Prioritizing Unit Test Creation for Test-Driven Maintenance of Legacy Systems

Emad Shihab, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Kingston, ON, Canada
{emads, zmjiang, bram, ahmed}@cs.queensu.ca

Robert Bowerman
Research In Motion
Waterloo, ON, Canada
rbowerman@rim.com

***Abstract*—Test-Driven Development (TDD) is a software development practice that prescribes writing unit tests before writing implementation code. Recent studies have shown that TDD practices can significantly reduce the number of pre-release defects. However, most TDD research thus far has focused on new development. We investigate the adaptation of TDD-like practices for already implemented code, in particular legacy systems. We call this adaptation of TDD-like practices for already implemented code "Test-Driven Maintenance" (TDM).**

In this paper, we present an approach that assists software development and testing managers, who employ TDM, utilize the limited resources they have for testing legacy systems efficiently. The approach leverages the development history of the project to generate a prioritized list of functions that managers should focus their unit test writing resources on. The list is updated dynamically as the development of the legacy system progresses. To evaluate our approach, we conduct a case study on a large commercial legacy software system. Our findings suggest that heuristics based on the function size, modification frequency and bug fixing frequency should be used to prioritize the unit test writing of legacy systems.

I. Introduction

Test-Driven Development (TDD) is a software development practice where developers consider a small subset of requirements, write and run unit tests that would pass once the requirements are implemented, implement the requirements and re-run the unit tests to make sure they pass [1], [2]. The unit tests are generally written at the granularity of the smallest separable module, which is the function in most cases [3].

Recently, empirical evidence has shown that TDD can reduce pre-release defect densities by as much as 90%, compared to other similar projects that do not implement TDD [4]. In addition, other studies showed that TDD helps produce better quality code [5], [6], improve programmer productivity [7] and strengthen the developer confidence in their code [8].

Most of the previous research to date studied the use of TDD for new software development. However, previous studies showed that more than 90% of the software development cost is spent on maintenance and evolution activities [9], [10]. Other studies showed that an average Fortune 100 company maintains 35 million lines of code and this amount of maintained code is expected to double every 7 years [11]. For this reason, we believe it is extremely beneficial to study the adaptation of TDD-like practices for the maintenance of already implemented code, in particular for legacy systems. In this paper we call this Test-Driven Maintenance (TDM).

Applying TDM to legacy systems is important because legacy systems are often instilled in the heart of newer, larger systems and continue to evolve with new code [12]. In addition, due to their old age, legacy systems lack proper documentation and become brittle and error-prone over time [13]. Therefore, TDM should be employed for these legacy systems to assure quality requirements are met and reduce the chance of failures due to evolutionary changes.

However, legacy systems are typically large and writing unit tests for an entire legacy system at once is time consuming and practically infeasible. To mitigate this issue, TDM uses the same divide-and-conquer idea of TDD. However, instead of focusing on a few tasks from the requirements documents, developers that apply TDM isolate functions of the legacy system and individually unit test them. Unit tests for the functions are incrementally written until a desired quality target is met.

The idea of incrementally writing unit tests is very practical and has 3 main advantages. First, it gives resource-strapped managers some breathing room in terms of resource allocation (i.e., it alleviates the need for long-term resource commitments). Second, developers can get more familiar with the legacy code through the unit test writing effort [14]. Third, unit tests can be easily maintained and updated in the future to assure the high quality of the legacy system [3].

Even after isolating functions of the large legacy system, the question of how to *prioritize* the writing of unit tests to achieve the best return on investment still lingers. Do we randomly write unit tests for functions? Do we write unit tests for the functions that we worked on most recently? Using the right prioritization strategy can save developers time, save the organization money and increase the overall product quality [15], [16].

In this paper, we present an approach that prioritizes the writing of unit tests for legacy software systems, based on the development history of these systems. We propose several heuristics to generate prioritized lists of functions to write unit tests for.

To evaluate our approach, we perform a case study on a large commercial legacy system. Our results show that using the proposed heuristics significantly improves the testing

efforts, in terms of potential bug detection, when compared to random test writing. Heuristics that prioritize unit testing effort based on function size, modification frequency and bug fixing frequency are the best performing.

Organization of Paper. We motivate our work using an example in Section II. Section III details our approach. The simulation-based case study is described in Section IV. The results of the case study are presented in Section V. A discussion on the effect of the simulation parameters on our results is provided in Section VI. The list of the threats to validity are presented in Section VII, followed by the related work in Section VIII. Section IX concludes the paper.

II. Motivating Example

In this section, we use an example to motivate our approach. Lindsay is a software development manager for a large legacy system that continues to evolve with new code.

To assure a high level of quality of the legacy system, Lindsay's team employs TDM practices. Using TDM, the team isolates functions of the legacy system and writes unit tests for them. However, deciding which functions to write unit tests for is a challenging problem that Lindsay and his team have to answer.

Writing unit tests for all of the code base is nearly impossible. For example, if a team has enough resources to write unit tests to assess the quality of 100 lines of system code per day, then writing unit tests for a 1 million line of code (LOC) system would take over 27 years. At the same time, the majority of the team is busy with new development and maintenance efforts. Therefore, Lindsay has to utilize his resources effectively in order to obtain the best return for his resource investment.

A primitive approach that Lindsay tries is to randomly pick functions and write unit tests for them or write tests for functions that are recently worked on. However, he quickly realizes that such an approach is not very effective. Some of the recently worked on functions are rarely used later, while others are so simple that writing unit tests for them is not a priority. Lindsay needs an approach that can assist him and his team prioritize the writing of unit tests for the legacy system. To assist development and testing teams like Lindsay's, we present an approach that uses the history of the project to prioritize the writing of unit tests for legacy software systems. The approach uses heuristics extracted from the project history to recommend a prioritized list of functions to write unit tests for. The size of the list can be customized to the amount of resources the team has at a specific time. The approach updates using the history of the project and continues to recommend functions to write unit tests for as the project progresses.

