **Test Run List** and from its name it contains test cases that need to be run by the continuous test runner. This list is managed by the **Test Cases Prioritize Manager** which in turn clears the list after consuming the test cases.

An important thing to mention is that items of both lists also contain the method that was being modified, to be able to give the developer right directions if anything fails.

The third list is the **Code Changes List** and it is actually a queue that holds code modifications when the project compilation fails. During compilation failure making modifications to the influence graphs could corrupt them and so influence graphs recreation will be required. Instead, this queue will hold code changes to a MAX_CODE_CHANGES_LIST. If exceeded, the queue will start popping old changes. This would be suitable for some projects. However, dropping changes would affect the **inclusiveness** of the RTS technique and so would mess with the safety factor.

The last database is the **Test Cases Metadata**. It's a specific data gathered and stored by the various modules and its main function is to help the prioritize manager to be able to prioritize test cases for run.

Each test case will have a record with all the metadata parameters. Table 4.2 shows the currently existing parameters.

Table 4.2 Test cases metadata

Parameter	Description
EXECUTION_TIME	The time in milliseconds a test case needs to run.
TIMES_FAILED	The number of times a test case failed.
LAST_FAILURE_DATE	The last failure date for a test case.
LAST_RERUN_DATE	The last date a test case was run.

4.1.3 Test Cases Prioritize Manager

This module reads test cases that exist in the **Test Run List,** prioritize them, and finally send them to the **Test Cases Runner** module to be executed. The function of this module is shown in Figure 4.3.

Figure 4.3 Test cases runner function

Test Cases Prioritize Manager waits for MAX_TIME_TO_PRIORITIZE seconds to check for test cases to prioritize. If matches found, it begins prioritizing them based on the TCP technique selected in PRIORITIZATION_TECHNIQUE parameter. Any future efficient prioritization techniques can be added and used without any problem.

After test cases are fetched from the **Test Run List** the list is cleared and the timer of the prioritize manager is reset.

Table 4.3 summarizes parameters used to customize the Test Cases Prioritize Manager.

4.1.4 Test Cases Runner

The module that actually runs the test cases. It is described as a producer that runs test cases and posts the results of the test running to the consumers. It is typically designed as an observable [62] which offers a registration API to outside observers. The observers could be UI, statistical modules, or any other external module that needs to be notified when **Test Cases Runner** state changes.

The **Test Cases Runner** is designed to be multithreaded, which means it is capable of running multiple test cases at the same time. This makes running test cases a lot faster. However, running many test cases in the background can consume the processing power and so developer experience will be negatively affected. For that purpose a parameter for this module TEST_RUNNER_THREADS is tuned by the developer depending on the development environment. Especially with the recent regular processors which has up to 8 logical threads can dedicate 2 threads for the test runner without affecting the developer experience.

Table 4.3 Test cases prioritize manager customization parameters

Parameter	Description
MAX_TIME_TO_PRIORITIZE	Maximum time in seconds to fetch test cases for prioritization.
PRIORITIZATION_TECHNIQUE	The technique used in prioritizing test cases, the basic set contains the prioritization techniques mentioned in [22]: • **Suite Order** test are run in the order they appear in the test suite. • **Round Robin** same as the first one, but after every detected change, the round is restarted. • **Random** randomly selects test cases to be rerun. • **Recent Errors** tests that failed most recently are ordered first. • **Frequent Errors** tests with the greatest numbers of reruns are ordered first. • **Quickest Test** tests that take shorter time to execute are ordered first.

When test cases are run; many metrics are measured and stored in the **Test Cases Metadata**. For example the execution time of a test case is measured then stored in the EXECUTION_TIME parameter in order to be used by the **Test Cases Prioritize Manager** if a prioritize technique related to that metric was used.

Observers such as IDE and the repository can register in this module to get notifications when a test case finishes execution, the first would notify the developer if anything goes wrong in an appropriate way and the second can read the analytical metrics and store them for later use.

Table 4.4 summarizes parameters used to customize the Test Cases Runner.

Table 4.4 Test cases runner customization parameters

Parameter	Description
TEST_RUNNER_THREADS	The number of parallel threads that can run test cases.

4.2 IDE Integration

As we mentioned earlier information about classes and events regarding changes in source code are needed for our algorithms. And because IDEs differ in the type of information offered and the way this information is encapsulated; we designed a basic API of our requirements to be developed by any IDE without touching the original core of the technique.

This way the design stays IDE free and can be applied on any IDE easily and directly just by implementing the basic API.

In the following subsections we shall see a quick glance of each part of the required API.

4.2.1 Code Line Analyzer

Code line analyzer should offer the following functions:

1. To identify a given line, e.g. invocation, assignment, declaration, return, assert … etc.

   This is required to be able to identify special code lines, return and assert in our case, as well as other lines, for example invocations must be detected and analyzed as paths to more in depth influence graphs.
2. To give a list of the variables used in this line.

   This is required to be able to identify variables influence in influence graphs, also influence dependencies for lines is devised from contained variables.
3. To give a list of the affected lines by a specific line.

   This is required for influence analysis. It depends on the list of variables and their dependencies.
4. To give a list of lines depending on a specific variable.

   This is also required for influence analysis.
5. To give a list of invocations existing in a line.

   This is required for in-depth influence analysis.

4.2.2 Code Block Analyzer

Code block analyzer should offer the following functions:

1. To offer class analysis as Code Blocks nested inside Code Blocks and with Code Lines as leafs.

   This is required for the analysis process for prewritten files, and also for files added to influence when a change happens.
2. To identify a given block, e.g. for loop, if, else, try, catch, class, method… etc.

4.2.3 Line Change Event

An event should be triggered by the IDE. This event should provide old line and new line as parameters to the handler.

These events are handled by the update algorithm and the RTS algorithm as the CT will be triggered using these events.

Depending on the required change type an event should exist, in our example we used this event. But our CT can adapt to use any other event such as file saved, file closed, or file focus lost.

4.3 Experimental IDE – xIDE

As we needed an IDE for the experimentation, the available IDEs that have a development platform such as NetBeans, Eclipse… etc. need a learning period that we can consume in creating an experimental IDE for research purposes.

The proposed experimental IDE is a java based IDE intended to be published open-source for researchers and developers who need a simple IDE structure to experiment tools and new technologies. The experimental IDE (xIDE) is equipped with a conceptually new static analyzer for source code that draws class trees in a statement-level granularity. This tree identifies every code line with a specific type to provide extra features for developers and researchers to use.

4.3.1 xIDE Components and Events

The main graphical components of the xIDE are:

1. **Project Navigator** considers a folder as a project and classifies files inside the project showing their types in icons.

2. **Code Navigator** unlike the regular code navigators, our code navigator depends on the novel in-depth static analysis that shows a navigator to the level of code lines.
3. **Source Code Panels** when a file is opened, it is opened in a new tab in a source code panel with numbering to lines as well as syntax check.
4. **Status Bar** this bar displays state messages from the various components.

The logical components of the xIDE are:

1. **Static analyzer** the most important component which analyzes source code and builds class trees that is used by the influence graph algorithms.
2. **Unit testing** the IDE is equipped with jUnit for unit testing.
3. **Code compiler** the java compiler is used in this IDE and can be altered to another compiler if desired.
4. **Syntax checker** the java compiler is used for that purpose and can be altered.

