International Journal of Artificial Intelligence and Applications for Smart Devices
Vol.2, No.2 (2014), pp.23-38
http://dx.doi.org/10.14257/ijaiasd.2014.2.2.03

Design and Implementation of a Selective Continuous Test Runner

Mohammed R. El Khoudary and Wesam M. Ashour

Computer Engineering Department, Islamic University of Gaza, Gaza, Palestine

Abstract

In this study we present the design and implementation of a selective continuous test runner (CT) as a way of reducing the time wasted while doing regression tests. Previous studies presented only a continuous test runner with random or semi-random test case prioritization (TCP) techniques. Unlike our CT which allows doing online regression test selection (RTS) as well as TCP. We evaluate our approach by doing a pilot study over a number of developers. Experimental results show significant decrease in regression time when using our technique.

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) (Beck K. , Test Driven Development: By Example, 2002). 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 (Wikipedia, 2005).

As these techniques prove to have advantages; they also brought disadvantages in debate. Industrially TDD 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.

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. Many approaches were introduced in that field like in (Chen, Rosenblum, & Vo, 1994; Elbaum, Malishevsky, & Rothermel, Test Case Prioritization: A Family of Empirical Studies, 2002; Willmor & Embury, 2005).

Test Case Prioritization (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 (Elbaum, Malishevsky, & Rothermel, Prioritizing test cases for regression testing, 2000; Walcott, Soffa, Kapfhammer, & Roos, 2006 ; Elbaum, Malishevsky, & Rothermel, Test Case Prioritization: A Family of Empirical Studies, 2002).

Other studies discussed the time wasted in regression tests when the ignorance time when the developer doesn't know about a bug until he/she runs the test suite and handled this by introducing the concept of Continuous Test Runners (CT) (Saff & Ernst, 2003). CT uses real-time integration with the development environment to asynchronously run tests that are always applied to the current version of a program.

The rest of the paper is organized as the following. Section II reviews some related work and CT techniques. In Section III we present the influence graph concept. In Section VI we introduce the proposed CT design. Pilot story and results are given in Section IV. Section V summarizes our proposed techniques and some proposed future work.

ISSN: 2288-6710 IJAIASD

2. Review of Related Works

One approach is TEST-RANK (Cibulski & Yehudai, 2011) 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. 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.

In (Saff & Ernst, 2003) 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 must also 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 techniques like suite order, round-robin, random, recent errors … etc.

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.

3. Influence Graph

Influence graph concept is a contribution made by us and is proposed for publication, in this section we introduce influence graph as a concept for RTS to be used with the CT.

A. 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:

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

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

B. Visualizing Influence Graph

Figure 1 shows a simple influence graph for code fragment.

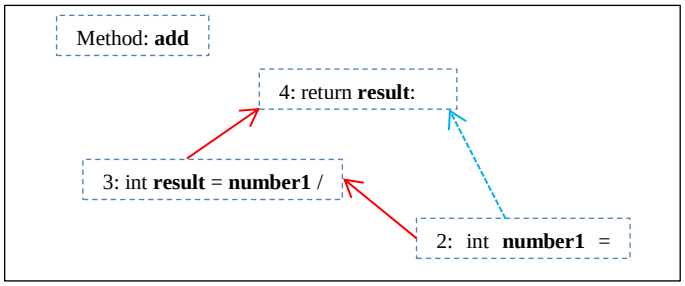

Figure 1. Influence Graph for Code Fragment 1

From Figure 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. Now let's see the influence graphs of code fragment 2 in Figure 2.

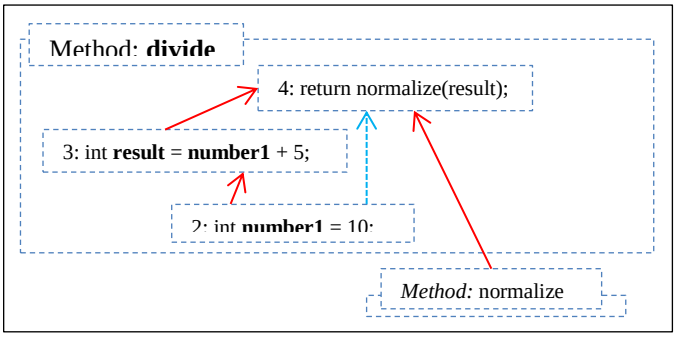

Figure 2. Influence Graph for Code Fragment 2

C. Usage of Influence Graph in RTS

We build influence graphs 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. 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 shows typical influence graphs for a test case.