III. Approach

In this section, we detail our approach, which is outlined in Figure 1. In a nutshell, the approach extracts a project's historical data from its code and bug repositories, calculates various heuristics and recommends a list of functions to write unit tests for. Then, it re-extracts new data from the software repositories to consider new development activity and starts the process of calculating heuristics and generating a list of functions again. In the next three subsections, we describe each phase in more detail.

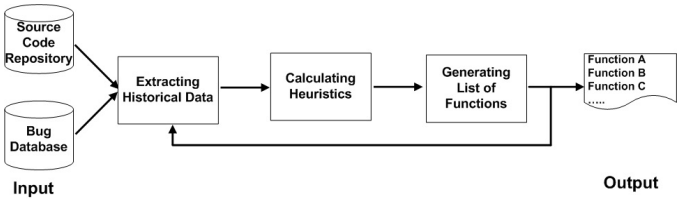


Fig. 1. Approach overview

A. Extracting Historical Data

The first step of the approach is to extract historical data from the project's development history. In particular, we combine modification information and source code from the source code control system (e.g., SVN [17], CVS [18]) with bug data stored in the bug tracking system (e.g., Bugzilla [19]). Each modification record contains the time of the modification (day, month, year and local time), the author, the files that changed, the version of the files, the line number where the change occurred in the file and a short description of the change. Similar to previous studies (e.g. [20]–[22]), we used a lexical technique to automatically classify modifications. The technique searches the modification record logs, that are stored in the source code repository, for keywords, such as "bug" or "bugfix", and bug identifiers (which we use to search the bug database) to do the classification. If a modification record contained a bug identifier or one of the keywords associated with a bug, then it is classified as a bug fixing modification. The modification records were grouped into two main categories: bug fixing modifications and general maintenance modifications.

The next step involves mapping the modification types to the functions that changed. To achieve this goal, we identify the files that changed and their file version numbers. Then, we extract the all versions of the files, parse them (to identify the individual functions) and compare consecutive versions of the files to identify which functions changed. Since we know which files and file versions were changed by a modification, we can pinpoint the functions modified by each modification. We label each of the changed functions with the modification type.

B. Calculating Heuristics

We use the extracted historical data to calculate various heuristics. The heuristics are used to prioritize different functions that we recommend for testing. We choose to use heuristics that can be extracted from a project's history for two main reasons: 1) legacy systems ought to have a very rich history that we can use to our advantage and 2) previous work in fault prediction showed that history based heuristics are good indicators of future bugs (e.g., [23], [24]). We conjecture that heuristics used for fault prediction will perform well, since

ideally, we want to write unit tests for the functions that have bugs in them.

The heuristics fall under four main categories: modification heuristics, bug fix heuristics, size heuristics and risk heuristics. The heuristics are listed in Table I. We also include a random heuristic that we use to compare the aforementioned heuristics to. For each heuristic, we provide a description, our intuition for using the heuristic and any related work.

The heuristics listed in Table I are a small sample of the heuristics that can be used to generate the list of functions. We chose to use these heuristics since previous work on fault prediction has proven their ability to perform well. However, any metric that captures the characteristics of the legacy system and can be linked to functions may be used to generate the list of functions.

C. Generating a List of Functions

Following the heuristic extraction phase, we use the heuristics to generate prioritized lists of functions that are recommended to have unit tests written for. Each heuristic generates a different prioritized list of functions. For example, one of the heuristics we use (i.e., MFM) recommends that we write tests for functions that have been modified the most since the beginning of the project. Another heuristic recommends that we write tests for functions that are fixed the most (i.e. MFF).

Then, we loop back to the historical data extraction phase, to include any new development activity and run through the heuristic calculation and list generation phases. Each time, a new list of functions, that should have unit tests written for them, is generated.

IV. Simulation-Based Case Study

To evaluate the performance of our approach, we conduct a simulation-based case study on a large commercial system. The software system is a legacy system, written in C and C++, which contains tens of thousands of functions totalling hundreds of thousands of lines of code. We used 5.5 years of the system's history to conduct our simulation, in which over 50 thousand modifications were analyzed[1]. In this section, we detail the case study steps and introduce our evaluation metrics.

A. Simulation study

The simulation ran in iterations. For each iteration we: 1) extract the historical data, 2) calculate the heuristics, 3) generate a prioritized list of functions, 4) measure the time it takes to write tests for the recommended list of functions and 5) filter out the list of functions that were recommended. Then, we advance the time (i.e., account for the time it took to write the unit tests) and do all of the aforementioned steps over again.

[1]We cannot disclose any more details about the studied system for confidentiality reasons.

Step 1. We used 5.5 years of historical data from the commercial legacy system to conduct our simulation. The first 6 months of the project were used to calculate the initial set of heuristics and the remaining 5 years are used to run the simulation.

Step 2. To calculate the heuristics, we initially look at the first 6 months of the project. If, for example we are calculating the MFM heuristic, we would look at all functions in the first 6 months of the project and rank them in descending order based on the number of times they were modified during that 6 month period. The amount of history that we consider to calculate the heuristics advances as we advance in the simulation. For example, an iteration 2 years into the simulation will use 2.5 years (i.e. the initial 6 months and 2 years of simulation time) of history when it calculates the heuristics.