And generally xIDE provides access to the following events:

1. **Line change event** when the developer finishes modifying a line an event with metadata containing old line and new line is triggered.
2. **File change event** when the developer opens a new file or an existing file an event with metadata containing old file and new file is triggered.
3. **Tab activated event** when the developer moves from an existing tab to another existing tab this event is triggered.
4. **Line left event** when the developer changes the place of the caret from a line to another this event is triggered with the old line number and the new line number.
5. **Line selected in code navigator event** when the developer selects a line in the code navigator this event is triggered with the selected line.
6. **File saved event** as the name refers an event when file is saved.
7. **File compiled event** as the name refers an event when file is compiled.

Researchers and platform developers have easy and full access over these events and components; also it is so easy to add new events depending on certain changes.

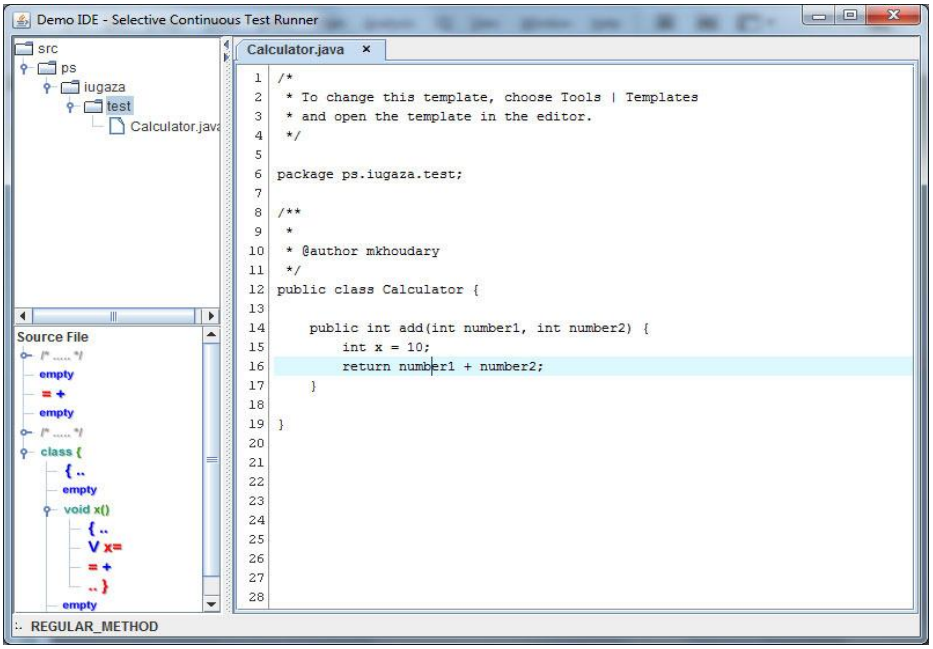

Figure 4.4 xIDE General Layout

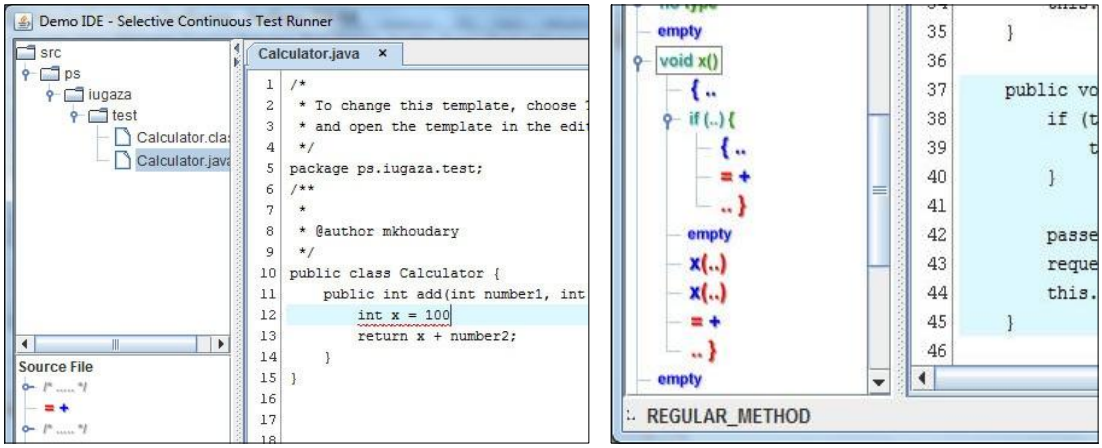

Figure 4.5 xIDE error marker and code analyzer

4.3.2 Static Analysis Module

As the design of this static analyzer is beyond the scope of this thesis we will briefly describe its functionality and concepts.

This module first builds the class tree using regular expressions to analyze code lines and blocks, then any modifications progressively updates this tree saving a lot of time when needing static analysis for class more frequently.

Regular static analysis algorithms like [63] need to rebuild the analysis each time to update it. In our approach it is only built once updated. This gives the ability to offer

researchers and platform developers a larger set of code change specific events like method added, method removed, method signature changed, class name changed… etc.

The class tree has the file as the root, blocks including classes, inner classes, methods, try blocks, catch blocks… etc. are parent nodes, and code lines are the leaves. Nodes contain data that can help in reaching them in their actual files anytime.

This static analysis as said depends on source code not byte code giving it the ability to adapt with any programming language such as Java, C#, C++ … etc. with limited changes. The analysis engine is built using regular expressions, modifying these expressions adapts the analyzer with other programming languages.

The regular expressions are used to analyze the code, identify the block/line type, and extracts data from these lines such as visibility, variable type, variable name, method name, method parameters, method signature… etc. A sample expression is shown in Figure 4.6.

```
\s*((public|private|protected)\s+)?(\w{1,})\s*\(\s*(.*)\)\s*(throws\s+((\w{1
,})(\s*,\s*)?)*)?\s*\{
```

Figure 4.6 Sample regular expression from code parser

This expression identifies constructor method signature and so identifies the beginning of a constructor method block, and from meanings of regular expressions [64] we can see that this regular expression describes a line that:

- Can start with an optional whitespace (spaces or tabs).
- Then optionally either public, private, or protected keywords.
- After wise a word with the method allowed characters.
- Then parentheses, and between them optionally parameters could be.
- an optional throws clause.
- and finally the block opening curly bracket.

This regular expression is well grouped so that each group can give the content without further analysis, for example group 8 gives the method name, group 9 gives the method parameters, group 2 gives the complete signature… etc.

And as the regular expressions get more complex writing them that way would be really painful and there will not be any maintainability in the code. Since we are going to open its source so for that purpose. A framework for regular expressions was built to build

these expressions, for example at compile time the expression illustrated in Figure 4.6 is written the way it looks in figure 4.7.

```
CONSTRUCTOR_METHOD(Pattern.compile(
            RegularExpressionsHelper.OPTIONAL_SPACE +
            RegularExpressionsHelper.ACCESSORS +
            RegularExpressionsHelper.CLASS_NAME +
            RegularExpressionsHelper.OPTIONAL_SPACE +
            RegularExpressionsHelper.METHOD_PARAMETERS +
            RegularExpressionsHelper.OPTIONAL_SPACE +
            RegularExpressionsHelper.intoOptionalBlock(
                RegularExpressionsHelper.THROWS_EXCEPTION) +
            RegularExpressionsHelper.OPTIONAL_SPACE +
            RegularExpressionsHelper.BLOCK_OPENING
            ))
```

Figure 4.7 Regular expression built using xIDE parser framework

As we can see this is a lot easier than the previous actual regular expression. Regular expressions are built and stored in memory once.