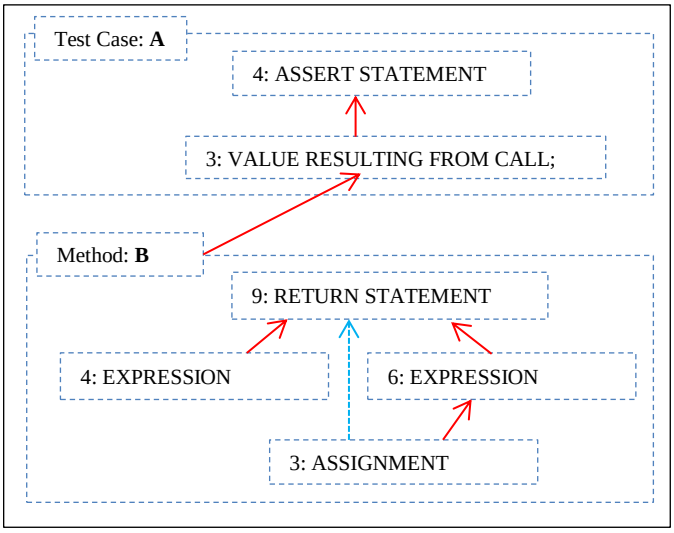

Figure 3. A Typical Influence Graph for a Test Case

D. Benefits of RTS with Influence Graphs

In (Graves, Harrold, Kim, Porter, & Rothermel, 2001) 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's made 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 (Cibulski & Yehudai, 2011) 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.

4. Design of the Proposed CT

The structure of the continuous test runner is shown in the block diagram in Figure 4.

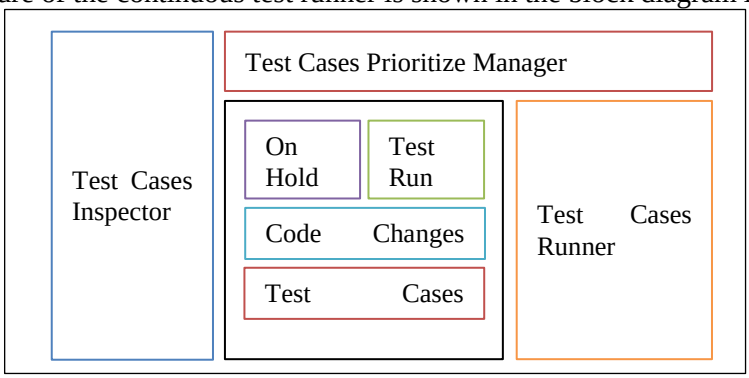

Figure 4. Proposed Continuous Test Runner Block Diagram

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.

E. Test Cases Inspector

Figure 5 shows a state diagram for the Test Cases Inspector.

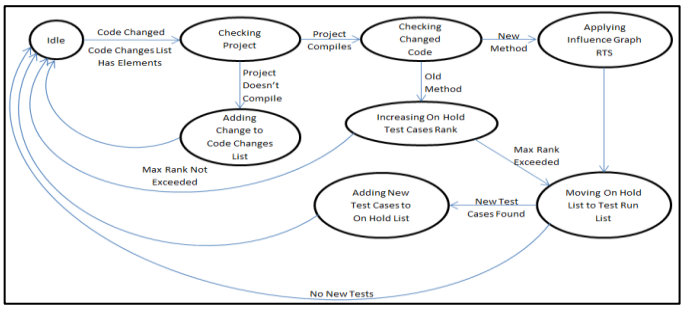

Figure 5. State Diagram for the Test Cases Inspector

As can be seen in Figure 5 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 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 I shows these parameters.