Step 3. Next, we recommend a prioritized list of 10 functions that should have unit tests written for them. One list is generated for each of the heuristics. The list size of 10 functions is an arbitrary choice we made. If two functions have the same score, we randomly choose between them. We study the effect of varying the list size on our results in detail in Section VI. Furthermore, in our simulation, we assume that once a list of 10 functions is generated, tests will be written for all 10 functions before a new list is generated.

Step 4. Then, we calculate the time it takes to write tests for these 10 functions. To do so, we measure the size of the functions and divide by the total resources available. The size of the functions is used as a measure for the amount of effort required to write unit tests for those 10 functions [33], [34]. Since complexity is highly correlated with size [25], larger functions will take more effort/time to write unit tests for.

The number of available resources is a simulation parameter, expressed as the number of lines of code that testers can write unit tests for in one day. For example, if one tester is available to write unit tests for the legacy system and that tester can write unit tests for 50 lines of code per day, then a list of functions that is 500 lines will take him 10 days. In our simulation, we set the total test writing capacity of the available resources to 100 lines per day. We study the study the effect of varying the resources available to write unit tests in more detail in Section VI.

Step 5. Once a function is recommended to have a test written for it, we filter it out of the pool of functions that we use to generate future lists. In other words, we assume that once a function has had a unit test written for it, it will not need to have a new test written for it from scratch in the future; at most the test may need to be updated. We make this assumption for the following reason: once the function is recommended and the initial unit test is written, then this initial unit test will make sure all of the function's current code is tested. Also, since the team adopts TDM practices any future additions/changes to the function will be accompanied by unit tests that test the new additions/changes, therefore, they will not need to be prioritized again.

We repeat the 5-step process mentioned above for a period of 5 years. To evaluate the performance of the different

TABLE I
LIST OF ALL HEURISTICS USED TO PRIORITIZE THE WRITING OF UNIT TESTS FOR LEGACY SYSTEMS

Category	Heuristic	Prioritization Order	Description	Intuition	Related Work
Modifications	Most Frequently Modified (MFM)	Highest to lowest	Functions that were modified the most since the start of the project.	Functions that are modified frequently tend to become disorganized over time, leading to more bugs.	The number of prior modifications to a file is a good predictor of its potential bugs [23], [25], [26], [27].
	Most Recently Modified (MRM)	Latest to oldest	Functions that were most recently modified.	Functions that were modified most recently are the ones most likely to have a bug in them (due to the recent changes).	More recent changes contribute more bugs than older changes [25].
Bug Fixes	Most Frequently Fixed (MFF)	Highest to lowest	Functions that were fixed the most since the start of the project.	Functions that are frequently fixed in the past will have to be fixed in the future.	Prior bugs are a good indicator of future bugs [28].
	Most Recently Fixed (MRF)	Latest to oldest	Functions that were most recently fixed.	Functions that were fixed most recently are more likely to have a bug in them in the future.	The recently fixed heuristic was used to prioritize buggy subsystems in [22].
Size	Largest Modified (LM)	Largest to smallest	The largest (in terms of total lines of code), modified functions. Total lines of code account for source, comment and blank lines of code.	Large functions are more likely to have bugs than smaller functions.	The simple lines of code metric correlates well with most complexity metrics (e.g., McCabe complexity) [25], [27], [29] and [30].
	Largest Fixed (LF)	Largest to smallest	The largest (in terms of total lines of code), fixed functions. Total lines of code account for source, comment and blank lines of code.	Large functions that need to be fixed are most likely to have more bugs than smaller functions that are fixed less.	The simple lines of code metric correlates well with most complexity metrics (e.g., McCabe complexity) [25], [27], [29] and [30].
Risk	Size Risk (SR)	Highest to lowest	Riskiest functions, defined as the number of bug fixing changes divided by the size of the function in lines of code.	Since larger functions may naturally need to be fixed more than smaller functions, we normalize the number of bug fixing changes by the size of the function. This heuristic will mostly point out the small functions that are fixed a lot (i.e., have high defect density).	Using relative churn metrics performs better than using absolute values when predicting defect density [31].
	Change Risk (CR)	Highest to lowest	Riskiest functions, defined as the number of bug fixing changes divided by the total number of changes.	The number of bug fixing changes normalized by the total number of changes. For example, a function that changes 10 times in total and out of those 10 times 9 of them were to fix a bug should have a higher priority to be tested than a function that changes 10 times where only 1 of those ten is a bug fixing change.	Using relative churn metrics performs better than using absolute values when predicting defect density [31].
Random	Random	Random	Randomly selects functions to write unit tests for.	Randomly selecting functions to test can be thought of as a base line scenario. Therefore, we use the random heuristic's performance as a base line to compare the performance of the other heuristics to.	Previous studies on test prioritization used a random heuristic to compare their performance [15], [16], [32].

heuristics, we periodically (every 3 months) measure the performance using two metrics: Usefulness and Percentage of Optimal Performance (POP), which we describe next.

B. Performance Evaluation Metrics

Usefulness

The first question that comes up after we write unit tests for a set of functions is - was writing the tests for these functions worth the effort? For example, if we write unit tests for functions that rarely change and/or have no bugs after the tests are written, then our effort may be wasted. Ideally, we would like to write unit tests for the functions that have bugs in them.

We define the usefulness metric to answer the aforementioned question. The usefulness metric is defined as *the percentage of functions we write unit tests for that have one or more bugs after the tests are written*. The usefulness metric indicates how much of our effort on writing unit tests is actually worth the effort.

We use the example in Figure 2 to illustrate how we calculate the usefulness. Functions A and B have more than 1 bug fix after the unit tests were written for them (after point 2 in Figure 2). Function C did not have any bug fix after we wrote the unit test for it. Therefore, for this list of three functions, we calculate the usefulness as $\frac{2}{3} = 0.666$ or 66.6%.