Regular analysis of source code can be processor consuming and needs a lot of code lines to match a one line expression. The above code, for example, would need nearly 25 – 40 lines of regular if/else and loops plus some temporary variables in addition to extra computation and optimization to source code. The power of regular expressions [65] appears in their superiority over regular analysis approaches.

As mentioned above static analysis is updated progressively as code changes by the developer, this is made through a module that binds itself to code change listeners and so any code change would be casted to the analysis. Having solo changes updated to the analysis minifies the time required to get an up to date static analysis of any class to either zero if loaded into memory or the time of loading analysis from disk.

4.3.3 Integrity with Influence Graph RTS and the Proposed CT

The API required by the influence graph RTS is implemented experimentally on the xIDE as well as the continuous test runner. In the next chapter we shall see the experimental study of the influence graph RTS and proposed CT over the xIDE.

Chapter 5: Experimentation and System Evaluation

5.1 Difficulties of Comparison

In [23] the authors proved the efficiency of their methodologies by evaluating the predicators they've made. It was very hard to compare our work to their work using the same comparison methodology as we do not have these predicates.

Comparing our work to the other approaches such as [24] and [58] is very hard as well because of the difference in concept; we are the first online RTS technique and because the mentioned previous approaches didn't put running the RTS technique online into their considerations so there was no point in comparing our experiments to theirs.

For that purpose we saw that evaluating the RTS features mentioned in Section 2.2.1 will evaluate the quality and effectiveness of our approach, and in the following experiments we are going to evaluate these features.

5.2 Log4J

For the purpose of experiments evaluation we chose Log4J from apache [66] as a test project. Log4J was used in [23] for a close evaluation.

Log4J is a library that implements the logging standard in Java; it consists of approximately 45,000 lines of code in 161 classes and 90 test classes. As we mentioned earlier this technique is intended to be progressive, meaning it should be used with a development technique like TDD so there will not be an overhead of preparing old files to be progressive. For Log4J; the library had to be prepared for usage with our influence graph by building class graphs and influence graphs to enable progressive update, the preparation process took 2420 milliseconds or 2.4 seconds to complete.

The 2.4 seconds overhead for such a project with 45,000 lines of code is a one-time overhead, meaning next time the project is opened there will not be any overhead as the graphs and the class analysis are already built.

And even if we didn't use storage; the time needed to build class analysis and influence graphs isn't considered an overhead as this is needed only once during the development session and usually it is done when the IDE starts.

5.3 Experiment 1

The goal of this experiment is to prove the influence graph **performance aspects** in the following features:

1. If the modification doesn't affect the return value of the method there will be no modifications on the influence graph, unlike [22, 23, 24] where any modification is considered as a modification for retesting.
2. When the modification occurs the graph will not be completely rebuilt like in [22, 23, 24]; instead only the affected nodes will be changed and any new nodes will be added.

Given method in Figure 5.1; this method is tested by the test case shown in Figure 5.2.

```
26 public String format(final LoggingEvent event) {
27     String something = "Mock";
28     String otherThing = "Mock2";
29     return something;
30 }
```

Figure 5.1 The method format used in experiment 1

```
116 public void testFormat() throws Exception {
117     Logger logger = Logger.getLogger(
                                  "org.apache.log4j.LayoutTest");
118     LoggingEvent event = new LoggingEvent(
                                        "org.apache.log4j.Logger",
                                        logger, Level.INFO,
                                        "Hello, World", null);
119
120     MockLayout layout = new MockLayout();
121
122     String result = layout.format(event);
123     assertEquals("Mock", result);
124 }
```

Figure 5.2 The test case testFormat that tests the method shown in Figure 5.1

Figure 5.3 shows the generated influence graph. Each influence graph is depicted with a distinct color. LayoutTest.java, for example, has a white background while MockLayout.java has a yellow one, also the assert node in LayoutTest.java line 123 is considered the root node of influence. It's also clear from the graph that any change in the visible lines will lead to a notification about a change in the test cases that contains the assert node in line 123, testFormat in this example.

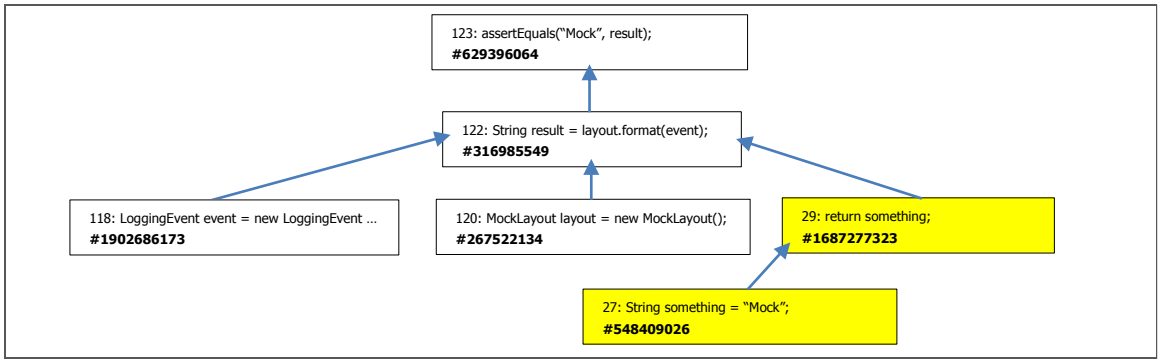

Figure 5.3 testFormat() generated influence graph

The numbers prefixed by a hash sign below each node represent the hash code of the influence node objects and they're put in purpose of showing that when a change occurs the graph isn't rebuilt but instead is updated only with the modifications. Let's start our experiment by modifying the code of the LayoutTest.java as shown in Figure 5.4

```
116 public void testFormat() throws Exception {
117     Logger logger = Logger.getLogger(
                                  "org.apache.log4j.LayoutTest");
118     LoggingEvent event = new LoggingEvent(
                                        "org.apache.log4j.Logger",
                                        logger, Level.INFO,
                                        "Hello, World", null);
119
120     MockLayout layout = new MockLayout();
121     String temp = "X";
122
123     String result = layout.format(event);
124     assertEquals("Mock", result);
125 }
```
Figure 5.4 testFormat() fragment after adding two uneffecting lines

Going back to check the influence graph; the graph in Figure 5.5 was generated. As we can see although lines 123 was changed to 124 the hash code of the node wasn't changed and that means none of the objects in the graph were replaced by other objects, also no test cases were introduced for retesting as the modifications do not interfere with the final result.

