Avoiding bugs pro-actively by change-oriented programming

Quinten David Soetens*
University of Antwerp
Middelheimlaan 1
Antwerpen, Belgium
quinten.soetens@ua.ac.be

Peter Ebraert†
University of Antwerp
Middelheimlaan 1
Antwerpen, Belgium
peter.ebraert@ua.ac.be

Serge Demeyer
University of Antwerp
Middelheimlaan 1
Antwerpen, Belgium
serge.demeyer@ua.ac.be

ABSTRACT

To minimize the cost of fixing bugs, they need to be identified as soon as possible. Testing the system is a means to detect such defects. However, when all the tests are run, it is often difficult to locate the precise error that causes a test to fail. We propose to use change-oriented programming as a means to detect bugs in a system while it is being developed. Not only may this technique assists in detecting bugs sooner, it can also be used to provide the developer with fine-grained information about the places in the program and test code that relate to the defect. We believe that this will decrease the probability of introducing bugs in a system and speed up the detection of the bugs that do get in.

1. INTRODUCTION

Already in the 1980's it was the goal of the *Programmer's Apprentice project* to provide every developer with an "apprentice" to assist him with the mundane tasks of software development [18] [19]. The apprentice is an intelligent computer program that actively participates in the software engineering process. The ultimate goal was to fully automate the development of software. We are still very far from achieving this and programming is still being done by human developers who are not flawless. As such, bugs in a software system are still inevitable.

The relative cost of fixing mistakes increases in every phase of the software engineering lifecycle [16]. It is therefore recommendable that we find defects in a system as soon as possible. Software testing is one way of doing this. If a problem in the implementation is found in the testing phase, it still costs 10 times more to fix than if it had already been found in the implementation phase itself [15]. Some industrial software companies run their entire test suite every night. Yet this still imposes a problem. As an entire day of development has a large impact on the software system, it is usually difficult to find the actual defect that causes a failing test.

In this paper, we introduce a technique to detect mistakes in the program code while it is being developed. The technique is based on *change-oriented programming* (ChOP); a programming style that centralizes change as the main development entity. Each time we perform a small set of changes, the test suite is run to verify that we have not introduced any defects in the system. Should any of the tests fail, we are still aware of what changes we introduced to produce this fault. Our technique is also capable of providing the developer with fine-grained information on which places in the program code (this is the source code that implements the functionality of the system) and test code (this is the source code that implements the tests) are related to the failing test. We assume that this will imply a significant decrease in the cost of finding and fixing the defect.

The rest of this paper is structured as follows. We start by introducing ChOP in Section 2 followed by an explanation of unit testing and what this looks like when developed in a change-based way (Section 3). Next, we discuss the different dependencies between changes (Section 4) and explain how they can be used to debug the system as it is being developed (Section 5). Afterwards, we show how these dependencies can also be used to optimize our approach (Section 6). We conclude with an overview of the related work (Section 7), the future work (Section 8) and a summary of the paper (Section 9).

2. CHANGE-ORIENTED PROGRAMMING

We first introduced *change-oriented programming* (ChOP) in [10]. ChOP is a style of programming that centralizes change. In order to create a piece of software in ChOP, a programmer does not need to write large streams of source code, but rather uses the integrated development environment (IDE) to *apply changes* to his source code.

Most mainstream IDE's already partially support ChOP. An example of this support can be found in the creation of a new class in the Eclipse IDE, where the programmer can create a class by right-clicking a package in order to add a new

*This work has been carried out in the context of the Interuniversity Attraction Poles Programme - Belgian State – Belgian Science Policy, project *MoVES*.

†Dr. Ebraert is funded by the "Agentschap voor Innovatie door Wetenschap en Technologie" via the Optimma research project.

class to that package. The IDE then provides the necessary dialogs to interact with the programmer in order to obtain all parameters of the class. Finally, it is the IDE that produces the source code of the newly created class and that inserts this code in the correct locations. Another example is the refactoring[1] support of the VisualWorks IDE [4]. When a programmer decides to *pull up* a method (move it to the superclass), he right-clicks the method and selects the "pull-up" menu item after which the IDE performs the actual modifications to the source code in order to execute the method pull up.