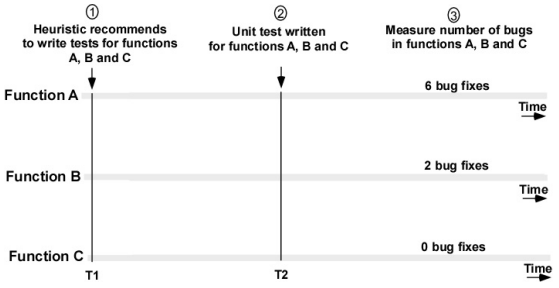


Fig. 2. Usefulness example

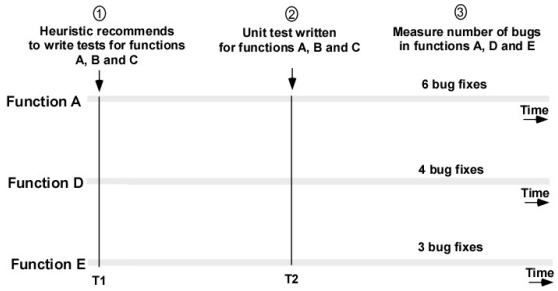


Fig. 3. POP example

Percentage of Optimal Performance (POP)

In addition to calculating the usefulness, we would like to measure how well a heuristic performs compared to the optimal (i.e., the best we can ever do). Optimally, one would have perfect knowledge of the future and write unit tests for functions that are the most buggy. This would yield the best return for their investment.

To measure how close we are to the optimal performance, we define a metric called Percentage of Optimal Performance (POP). To calculate the POP, we generate two lists: one is the list of functions generated by the heuristic and the second is a optimal list of functions. The optimal list contains the functions with the most bugs from the time the unit tests were written till the end of the simulation. Assuming that the list size is 10 functions, we calculate the POP as the number of bugs in the top 10 functions, from the list generated by the heuristics, divided by the number of bugs the top 10 optimal functions contain (i.e., functions in the optimal list). Simply put, the POP is *the percentage of bugs we can avoid using a heuristic compared to the best we can do if we had perfect knowledge of the future*.

We illustrate the POP calculation using the example shown in Figure 3. At first, we generate a list of functions that we write unit tests for using a specific heuristic (e.g., MFM or MFF). Then, based on the size of these functions, we calculate the amount of time it takes to write unit tests for these functions (point 2 in Figure 3). From that point on, we calculate the number of bugs for all of the functions and rank them in descending order. For the sake of this example, let us assume we are considering the top 3 functions. Assuming our heuristic identifies functions A, B and C as the functions we need to write unit tests for, however, these functions may not be the ones with the most bugs. Assuming that the functions with the most bugs are functions A, D and E (i.e., they are the top 3 on the optimal list). From Figure 2 , we can see that functions A, B and C had 8 bug fixes in total after the unit tests were written for them. At the same time, Figure 3 shows that the optimal functions (i.e., functions A, D and E) had 13 bug fixes in them. Therefore, the best we could have done is to remove 13 bugs. We were able to remove 8 bugs using our heuristic, hence our POP is $\frac{8}{13} = 0.62$ or 62%.

It is important to note that the key difference between the usefulness and the POP values is that usefulness is the percentage of *functions* that we found useful to write unit tests for. On the other hand, POP measures the *bugs* we could avoid using a specific heuristic.

V. Case Study Results

In this section, we present the results of the usefulness and POP metrics for the proposed heuristics. Ideally, we would like to have high usefulness and POP values. To evaluate the performance of each of the heuristics, we use the random heuristic as our baseline. If we cannot do better than just randomly choosing functions to add to the list, then the heuristic is not that effective. Previous studies on test case prioritization, such as [15], [16], [32], commonly used random lists to compare their proposed solutions to. Since the random heuristic can give a different ordering each time, we use the average of 5 runs, each of which uses different randomly generated seeds.

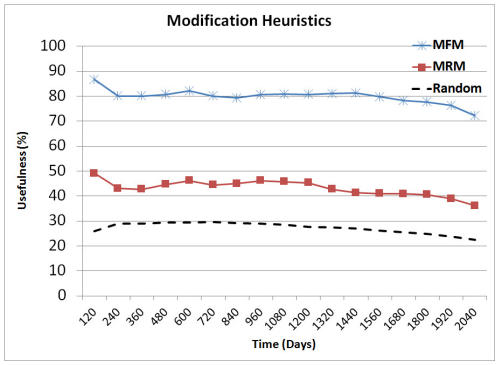
A. Usefulness

Usefulness measures how many of the functions recommended to have unit tests written for have one or more bugs after the unit tests are written. We calculate the usefulness for each heuristic and plot it over time in Figure 4. The dashed black line in each of the figures depicts the results of the random heuristic. From the figures, we can observe that in all cases and throughout the simulation, the proposed heuristics outperform the random heuristic.

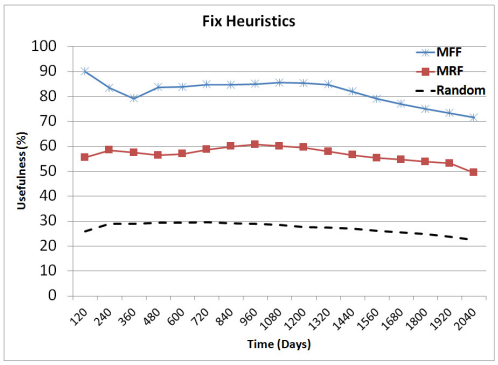
The median usefulness values for each of the heuristics are listed in Table II. Since the usefulness values change over the course of the simulation, we chose to present the median values to avoid any sharp fluctuations. The last row of the table shows the usefulness achieved by the random heuristic. The heuristics are ranked from 1 to 9, with 1 indicating the best performing heuristic and 9 the worst.