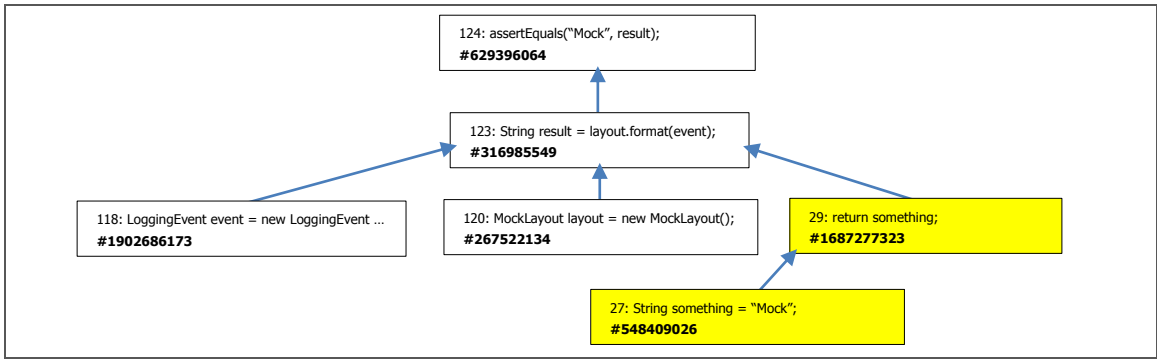

Figure 5.5 irrelevant modifications effect on testFormat() influence graph

Now let's link the temp variable to the result variable by performing the modifications shown in Figure 5.6.

```
116 public void testFormat() throws Exception {
117     Logger logger = Logger.getLogger(
                                    "org.apache.log4j.LayoutTest");
118     LoggingEvent event = new LoggingEvent(
                                        "org.apache.log4j.Logger",
                                        logger, Level.INFO,
                                        "Hello, World", null);
119
120     MockLayout layout = new MockLayout();
121     String temp = "X";
122
123     String result = layout.format(event) + temp;
124     assertEquals("Mock", result);
125 }
```

Figure 5.6 Modifications on testFormat() that links temp to the return value

Again we can see in Figure 5.7 that all the hash codes for the nodes are the same, in addition; the node String temp = "X"; was added to the influence graph as it now influences code line 123 that influences the assert node in line 124. Also test case testFormat was marked for retesting as this modification affects the overall result that the assert node tests.

Now let's the modifications shown in Figure 5.8 to the MockLayout.java to see the changes that could happen to the influence graph.

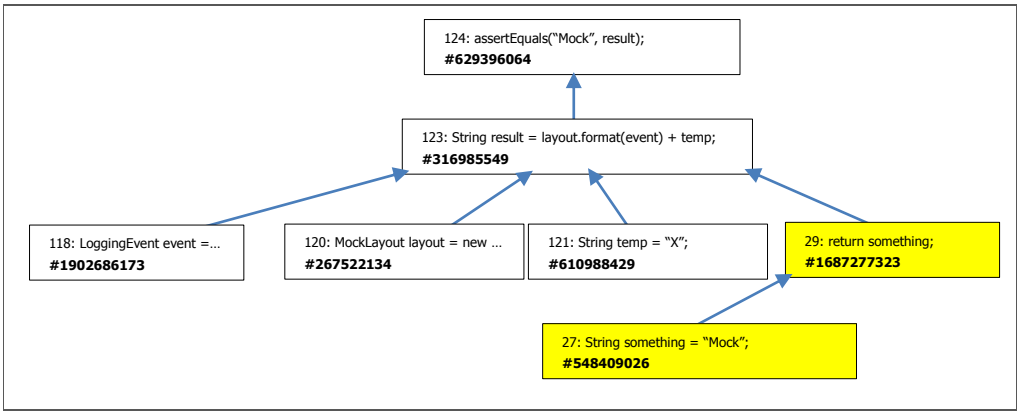

Figure 5.7 relevant modifications effect on testFormat() influence graph

```
26  public String format(final LoggingEvent event) {
27      String something = "Mock";
28      String otherThing = "Mock2234";
29      return something;
30  }
```

Figure 5.8 The body of format() method after irrelevant modifications

Nothing was changed after adding **234** to the **otherThing** as this variable doesn't participate in the return value of the method, but when doing the modifications in Figure 5.9 a change will happen, the change is shown in Figure 5.10.

```
26  public String format(final LoggingEvent event) {
27      String something = "Mock";
28      String otherThing = "Mock2234";
29      return something + otherThing;
30  }
```

Figure 5.9 The body of format() method after relevant modifications

As we can see MockLayout.java line 28 was added to the graph, so now **testFormat** as well as **testLineSepLen** are marked for retesting.

The outcomes of this simple experiment are:

1. Influence graphs are updated progressively along with the code modifications.
2. Influence graphs nodes are reused as much as possible to reduce memory usage and CPU power.
3. As a natural result all relevant test cases were marked for retesting.

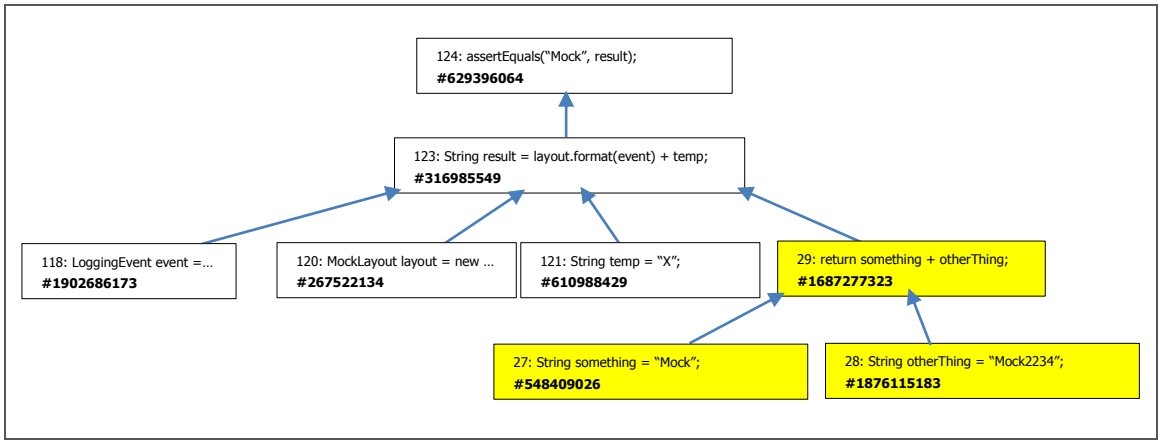

Figure 5.10 testFormat() influence graph after adding otherThing variable

5.4 Experiment 2

The goal of this experiment is to prove the influence graph efficiency aspects as follows:

1. Influence graphs for log4J test cases will be built.
2. An automated code mutator will invoke modifications in all the log4J library methods.
3. Selected test cases after modifications will be recorded.
4. The efficiency will be measured by watching how the approach selects all the relevant test cases.

5.4.1 Experiment Automation

Because tracking each method and test cases is very time consuming and to achieve the results faster we created a quick code mutator that makes mutations to methods code that can fire retests if exist.

Also to detect methods usage in test cases in another way than the Influence Graph RTS we designed a simple natural language processor that tries to Figure out methods usage in test cases by seeking for methods being called in test cases.

The code mutator will be run on all project methods and we will observe the results in the next sub section.

The used mutation technique is so simple:

1. The mutator seeks for methods with influence graphs excluding test cases.

2. Then the mutator selects a node that influences the return value directly or indirectly and fires a change in that node. The change is an addition to the expression if it is an expression, a concatenation if it is a String.
3. Finally the mutator leaves the rest to the RTS which propagates the change event and returns the test cases to be rerun.