The idea of ChOP is that the entire software system is developed as illustrated above: by making the IDE execute changes. We believe, however, that pure ChOP is unrealistic as it is unlikely that a programmer will actually execute a dialog for every fine-grained modification (such as the addition of a statement to a method body). Therefore, we have proposed a *logging technique* which can be used to watch over the developer when taking development or evolution actions. For that, the IDE is instrumented with a logging mechanism responsible to instantiate change objects that represent all the actions the developers take. The advantage of the logging approach is that it seems more realistic than pure ChOP, as it does not require an adaptation of the development process. A drawback might involve privacy issues, as the programmers are required to allow "Big Brother" to watch them develop software.

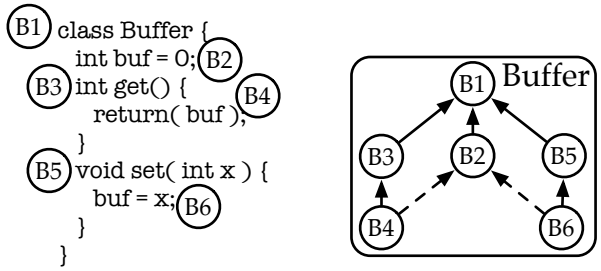

Figure 1: Buffer: source code (left), change objects (right)

Figure 1 shows both the source code (on the left) and the changes (on the right) of a `Buffer`. In order to obtain the changes, the developer does not need to alter his development process. He just writes source code as he is used to. Behind the scenes, the logger instantiates the change objects and preserves the link between every piece of code and the change objects that were involved in the creation of that piece of code. In Figure 1, this link is made explicit by annotations in the source code that refer to the change objects. Every change object is identified by a unique number: $B1$ is a change that adds a class `Buffer`, $B4$ is a change that adds an access of the instance variable `buf`.

In ChOP, every change has a set of preconditions that should be satisfied before it may be applied. Such preconditions are related to system invariants imposed by the programming language (usually defined by the meta-model of the language). For example, methods can only be added to existing classes. Preconditions enable expressing dependency relationships between changes. In the above example, for instance, the change that adds the method `get()` to the `Buffer` class depends on the change that adds the `Buffer` class to the system, as the latter creates the container to which the former adds.

More generally, a change object c_1 is said to depend on another change object c_2 ($c_1 \rightarrow c_2$) if the application of c_1 without c_2 would violate the system invariants. We come back to the dependencies in Section 4.

3. UNIT TESTING AND CHOP

In order to minimize the presence of defects in a software system, it needs to be tested. Testing uses a combination of input and state to execute the program code in order to reveal the presence of bugs in a system [3]. *Unit Testing* is a testing method in which a programmer tests the individual building blocks (units) of the program code. In object-oriented programming, unit testing is typically used to test individual classes.

A popular way of implementing unit tests is by using one of the frameworks collectively known as xUnit [12, Chapter 3]. These frameworks have a common design based on the one by Kent Beck for his implementation of SUnit, the unit testing framework for Smalltalk [2]. Since then, SUnit has been ported to several programming languages (e.g., JUnit for Java or CppUnit for C++). An xUnit framework implements the four phases of a test execution:

1. Set up – Setting up the test.
2. Exercise – Interact with the unit under test.
3. Verify – Determine whether the outcome is as expected.
4. Tear down – Tear down the test to return to the original state.

These phases are typically implemented in three methods: a `setup` method that handles the first phase, a `test` method to handle the second and third phases and a `teardown` method to handle the last phase. It is the task of the test developer to implement these methods whereas the xUnit framework makes sure that these methods are executed in the right order. One can write several `test` methods in a test. Each of these `test` methods will then be executed separately every time preceded by a call to the setup method and followed by a call to the teardown method.

The tests written for an xUnit framework are usually written in the same programming language as the units under test (e.g. a test in the JUnit framework is written in Java and is capable of testing Java code). As such, both the changes required for creating a test and the changes used for writing the program code, adhere to the same meta-model. We can therefore use the same logging technique as proposed in Section 2 to watch over the developer when he is writing test cases.