The LF, LC, MFF and MFM are the top performing heuristics, having median values in the range of 80% to 87%. These heuristics perform much better than the, i.e., writing tests for functions that they currently work on (i.e., MRM and MRF). We can observe from the Figures 4(a) and 4(b) that the recency heuristics (i.e., MRM and MRF) perform poorly compared to their frequency counterparts (i.e., MFM and MFF) and the size-based heuristics.

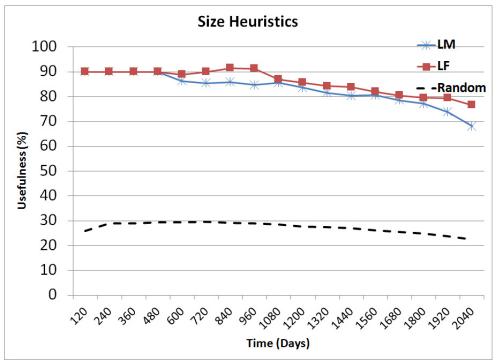
Taking the median values in Table II, we can see that for MFM we were able to achieve 80% median usefulness. This means that approximately 8 out of the 10 functions we wrote



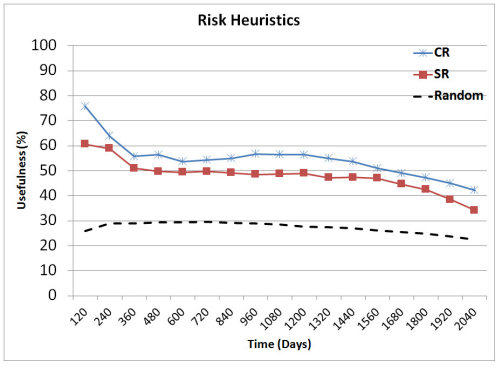
(a) Usefulness of modification heuristics



(b) Usefulness of fix heuristics

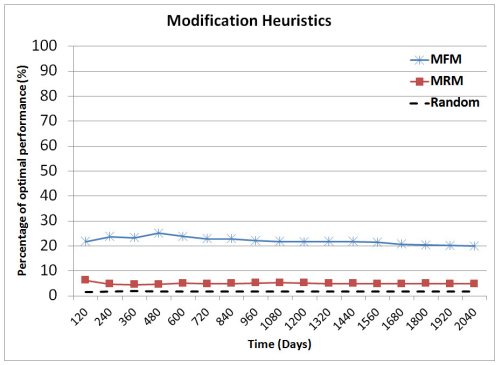


(c) Usefulness of size heuristics

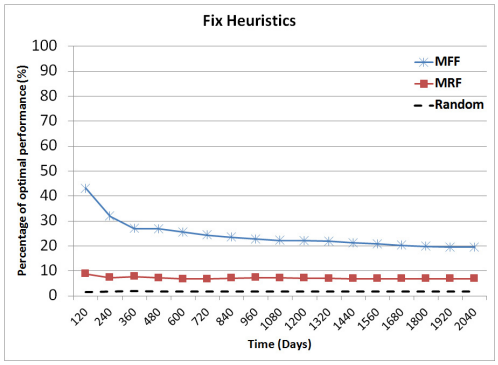


(d) Usefulness of risk heuristics

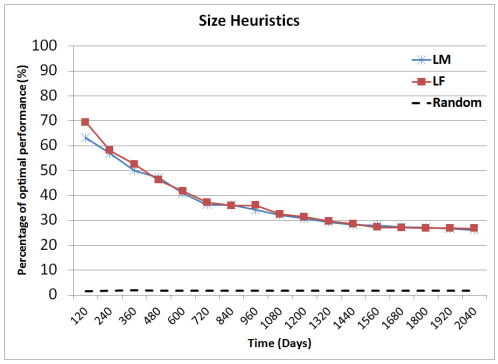
Fig. 4. Usefulness of heuristics compared to the random heuristic



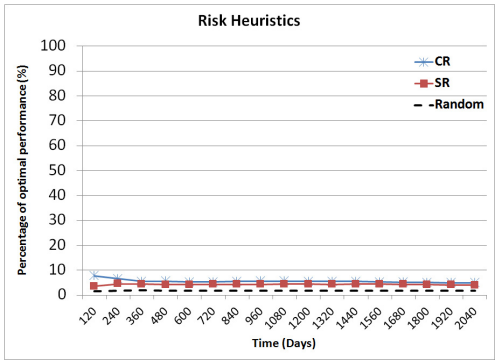
(a) POP of modification heuristics



(b) POP of fix heuristics



(c) POP of size heuristics



(d) POP of risk heuristics

Fig. 5. POP of heuristics compared to the random heuristic

TABLE II
USEFULNESS RESULTS

Heuristic	Median Usefulness	Rank
LF	87.0%	1
LM	84.7%	2
MFF	83.8%	3
MFM	80.0%	4
MRF	56.9%	5
CR	55.0%	6
SR	48.8%	7
MRM	43.1%	8
Random	27.7%	9

TABLE III
PERCENTAGE OF OPTIMAL PERFORMANCE RESULTS

Heuristic	Median POP	Rank
LF	32.4%	1
LM	32.2%	2
MFF	22.2%	3
MFM	21.8%	4
MRF	7.0%	5
CR	5.5%	6
MRM	4.9%	7
SR	4.3%	8
Random	1.7%	9

unit tests for had one or more bugs in the future. Therefore, writing the unit tests for these functions was useful. On the contrary, for the random heuristic, approximately 3 out of every 10 functions we wrote unit tests for had 1 or more bugs after the unit tests were written.