5.4.2 Experiment Results

The time needed to build influence graphs for all test cases was 160 milliseconds for 89 test classes, we shall put in mind that this is one-time latency; this marks the performance of the Influence Graph creation algorithm.

The process of automatic mutation associated with executing both Influence Graph Enhancement algorithm and Influence Graph RTS took 690 milliseconds for mutating nearly 600 methods. The observed list is shown in table 5.1.

Table 5.1 RTS results on mutation

Class Method	No of retests marked by our technique	No of test cases detected by the natural language processor
DateLayout.getDateFormat()	2	2
DateLayout.getTimeZone()	2	2
RollingCalendar.getDatePattern()	1	0
RollingCalendar.computeCheckPeriod()	2	0
HTMLLayout.getLocationInfo()	1	3
HTMLLayout.getTitle()	1	3
Compare.compare(Class testClass, String file1, String file2, BufferedReader in1, BufferedReader in2)	18	0
Logger.getRootLogger()	7	52
Logger.getLogger(String name, LoggerFactory factory)	4	95
Logger.isTraceEnabled()	1	2
MockLayout.format(final LoggingEvent event)	3	3
TTCCLayout.getThreadPrinting()	1	3
TTCCLayout.getCategoryPrefixing()	1	3
TTCCLayout.getContextPrinting()	1	3
TTCCLayout.format(LoggingEvent event)	1	1
RollingFileAppender.getMaxBackupIndex()	2	2
RollingFileAppender.getMaximumFileSize()	1	1
Hierarchy.getLogger(String name, LoggerFactory factory)	1	3
VectorErrorHandler.getMessage(final int index)	2	2
VectorErrorHandler.size()	3	3
AppenderAttachableImpl.getAppender(String name)	2	0

LoggingEvent.getLocationInformation()	1	2
LoggingEvent.getLevel()	9	10
LoggingEvent.getLoggerName()	4	5
LoggingEvent.getMessage()	8	12
LoggingEvent.getStartTime()	1	0
LoggingEvent.getThreadName()	4	4
VectorAppender.getVector()	4	6
VectorAppender.isClosed()	2	2
PatternAbbreviator.getAbbreviator(final String pattern)	1	0
PatternAbbreviator.getDefaultAbbreviator()	1	0
FileAppender.getThreshold()	1	1
FileAppender.isAsSevereAsThreshold(Priority newPriority)	1	1
Category.getAllAppenders()	6	0
Category.getAppender(String name)	2	0
Category.getEffectiveLevel()	1	0
Category.getName()	2	0
Category.getLevel()	1	0
Category.getResourceBundle()	3	0
LevelRangeFilter.getLevelMin()	1	1
XMLLayout.getLocationInfo()	1	3
CachedDateFormat.findMillisecondStart(final long time, final String formatted, final DateFormat formatter)	5	0
CachedDateFormat.getMaximumCacheValidity(final String pattern)	1	0
UtilLoggingLevel.toLevel(final String sArg, final Level defaultLevel)	1	1
PatternLayout.getConversionPattern()	1	1
PatternLayout.format(LoggingEvent event)	2	8
MDC.get(String key)	1	0
MDC.getContext()	1	0
MDC.get0(String key)	1	0
MDC.getContext0()	1	0
Priority.toString()	12	3
Priority.toPriority(String sArg, Priority defaultPriority)	6	2
BoundedFIFO.get()	5	14
BoundedFIFO.getMaxSize()	1	1
BoundedFIFO.isFull()	4	0
BoundedFIFO.length()	4	14
BoundedFIFO.wasEmpty()	4	0
LogCapture.getMessage()	36	12
NullEnumeration.getInstance()	6	0
SerializationTestHelper.serializeClone(final Object obj)	1	0
SerializationTestHelper.deserializeStream(final String witness)	4	0
HUPNode.getPort()	1	0

EnhancedPatternLayout.getConversionPattern()	1	1
EnhancedPatternLayout.format(final LoggingEvent event)	1	3
LogManager.getLoggerRepository()	1	0
LogManager.getRootLogger()	7	0
LogManager.getLogger(final String name, final LoggerFactory factory)	4	0
CyclicBuffer.getMaxSize()	2	2
CyclicBuffer.get()	3	12
CyclicBuffer.length()	1	7
Level.toLevel(String sArg, Level defaultLevel)	15	10
RendererMap.get(Class clazz)	10	11
RendererMap.getDefaultRenderer()	1	1
SMTPAppender.getEvaluator()	1	1

Before analyzing the resulted table we shall show another advantage to our mutation tool associated with Influence Graph RTS is when it is run in reverse, to detect methods that didn't affect any test case. So developers and testing engineers can have a look in a very short time at logic that's not covered with test cases. This gives a look at how comprehensive their test suite is.

Now let's go back to the results table, as we can observe:

1. 20 methods out of the 74 methods, marked in green, matched exactly the number of test cases discovered by natural language processing.
2. 29 methods out of the 74 methods, marked in yellow, didn't match any test case in the test suite because of either a failure in the natural language processor or, more importantly, because these methods were used indirectly by the test case. The only way to discover this other than using Influence Graph techniques is to do dynamic analysis like in [23].
3. 3 methods out of the 74 methods, marked in blue, had test cases resulting from Influence Graph RTS more than the ones discovered by the natural language processor for the same reasons mentioned in point 2.
4. Finally 22 methods out of the 74 methods, marked in red, showed more test cases with natural language processor than the ones discovered by the Influence Graph RTS. To explain this result we will do manual observation on the 22 methods in the next subsection.

5.4.3 Manual Observation Results for a Subset of Methods

As me mentioned in the previous section 23 methods had more natural language processor matches on test cases than test cases discovered by Influence Graph RTS. This short list is easier to test and observe manually than the whole set of 74 methods. The 23 methods are shown in table 5.2.

Table 5.2 RTS results 4th category manual observation

Class Method	No of retests marked by our technique	No of test cases detected by the natural language processor	No of retests that should be done
HTMLLayout.getLocationInfo()	1	3	1
HTMLLayout.getTitle()	1	3	1
Logger.getRootLogger()	7	52	7
Logger.getLogger(String name, LoggerFactory factory)	4	95	4
Logger.isTraceEnabled()	1	2	1
TTCCLayout.getThreadPrinting()	1	3	1
TTCCLayout.getCategoryPrefixing()	1	3	1
TTCCLayout.getContextPrinting()	1	3	1
Hierarchy.getLogger(String name, LoggerFactory factory)	1	3	1
LoggingEvent.getLocationInformation()	1	2	1
LoggingEvent.getLevel()	9	10	9
LoggingEvent.getLoggerName()	4	5	4
LoggingEvent.getMessage()	8	12	8
LoggingEvent.getRenderedMessage()	1	2	1
VectorAppender.getVector()	4	6	4
XMLLayout.getLocationInfo()	1	3	1
PatternLayout.format(LoggingEvent event)	2	8	2
BoundedFIFO.get()	5	14	5
BoundedFIFO.length()	4	14	4
EnhancedPatternLayout.format(final LoggingEvent event)	1	3	1
CyclicBuffer.get()	3	12	3
CyclicBuffer.length()	1	7	1
RendererMap.get(Class clazz)	10	11	10

As we manually examined the 23 methods we had the following observations:

1. 17 out of the 23 methods, marked in green, had a slight difference than the influence graph approach but when tested manually matched our approach

exactly, the difference came from overloading as the natural language processor couldn't differ between overloaded methods.