Table I. 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.

F. 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 II shows the currently existing parameters.

Table II. 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.

G. 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 6.

Figure 6. 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 III summarizes parameters used to customize the Test Cases Prioritize Manager.

Table III. 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 (Saff & Ernst, 2003): • **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.

H. 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 (Kuchana, 2004) 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.

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 IV Summarizes Parameters used to Customize the Test Cases Runner

Table IV. Test Cases Runner Customization Parameters

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

5. Experiments and Results

A. Experiment 1

1. The goal of this experiment is to prove the influence graph **performance aspects** in the following features:
2. If the modification doesn't affect the return value of the method there will be no modifications on the influence graph, unlike (Cibulski & Yehudai, 2011; Chen, Rosenblum, & Vo, 1994; Srivastava & Thiagarajan, 2002) where any modification is considered as a modification for retesting.
3. When the modification occurs the graph will not be completely rebuilt like in (Cibulski & Yehudai, 2011; Chen, Rosenblum, & Vo, 1994; Srivastava & Thiagarajan, 2002); instead only the affected nodes will be changed and any new nodes will be added.
4. The time required to update an influence graph in response to a change should be very small to be able to utilize it into a continuous test runner.

Given the following method:

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

This method is tested by the following test case:

```
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 6 shows the generated influence graph.

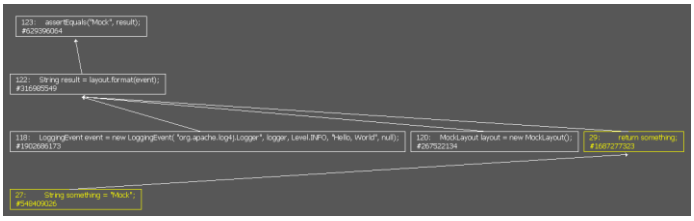

Figure 7. testFormat() Generated Influence Graph

As we can see in Figure 6; each influence graph is depicted with a distinct color. LayoutTest.java, for example, is drawn in white while MockLayout.java is drawn with yellow, 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.

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 follows:

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

Going back to check the influence graph; the graph in Figure 7 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.

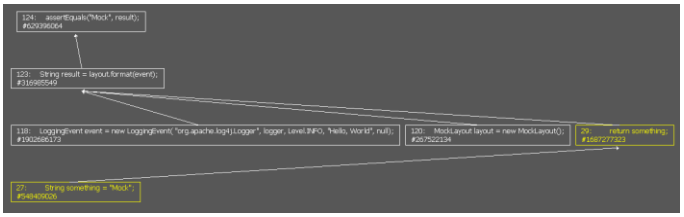

Figure 8. Irrelevant Modifications Effect on testFormat() influence Graph

Now let's link the temp variable to the result variable by performing the following modification:

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


The generated graph is shown in Figure 9.

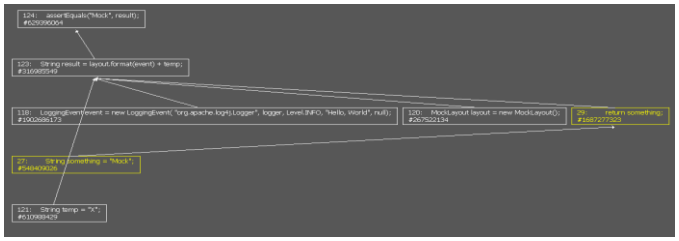

Figure 9. Relevant Modifications Effect on testFormat() influence Graph

Again we can see in Figure 5.3 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 do some modifications to the MockLayout.java to see the changes that could happen to the influence graph:

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

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 modification in the next code fragment a change will happen, the change is shown in figure 5.4.

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

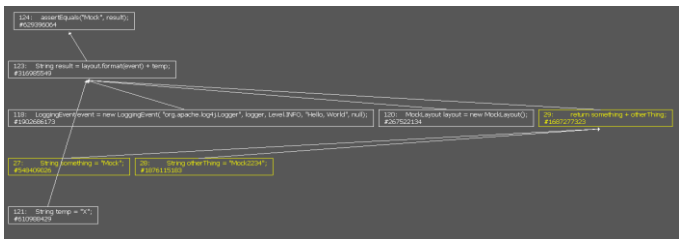

Figure 10. testFormat() influence Graph after adding otherThing Variable

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.

B. Pilot Study

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. (Crossman, 2013)

Other researchers used this methodology in their papers such as (Test Case Quality in Test Driven Development: A Study Design and a Pilot Experiment, 2012); 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 (Reece, 1997; Lo, 2012).