Size, modification frequency and fix frequency heuristics should be used to prioritize the writing of unit tests for legacy systems. These heuristics achieve median usefulness values between 80–87%.

B. Percentage of Optimal Performance (POP)

In addition to calculating the usefulness of the proposed heuristics, we would like to know how close we are to the optimal list of functions that we should write unit tests for if we have perfect knowledge of the future. We present the POP values for each of the heuristics in Figure 5. The performance of the random heuristic is depicted using the dashed black line. The figures show that in all cases, the proposed heuristics outperform the random heuristic.

The median POP values are shown in Table III. The POP values for the heuristics are lower than the usefulness values. The reason is that usefulness gives the percentage of functions that have one or more bugs. However, POP measures the percentage of bugs the heuristic can potentially avoid in comparison to the best we can do, if we have perfect knowledge of the future.

Although the absolute POP percentages are lower compared to the usefulness measure, the ranking of the heuristics remained quite stable (except for the SR and MRM, which exchanged 7th and 8th spot). Once again, the best performing heuristics are LF, LM, MFF and MFM. The median values for these top performing heuristics are in the 20% to 32.4% range. These values are much higher than the 1.7% that can be achieved using the random heuristic.

We can observe a decline in the usefulness and POP values at the beginning of the simulation, shown in Figures 4 and 5. This decline can be attributed to the fact that initially, there are many buggy functions for the heuristics to choose from. Then, after these buggy functions have been recommended, we remove them from the pool of functions that we can recommend. Therefore, the heuristics begin to recommend some functions that are not buggy. This phenomenon is evident in all types of simulations, where a small warm-up period is required for the simulation to reach steady state [35].

Size, modification frequency and fix frequency heuristics should be used to prioritize the writing of unit tests for legacy systems. These heuristics achieve POP values between 21.8–32.4%.

VI. DISCUSSION

During our simulation study, we needed to decide on two simulation parameters: list size and available resources. In this section, we discuss the effect of varying these simulation parameters on our results. It is important to study the effect of these simulation parameters on our results because it helps us better understand the results we obtain from the simulation. Due to space limitations, we only present the POP values.

A. Effect of List Size

In our simulations, each of the heuristics would recommend a list of functions that should have unit tests written for. Throughout our study, we used a list size of 10 functions. However, this list size was an arbitrary choice. We could have set this list size to 5, 20, 40 or even 100 functions. The size of the list will affect the usefulness and POP values.

To analyze the effect of list size, we vary the list size and measure the corresponding POP values. We measure the median POP for each list size and plot the results in Figure 6. The y-axis is the log of the median POP value and the x-axis is the list size. We observe a common trend - an increase in the list size increases the POP for all heuristics. Once again, our top performing heuristics are unchanged with LF, LC, MFF and MFM scoring in the top for all list sizes. The same trend was also observed for the usefulness values.

This trend can be explained by the fact that a bigger list size will make sure that more functions have unit tests written for them earlier on in the project. Since these functions are tested earlier on, we are able to avoid more bugs and the POP increases.

B. Effect of Available Resources

Another important simulation parameter that we needed to set in the simulations is, the effort available to write unit tests. This parameter determines how fast a unit test can be written.

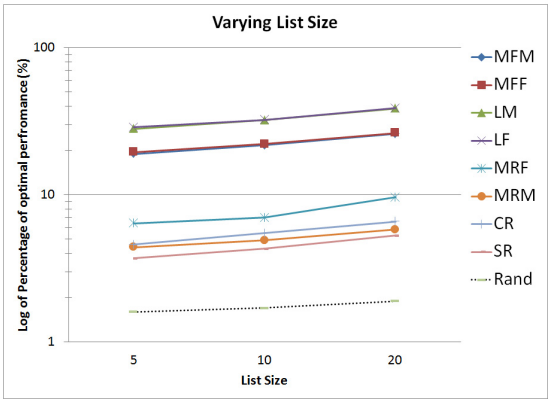


Fig. 6. Effect of varying list size on POP

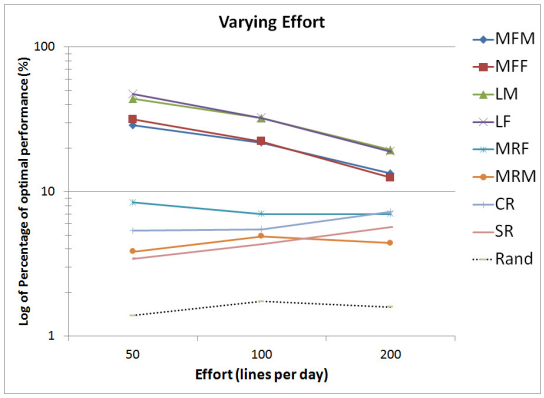


Fig. 7. Effect of varying effort on POP

For example, if a function is 100 lines of code, and a tester can write unit tests for 50 lines of code per day, then she will be able to write unit tests for that function in 2 days.

Throughout our study, we set this value to 100 lines per day. If this value is increased, then testers can write unit tests faster (due to an increase in man power or due to more efficient testers) and write tests for more functions. On the other hand, if we decrease this value, then it will take longer to write unit tests.

We varied this value from 50 to 200 lines per day and measured the median POP. The results are plotted in Figure 7. We observe three different cases:

1) POP decreases as we increase the effort for heuristics LF, LM, MFF and MFM.
2) POP increases as we increase the effort for heuristics CR and SR.
3) POP either increases, then decreases or decreases, then increases as we increase the effort for heuristics MRF, MRM and Random.