2. The test of the methods, marked in yellow, our approach matched the manual observation, and the difference from the natural language processor came from having these methods used within the context of a test case but without affecting the final result that's tested using assert statements.

From this experiment we can see that our approach preserves inclusiveness and test cases which are meant to be retested are retested.

We also saw that making modifications on nearly 600 methods required only 690 milliseconds of enhancement to influence graphs, and as our target is directed to online testing a developer can really work on one method at a time unless he/she is making automatic refactoring then the shown number of millisecond will not be an overhead.

5.5 Pilot Study

Approaches in software engineering that depends on human interaction are very hard to model mathematically and also relying on a mathematical proof is pointless when the approach to be proved depends heavily on human interactions.

A pilot study is a research project that is conducted on a limited scale that allows researchers to get a clearer idea of what they want to know and how they can best find it out without the expense and effort of a full-fledged study [67].

Other researchers used this methodology in their papers such as [68]; the researchers made a pilot study on a class of students and collected their data from students through that class, also other studies such as [68, 69].

5.5.1 Aims

To evaluate the efficiency of the use of influence graph approach in a CT when used in a real programming project.

5.5.2 Study Design

We programmed part of the student financial system at the Gaza Islamic University and associated it with a group of test cases chosen carefully to reflect nested errors when the code is modified. The simple project consists of 16 classes and 8 test cases.

We also prepared a set of 4 well-studied functional modifications that can make a variation of errors depending on the developer experience.

To assess the efficacy of our method 20 developers were recruited with various levels of experience and split into two groups (10 per each group). Group 1 applied the Legacy Test Last Method (did the modifications regularly and ran the test suite at last) while Group 2 applied our method CT with Influence Graph RTS. Unit testing was introduced to both groups, as well as description about the project code and logic.

Table 5.2 shows the distribution of the developer's experience.

Table 5.3 Distribution of the developers' experience

Experience Level	Developers Count per Group
3+ Years of experience in Development	4
Fresh Graduates	3
Under Graduates	3
Total per Group	10

5.5.3 Study Measures

For each participant two parameters were recorded:

1. Development Time (in second): reflects the time the developer took to accomplish the development task.
2. Regression Time (in second): reflects the time the developer took to fix the errors resulted from the development made, and as [22] proved

As the development (ignorance) time increases the regression time increases as well. The main outcome for this pilot study was the regression time.

5.5.4 Sample Size and Power Calculation

Sample size determination was based on power calculation using two-sided two-sample t-test. This test requires the outcome of interest to be normally distributed. The distribution of regression time was not normal and therefore we had to apply log-transformation to achieve normality.

Using PASS 2008, sample sizes of 10 per group were found to achieve 100% power to detect a minimum difference of 1.0 for log-transformed regression time between the two groups with a significance level (alpha) of 0.05 using a two-sided two-sample t-test.

5.5.5 Results

Our results prove that using our method significantly reduced the regression time compared to using the conventional method as shown in table 5.3. The median regression time among group 2 participants, who applied our method, was significantly lower than the median regression time among group 1 participants (485.0 second vs. 1189.0 second, P value=0.03).

Table 5.4 Summary Statistics for Dev. and Reg. Times by Study Group

	Total N=20	**Developers Group 1 Legacy Test Last Method N=10**	**Developers Group 2 CT with Influence Graph RTS N=10**
Development Time, S,	650.5 (358.0, 712.0)	674.5 (372.0, 713.0)	431.0 (351.0, 681.0)
Regression Time, S	1092.0 (485.0, 1278.5)	**1189.0 (912.0, 1417.0)***	**485.0 (446.0, 1155.0)***

Median (25^{th}p, 75^{th} p) was presented.

* Rank Sums Test P value =0.03

Interestingly, irrespective of experience level, all participants of group 2, which applied our method, reported lower regression time in comparison to participants from the other group (Figure 5.11).

Adjusting for experience level, participants who applied our method (group 2) independently associated with less regression time compared to participants who applied the conventional method (table 5.4).

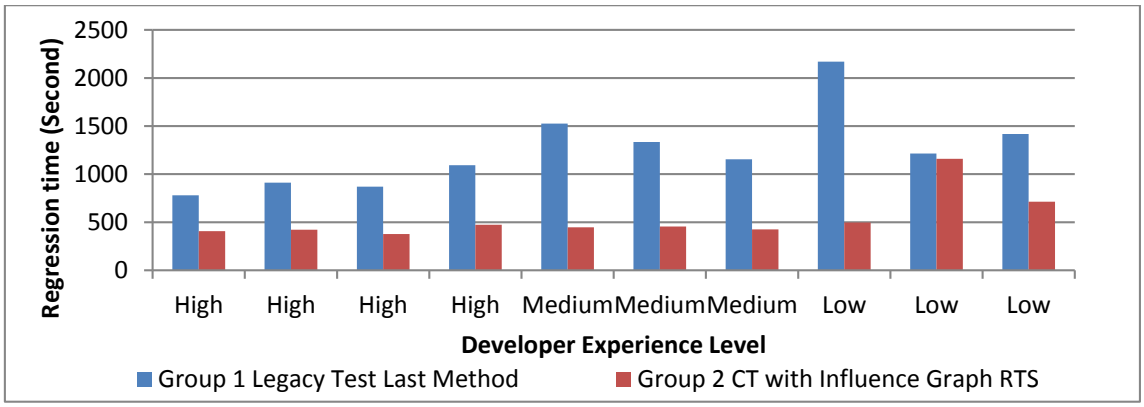

Figure 5.11 Regression Time for Study Participants by Level of Experience

Table 5.5 Group Effect on Dev.* and Reg.* Times Adjusting for Experience Level

Variables	Development Time* β	Development Time* SE	P value
Group 1	Ref		0.1
Group 2	-0.21	0.12	
Level			
Low	**0.75**	**0.15**	**0.0001**
Medium	**0.57**	**0.15**	**0.006**
High	**Ref**		
	Regression Time*		
Group 1	**Ref**		**0.003**
Group 2	**-0.58**	**0.17**	
Level			
Low	**0.54**	**0.20**	**0.02**
Medium	**0.25**	**0.20**	0.2
High	**Ref**	**Ref**	

* Log transformed

5.5.6 Discussion:

- Development times in the two groups are nearly matching between the two groups.
- The median of the time taken for regression in the first group was 1189 seconds while the median of the time taken for regression for the second group was 485 seconds.
- The average shows great enhancement of the RTS technique over the other technique.
- The regression time for developers in the first group was increasing as the development time for the developer increases and that's natural and was stated and proved in [22] that as the developer ignorance time increases and the time needed to fix errors increases as well.

- The median of the regression time for developers in the second group was around 485 seconds and we saw that it's not affected that much with the development time, that's because the CT gives the developers online comments about what parts of test suite that has errors and what are the errors.
- From our experiment we can conclude that our CT associated with the Influence Graph RTS helped in reducing the wasted time developers take to fix their code after making modifications.
- We used different developers for the two groups because doing the modifications once makes it easier and quicker to do them for another time and that'd have biased our results as the second experiment will always have better results.