This study was conducted to evaluate the efficiency of the proposed CT design when used in a real programming project.

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 V shows the distribution of the developer's experience.

Table V. 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

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 (Saff & Ernst, 2003) proved

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

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.

Results

Our results prove that using our method significantly reduced the regression time compared to using the conventional method as shown in table IV. 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 VI. Summary Statistics for Development Time and Regression Time in Seconds 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 11).

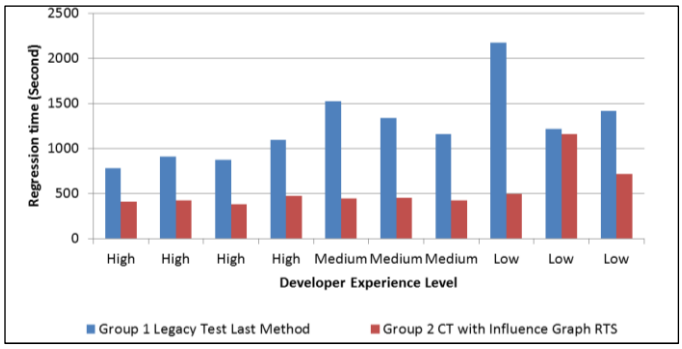

Figure 11. Regression Time for Study Participants By Level of Experience

Adjusting for experience level, participants who applied our method (group2) independently associated with less regression time compared to participants who applied the conventional method (table VII).

Table VII. Group Effect on Development Time* and Regression Time* Adjusting for Experience Level

	Development Time*		P value
Variables	β	SE	
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

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 proven in (Saff & Ernst, 2003) 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.

6. Conclusion and Future Work

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 concept 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 become.

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

For our CTR to be highly customizable we presented a sit of tunable configurations that allow developers to control the behavior of the CTR 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 CTR.

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 taking performance measures on real code and the others on an experimental project and was tested over 20 real developers with various levels of skill.

The following enhancements can be made in the future:

- 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.

References

[1] K. Beck, Test Driven Development: By Example. United States of America: Addison-Wesley, 2002.
[2] (2005, November) Wikipedia. [Online]. http://en.wikipedia.org/wiki/Test-driven_development#cite_note-Beck-
[3] 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.
[4] 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.
[5] 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.
[6] 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.
[7] 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.
[8] 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.
[9] 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.
[10] 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.
[11] Partha Kuchana, Software Architecture Design Patterns in Java. New York, United States of America: AUERBACH Publications, 2004.
[12] Amitabh Srivastava and Jay Thiagarajan, "Effectively Prioritizing Tests in Development Environment," Microsoft Research, Redmond, TechReport MSR-TR-2002-15, 2002.
[13] Ashely Crossman. (2013, February) Sociology. [Online]. http://sociology.about.com/od/P_Index/g/Pilot-Study.htm
[14] "Test Case Quality in Test Driven Development: A Study Design and a Pilot Experiment," in Evaluation & Assessment in Software Engineering (EASE 2012), 16th International Conference on, Sweden , 2012, pp. 223-227.
[15] Gloria A. Reece, "Reverse engineering of user documents: a pilot study," in Professional Communication Conference, 1997. IPCC '97 Proceedings. Crossroads in Communication., 1997 IEEE International, Memphis, 1997, pp. 449-462.
[16] Fai-hang Lo, "Pilot study of the use of mobile device for the study of life sciences students in Hong Kong," in Information Technology in Medicine and Education (ITME), 2012 International Symposium on, Hong Kong, 2012, pp. 94-97.