We examined the results in more depth to try and explain the observed behavior. We found that in case 1, the POP decreases as we increase the effort because as we write tests for more functions (i.e., increasing effort available to 200 lines per day), we were writing tests for functions that did not have bugs after the tests were written. Or in other words, as we decrease the effort, less functions had unit tests written for them and this decreases the chance of prioritizing functions that do not have as many (or any) bugs in the future. In case 2, we found that the risk heuristics mostly identified functions that had a small number of bugs. Since an increase in effort means more functions can have unit tests written for them, therefore, we see an increase in the POP as effort is increased. In case 3, the MRF and MRM heuristics identify functions that are most recently modified or fixed. Any change in the effort will change the time it takes to write unit tests. This change in time will change the list of functions that should have unit tests written for them. Therefore, an increase or decrease in the effort randomly affects the POP. As for the random heuristic, by definition, it picks random functions to write unit tests for. Therefore, an increase or decrease in the effort randomly affects its POP.

VII. Threats to Validity

Construct validity: We used the POP and Usefulness measures to compare the performance of the different heuristics. Although POP and Usefulness are good measures, they may not capture all of the costs associated with creating the unit tests, maintaining the test suites and the cost of the different bugs (i.e., minor vs. major bugs).

Internal validity: In our simulations, we used 6 months to calculate the initial set of heuristics. Changing the length of this period may effect the results from some heuristics. In the future, we plan to study the effect of varying this initial period in more detail.

Our approach assumes that there is enough history about each function so that the different heuristics can be extracted. Although our approach is designed for legacy systems, in certain cases new functions may be added, in which case little or no history can be found. In such cases, we ask practitioners to carefully examine and monitor such functions manually until enough history is accumulated to use our approach.

When calculating the amount of time it takes to write the unit test for a function in our simulations, we make the assumption that all lines in the function will require the same effort. This may not be true for some functions.

Additionally, our simulation assumes that if a function is recommended once, it needs not be recommended again. This assumption is fueled by the fact that TDM practices are being used and after the initial unit test, all future development will be accompanied by unit tests that test the new functionality.

We assume that tests can be written for individual functions. In some cases, functions are closely coupled with other functions. This may make it impossible to write unit tests for the functions or impose the condition that unit tests for these closely coupled functions need to be written simultaneously.

Throughout our simulation study, we assume that all bug fixes are treated equally. However, some bugs have a higher severity and priority than others. In the future, we plan to consider the bug severity and priority in our simulation study.

External validity: Although our study focused on a large commercial legacy software system with a rich history, our findings might not generalize for all commercial or open source software systems.

VIII. Related Work

The related work can be categorized into two main categories: test case prioritization and fault prediction using historical data.

Test Case Prioritization.

The majority of the existing work on test case prioritization has looked at prioritizing the execution of tests during regression testing to improve the fault detection rate [15], [32], [36]–[38].

Rothermel *et al.* [38] proposed several techniques that use previous test execution information to prioritize test cases for regression testing. Their techniques ordered tests based on the total coverage of code components, the coverage of code components not previously covered and the estimated ability to reveal faults in the code components that they cover. They showed that all of their techniques were able to outperform untreated and randomly ordered tests. Similarly, Aggrawal *et al.* [37] proposed a model that optimizes regression testing while achieving 100% code coverage.

Elbaum *et al.* [15] showed that test case prioritization techniques can improve the rate of fault detection of test suites in regression testing. They also compared statement level and function level techniques and showed that at both levels, the results were similar. In [32], the same authors improved their test case prioritization by incorporating test costs and fault severities and validated their findings using several empirical studies [32], [39], [40].

Kim *et al.* [41] utilized historical information from previous test suite runs to prioritize tests. Walcott *et al.* [42] used genetic algorithms to prioritize test suites based on the testing time budget.

Our work differs from the aforementioned work in that we do not assume that tests are already written, rather, we are trying to deal with the issue of which functions we should write tests for. We are concerned with the prioritization of unit test writing, rather than the prioritization of unit test execution. Due to the fact that we do not have already written tests, we have to use different heuristics to prioritize which functions of the legacy systems we should write unit tests for. For example, some of the previous studies (e.g., [41]) use historical information based on previous test runs. However, we do not have such information since the functions we are trying to prioritize have never had tests written for them in the first place.

Fault Prediction using Historical Data.

Another avenue of closely related work is the work done on fault prediction. Nagappan *et al.* [24], [31] showed that dependency and relative churn measures are good predictors of defect density and post-release failures. Holschuh *et al.* [43] used complexity, dependency, code smell and change metrics to build regression models that predict faults. They showed that these models are accurate 50-60% of the time, when predicting the 20% most defect-prone components. Additionally, studies by Arishlom *et al.* [23], Graves *et al.* [25], Khoshgoftaar *et al.* [26] and Leszak *et al.* [27] have shown that prior modifications are a good indicator of future bugs. Yu *et al.* [28], and Ostrand *et al.* [29] showed that prior bugs are a good indicator of future bugs. Hassan [44] showed that the complexity of changes are good indicators of potential bugs.

Mende and Koschke [34] examined the use of various performance measures of bug prediction models. They concluded that performance measures should always take into account the size of source code predicted as defective, since the cost of unit testing and code reviews is proportional to the size of a module.

Other work used the idea of having a cache that recommends buggy code. Hassan and Holt [22] used change and fault metrics to generate a Top Ten list of subsystems (i.e., folders) that managers need to focus their testing resources on. Kim *et al.* [45] use the idea of a cache that keeps track of locations that were recently added, recently changed and where faults were fixed to predict where future faults may occur (i.e. faults within the vicinity of a current fault occurrence).