Chapter 6: Conclusion and Future Work

This chapter presents the conclusion from this thesis. In Section 6.1, we provide a summary of the thesis. Future works are proposed in Section 6.2.

6.1 Conclusion

Test Case Prioritization (TCP) and Regression Test Selection (RTS) are two techniques used to reduce and prioritize test cases existing in a test suite to be able to advise the developer with the test cases that should be retested relative to the context he/she is currently working on.

Many TCP and RTS approaches were proposed but none of them worked online during development time. Instead, many of them depend on dynamic analysis and offline database rebuilding to provide a database that can be searched later during development time for relevant test cases.

The older methods might work at first, but the longer the development goes on without refreshing the databases the more deviation and misses the TCP and RTS will produce.

Additionally with the heavy processing needed for the older proposed techniques, it is impractical to fit these techniques with a Continuous Test Runner (CT) because of their demanding to various phases including dynamic program analysis.

In this research we presented a progressive RTS technique that is able to do static analysis on the source code during development time and build/maintain graphs we named influence graphs that are able to provide the developer with instant answers about relevant test cases at the relevant context.

Our technique guaranteed inclusiveness as it actually has the data needed to provide a correct answer so no inference or prediction, also guaranteed efficiency and precision due to the simplicity of the algorithms and their progressive design. Finally generality is guaranteed in our design as we do not depend on any platform specific features or characteristics.

We have increased the effectiveness of our Influence Graph Based RTS by providing a design for a CT that's flexible and tunable depending upon developer and project needs. Our CT can use any RTS technique and can employ any TCP technique as well.

For our CT to be highly customizable we presented a sit of tunable configurations that allow developers to control the behavior of the CT and additionally to decrease the confusion that might arise when using this tool, so depending on the development style the developer can control the behavior of the proposed CT.

The speed of our algorithms has been shown to be very fast when running from the scratch on a huge library called Log4J and not putting progressive feature into consideration; progressive feature can make this fast interaction even faster.

We evaluated our work through several experiments some of them were triggered using a simple code mutator on the library Log4J and the others on an experimental project that was developed for the thesis use and was tested over 20 real developers with various levels of skill.

6.2 Future Work

The following enhancements can be made in the future:

- Upgrading influence graph to cover exceptions in modification detection process.
- Using lexicographical comparison between an original code line N and a modified original code line N' if no change occurs then no need to propagate retesting. Example *int x = y + z;* changed to *int x = z + y;* nothing should happen.
- Providing more suitable solutions for the dynamic dispatch problem.
- Upgrading and using Influence Graph Concept in Test Coverage and so discover areas with no tests at all.
- Providing the proposed CT with more TCP techniques.
- Creating plugins for various IDEs like Netbeans, Eclipse… etc that employs the proposed CT.
- Doing an extended empirical study about the feasibility of the Influence Graph based RTS with CT.

Bibliography

[1] Kent Beck, *Test Driven Development: By Example*. United States of America: Addison-Wesley, 2002.

[2] Glenford J. Myers, *The Art of Software Testing*. New York, United States of America: John Wiley & Sons, 1979.

[3] Cem Kaner, "Exploratory Testing," in *Quality Assurance Institute Worldwide Annual Software Testing Conference*, Orlando, FL, November 2006.

[4] Paul Ammann and Jeff Offutt, *Introduction to Software Testing*. Cambridge, United States of America: Cambridge University Press, 2008.

[5] Roy Osherove, *The Art of Unit Testing*. United States of America: Manning Publications Co., 2009.

[6] Paul Hamill, *Unit Test Frameworks*, 1st ed., Mike Hendrickson, Ed. United States of America: O'REILLY, 2004.

[7] Kent Beck, *Extreme Programming Explained*. United States of America: Addison-Wesley, 1999.

[8] (2005, November) Wikipedia. [Online]. http://en.wikipedia.org/wiki/Test-driven_development#cite_note-Beck-

[9] Jacob Proffitt. (2008, January) The Runtime. [Online]. http://theruntime.com/blogs/jacob/archive/2008/01/22/tdd-proven-effective-or-is-it.aspx

[10] Noel Llopis. (2005, February) Stepping Through the Looking Glass: Test-Driven Game Development (Part 1). [Online]. http://gamesfromwithin.com/stepping-through-the-looking-glass-test-driven-game-development-part-1

[11] Maria Siniaalto, "A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage ," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, Oulu, 2007, pp. 275-284.

[12] Matthias M. Müller and Frank Padberg, "About the Return on Investment of Test-Driven Development," in *International Workshop on Economics-Driven Software Engineering Research EDSER-5*, Washington, DC, 2003, pp. 26-31.

[13] Steve Loughran. (2006, November) Testing. [Online]. http://people.apache.org/~stevel/slides/testing.pdf

[14] Gerard Meszaros. (2011, February) Fragile Test. [Online]. http://xunitpatterns.com/Fragile%20Test.html

[15] Laurie A. Williams, "Test-driven development as a defect-reduction practice," in *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, North Carolina, 2003, pp. 34-45.

[16] John Huan Vu, "Evaluating Test-Driven Development in an Industry-Sponsored Capstone Project," in *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, California, 2009, pp. 229-234.

[17] Adam M. Geras, Michael R. Smith, and James Miller, "A prototype empirical evaluation of test driven development," in *Software Metrics, 2004. Proceedings. 10th International Symposium on*, Alta., 2004, pp. 405-416.

[18] Nuno Laranjeiro, "Extending Test-Driven Development for Robust Web Services," in *Dependability, 2009. DEPEND '09. Second International Conference on*, Athens, 2009, pp. 122-127.

[19] Theodore D. Hellmann, "Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, Calgary, 2010, pp. 444-447.

[20] Taha Karamat, "Reducing Test Cost and Improving Documentation In TDD (Test Driven Development)," in *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2006. SNPD 2006. Seventh ACIS International Conference on*, Abbottabad, 2006, pp. 73-76.

[21] Sebastian M. Wieczorek, Alin Stefanescu, Mathias Fritzsche, and Joachim Schnitter, "Enhancing Test Driven Development with Model Based Testing and Performance Analysis," in *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic & Industrial Conference*, Darmstadt, 2008, pp. 82-86.

[22] David Saff and Michael D. Ernst, "Reducing wasted development time via continuous testing," in *Fourteenth International Symposium on Software Reliability Engineering*, Cambridge, 2003, pp. 281-292.

[23] Hagai Cibulski and Amiram Yehudai, "Regression Test Selection Techniques for Test-Driven Development," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, Tel-Aviv, 2011, pp. 115-124.

[24] Yih-Farn R. Chen, David S. Rosenblum, and Kiem-Phong P. Vo, "TestTube: A System for Selective Regression Testing," in *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, Murray Hill, 1994, pp. 211-220.

[25] Amitabh Srivastava and Jay Thiagarajan, "Effectively Prioritizing Tests in Development Environment," Microsoft Research, Redmond, TechReport MSR-TR-2002-15, 2002.

[26] Gregg Rothermel and Mary Jean Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 2, pp. 173-210, April 1997.

[27] Thomas Ball, "On the limit of control flow analysis for regression test selection," in *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis* , New York, 1998, pp. 134-142.

[28] David Binkley, "Semantics guided regression test cost reduction," *Software Engineering, IEEE Transactions on*, vol. 23, no. 8, pp. 498-516, August 1997.

[29] Filippos I Vokolos and Phyllis G Frankl, "Pythia: A regression test selection tool based on textual differencing," in *3rd internatinal conference on on Reliability, quality and safety of software-intensive systems*, London, 1997, pp. 3-21.

[30] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel, "Prioritizing test cases for regression testing," in *The 2000 ACM SIGSOFT international symposium on Software testing and analysis* , New York, 2000, pp. 102-112.

[31] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929-948, October 2001.

[32] Peter Henderson and Mark Weiser, "Continuous execution: the VisiProg environment," in *The 8th international conference on Software engineering*, Los Alamitos, 1985, pp. 68-74.

[33] Lisa Crispin and Janet Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*. New York, United States of America: Addison-Wesley, 2009.

[34] Maria Siniaalto and Pekka Abrahamsson, "A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage," in *The First International Symposium on Empirical Software Engineering and Measurement* , Washington, DC, 2007, pp. 275-284.

[35] K. Beck, "Aim, fire," *Software, IEEE*, vol. 18, no. 5, pp. 87-89 , Sep/Oct 2001.

[36] David Astels, *Test-Driven Development: A Practical Guide: A Practical Guide*, Kathleen M. Caren, Ed. New Jersey, United States of America: Prentice Hall PTR, 2003.

[37] Thirumalesh Bhat and Nachiappan Nagappan, "Evaluating the efficacy of test-driven development: industrial case studies," in *The 2006 ACM/IEEE international symposium on Empirical software engineering* , New York, 2006 , pp. 356-363.

[38] Kim Man Lui and Keith C.C. Chan, "Test driven development and software process improvement in China," in *Extreme Programming and Agile Processes in Software Engineering 5th International Conference XP 2004 Proceedings Lecture Notes in Comput Sci*, vol. 3092, Garmisch-Partenkirchen, 2004, pp. 219-222.

[39] Laurie Williams, E. Michael Maximilien, and Mladen Vouk, "Test-Driven Development as a Defect-Reduction Practice," in *The 14th International Symposium on Software Reliability Engineering* , Washington, DC, 2003, p. 34.

[40] Gerardo Canfora, Aniello Cimitile, Felix Garcia, Mario Piattini, and Corrado Aaron Visaggio, "Evaluating advantages of test driven development: a controlled experiment with professionals," in *ISESE '06 Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, New York, 2006 , pp. 364-371.

[41] Matthias M. Müller, "The effect of test-driven development on program code," in *XP'06 Proceedings of the 7th international conference on Extreme Programming and Agile Processes in Software Engineering*, Berlin, 2006, pp. 94-103.

[42] Boby George and Laurie Williams, "A structured experiment of test-driven development," *Information and Software Technology*, vol. 42, no. 5, pp. 337–342, April 2004.

[43] A. Geras, M. Smith, and J. Miller, "A Prototype Empirical Evaluation of Test Driven Development," in *METRICS '04 Proceedings of the Software Metrics, 10th International Symposium* , Washington, DC, 2004, pp. 405-416.

[44] Juho Jäälinoja, Pekka Abrahamsson, and Antti Hanhineva, "Improving Business Agility Through Technical Solutions: A Case Study on Test-Driven Development in Mobile Software Development," *Business Agility and Information Technology Diffusion*, vol. 180, no. 1, pp. 227-243, 2005.

[45] David S. Janzen and Hossein Saiedian, "On the Influence of Test-Driven Development on Software Design," in *CSEET '06 Proceedings of the 19th Conference on Software Engineering Education & Training* , Washington, DC, 2006, pp. 141-148.

[46] Reid Kaufmann and David Janzen, "Implications of test-driven development: a pilot study," in *OOPSLA '03 Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, 2003, pp. 298-299.

[47] Matthias M. Müller and Oliver Hagner, "Experiment about test-first programming," *Software, IEE Proceedings*, vol. 149, no. 5, pp. 131-136 , October 2002.

[48] Matjaž Pančur, Mojca Ciglarič, Matej Trampuš, and Tone Vidmar, "Towards empirical evaluation of test-driven development in a university environment," in *EUROCON 2003. Computer as a Tool. The IEEE Region 8*, Slovenia, 2003, pp. 83-86.

[49] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano, "On the Effectiveness of the Test-First Approach to Programming," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 226-237, March 2005.

[50] Daniel H. Steinberg, "The effect of unit tests on entry points, coupling and cohesion in an introductory Java programming course," in *XP Universe*, Raleigh, NC, 2001.

[51] Stephen H. Edwards, "Using software testing to move students from trial-and-error to reflection-in-action," in *SIGCSE '04 Proceedings of the 35th SIGCSE technical symposium on Computer science education* , New York, 2004, pp. 26-30.

[52] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 2, pp. 184-208, April 2001.

[53] David Willmor and Suzanne M. Embury, "A Safe Regression Test Selection Technique for Database-Driven Applications," in *ICSM '05 Proceedings of the 21st IEEE International Conference on Software Maintenance* , Washington, DC, 2005, pp. 421-430.

[54] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos, "TimeAware test suite prioritization," in *ISSTA '06 Proceedings of the 2006 international symposium on Software testing and analysis* , New York, 2006 , pp. 1-12.

[55] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159-182, February 2002.

[56] Gary McGrew. (2008, November) David A. Wheeler's Blog. [Online]. http://www.dwheeler.com/blog/2008/11/04/#automated-code-reviews-security

[57] Francesco Logozzo and Manuel Fähndrich, "On the relative completeness of bytecode analysis versus source code analysis," in *CC'08/ETAPS'08 Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction* , Berlin, 2008, pp. 197-212.

[58] Fausto Spoto, "Nullness Analysis in Boolean Form," in *SEFM '08 Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods* , Washington, DC, 2008, pp. 21-30.

[59] Roopesh Shenoy. (2012, June) InfoQ. [Online]. http://www.infoq.com/news/2012/06/continuous-test-runner-net

[60] Jean-Philippe Lang. (2012, April) Piece Framework. [Online]. http://piece-framework.com/projects/stagehand-testrunner/wiki

[61] David Gageot. (2012, December) Infinitest. [Online]. http://infinitest.github.com/

[62] Partha Kuchana, *Software Architecture Design Patterns in Java*. New York, United States of America: AUERBACH Publications, 2004.

[63] Jeffrey Dean, David Grove, and Craig Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," in *ECOOP '95 Proceedings of the 9th European Conference on Object-Oriented Programming* , London, 1995, pp. 77-101.

[64] Oracle. (2012, December) Java Regular Expressions. [Online]. http://docs.oracle.com/javase/tutorial/essential/regex/

[65] Alfred V. Aho, *Algorithms for finding patterns in strings*. Cambridge, United States of America: MIT Press, 1990.

[66] Jung-Min Kim, Adam Porter, and Gregg Rothermel, "An empirical study of regression test application frequency," in *The 22nd international conference on Software engineering*, New York, 2000, pp. 126-135.

[67] Gregg Rothermel and Mary Jean Harrold, "Analyzing regression test selection techniques," *Software Engineering, IEEE Transactions on*, vol. 22, no. 8, pp. 529-551 , August 1996.