There are two key differences between our work and the work on fault prediction. First, our work prioritizes functions at a finer granularity than previous work on fault prediction. Instead of identifying buggy files or subsystems, we identify buggy functions. This difference is critical since we are looking to write unit tests for the recommended functions. Writing unit tests for entire subsystems or files may be wasteful, since one may not need to test all of the functions in the file or subsystem. Second, fault prediction techniques provide a list of potentially faulty components (e.g., faulty directories or files). Then it is left up to the manager to decide how to test this directory or file. Our work puts forward a concrete approach to assist in the prioritization of unit test writing, given the resources available and knowledge about the history of the functions.

IX. Conclusions

In this paper, we present an approach to prioritize the writing of unit tests for legacy systems. Different heuristics are used to generate lists of functions that should have unit tests written for them. To evaluate the performance of each of the heuristics, we perform a simulation-based case study on a large commercial legacy software system. We compared the performance of each of the heuristics to that of a random heuristic, which we use as a base line comparison. All of the heuristics outperformed the random heuristic in terms of usefulness and POP. Furthermore, our results showed that heuristics based on the function size, modification frequency and bug fixing frequency perform the best for the purpose of unit test writing prioritization. Finally, we studied the effect of varying list size and the resources available to write unit test on the performance of the heuristics.

In the future we plan to combine heuristics and study whether there exists an optimal combinations of heuristics that can be used to effectively prioritize the writing of unit tests.

ACKNOWLEDGMENTS

This work has been supported in part by research grants from the Natural Science and Engineering Research Council of Canada and a Walter C. Sumner Foundation Fellowship. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of Research In Motion. Moreover, our results do not in any way reflect the quality of Research In Motion's software products.

REFERENCES

[1] L. Williams, E. M. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice," in *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, 2003.

[2] K. Beck and M. Fowler, *Planning extreme programming*. Addison-Wiley, 2001.

[3] P. Runeson, "A survey of unit testing practices," *IEEE Softw.*, vol. 23, no. 4, 2006.

[4] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," *Empirical Softw. Engg.*, vol. 13, no. 3, 2008.

[5] B. George and L. Williams, "An initial investigation of test driven development in industry," in *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, 2003.

[6] ——, "A structured experiment of test-driven development," *Information and Software Technology*, vol. 46, no. 5, 2003.

[7] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Transaction on Software Engineering*, vol. 31, no. 3, 2005.

[8] M. M. Müller and W. F. Tichy, "Case study: extreme programming in a university environment," in *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, 2001.

[9] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, no. 3, 2000.

[10] J. Moad, "Maintaining the competitive edge," *Datamation*, vol. 64, no. 66, 1990.

[11] H. Muller, K. Wong, and S. Tilley, "Understanding software systems using reverse engineering technology," 1994.

[12] K. Bennett, "Legacy systems: Coping with success," *IEEE Softw.*, vol. 12, no. 1, 1995.

[13] J. Bisbal, D. Lawless, B. Wu, and J. Grimson, "Legacy information systems: Issues and directions," *IEEE Software*, vol. 16, no. 5, 1999.

[14] E. R. Harold, "Testing legacy code," http://www.ibm.com/developerworks/java/library/j-legacytest.html.

[15] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, 2000.

[16] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a junit testing environment," in *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, 2004.

[17] M. J. Rochkind, "The source code control system," *IEEE Transactions on Software Engineering*, vol. 1, no. 4, 1975.

[18] W. F. Tichy, "Rcs - a system for version control," *Software: Practice and Experience*, vol. 15, no. 7, 1985.

[19] "Bugzilla," http://www.bugzilla.org/.

[20] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004.

[21] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, 2000.

[22] A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, 2005.

[23] E. Arisholm and L. C. Briand, "Predicting fault-prone components in a java legacy system," in *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, 2006.

[24] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, 2007.

[25] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions of Software Engineering*, vol. 26, no. 7, July 2000.

[26] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl, "Data mining for predictors of software quality," *International Journal of Software Engineering and Knowledge Engineering*, vol. 9, no. 5, 1999.

[27] M. Leszak, D. E. Perry, and D. Stoll, "Classification and evaluation of defects in a project retrospective," *J. Syst. Softw.*, vol. 61, no. 3, 2002.

[28] T.-J. Yu, V. Y. Shen, and H. E. Dunsmore, "An analysis of several software defect models," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, 1988.

[29] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, 2004.

[30] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles, "Towards a theoretical model for software growth," in *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2007.

[31] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005.

[32] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, 2001.

[33] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data mining techniques for building fault-proneness models in telecom java software," in *ISSRE '07: Proceedings of the The 18th IEEE International Symposium on Software Reliability*, 2007.

[34] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *PROMISE '09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, 2009.

[35] R. K. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[36] W. E. Wong, J. R. Horgan, S. London, and H. A. Bellcore, "A study of effective regression testing in practice," in *ISSRE '97: Proceedings of the Eighth International Symposium on Software Reliability Engineering*, 1997.

[37] K. K. Aggrawal, Y. Singh, and A. Kaur, "Code coverage based technique for prioritizing test cases for regression testing," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, 2004.

[38] G. Rothermel, R. J. Untch, and C. Chu, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, 2001.

[39] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, 2002.

[40] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Control*, vol. 12, no. 3, 2004.

[41] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, 2002.

[42] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, 2006.

[43] T. Holschuh, M. Puser, K. Herzig, T. Zimmermann, P. Rahul, and A. Zeller, "Predicting defects in sap java code: An experience report." in *ICSE '09: Proceedings of the 31th International Conference on Software Engineering*, 2009.

[44] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, 2009.

[45] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, 2007.