Figure 2 presents the test code of the `Buffer` from Figure 1. The corresponding changes produced by the logger are depicted in Figure 3. The link between changes and source

[1]In software engineering, refactoring source code means improving it without changing its overall results, and is sometimes informally referred to as "cleaning it up" [11].

```
T1 class BufferTest extends TestCase {
     Buffer buffer; T2
     void setUp(){
          buffer = new Buffer();
     }
     void tearDown(){}
T3   void testGet(){
          buffer.buf = 1337; T4 T5
     T6 assertEquals(buffer.get(), 1337); T7 T8
     }
T9   void testSet(){
          buffer.set(7331); T10 T11
     T12 assertEquals(buffer.buf, 7331); T13 T14
     }
}
```

Figure 2: Annotated source code of BufferTest.

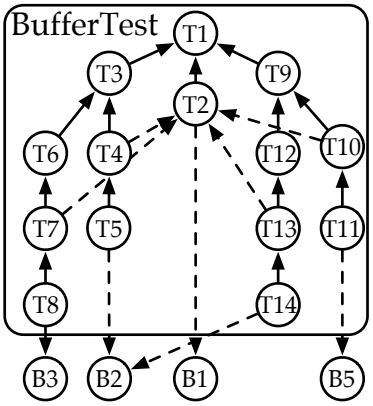

Figure 3: Changes required to create BufferTest.

code are made explicit by the logger in terms of code annotations. For simplicity's sake we opted not to annotate the `setUp()` and `tearDown()` methods. The changes for the `BufferTest` are identified by the unique numbers $T1$ - $T14$: $T1$ is the change that adds the test class `BufferTest`; $T2$ is the change that adds the `buffer` attribute; $T3$ and $T9$ are the changes that add a method to test the methods `get()` and `set()` of the class `Buffer`; $T4$, $T7$, $T10$ and $T13$ are the changes that add an access to the `buffer` attribute; $T5$ and $T14$ are the changes that add an access to `buf`; $T6$ and $T12$ are the changes that add an invocation to the `assertEquals()`[2] method; and $T8$ and $T11$ are the changes that add a method invocation to respectively the methods `get()` and `set()` of the class `Buffer`.

4. DEPENDENCIES

In previous work, we distinguished between different kinds of dependencies that exist between change objects [9]. *Structural* dependencies are implied by the meta-model and can consequently be logged automatically when the changes are instantiated. *Semantical* dependencies can only be derived from semantical information. This consists of programming and domain knowledge. In Figure 1, there are several structural dependencies: $B4$ depends on the change that adds the method `get` ($B3$) and on the change that adds the instance variable `buf` that it accesses ($B2$). The figure, however, does not contain any semantical dependency.

For the purpose of this paper, we focus on one particular kind of structural dependencies: *hierarchical* dependencies (denoted by the full arrows in Figures 1 and 3). A change c_1 is said to *hierarchically depend* on a change c_2 if the *subject* of c_1 is contained by the *subject* of c_2. The *subject* of a change is a building block of the programming language used to develop the software system. That language is specified by a meta-model describing the building blocks and the relations between them. One of those relations is the containment (e.g., "a class contains a method" or a "method contains a statement").

Consider for instance $B3$ from Figure 1. It adds a method `get()` to the class `Buffer` that was added by $B1$. As the `get()` method is contained by the `Buffer` class, $B3$ is said to hierarchically depend on $B1$. We denote the non-hierarchical dependencies by the dashed arrows in Figures 1 and 3.

The dependencies between change objects incorporate a link between the program code and the test code. For instance, the annotations within the source code allow us to automatically derive that the `testSet()` test is related to the `set()` method and vice versa. Concretely, we know that $T11$ (the change that adds an invocation of the `set()` method in `testSet()`) depends on $B5$ (the change that adds the `set()` method to the `Buffer`). We also know that $T11$ hierarchically depends on $T10$ (the change that adds an access of `buffer`) and that $T10$ hierarchically depends on $T9$ (the change that adds the `testSet()` method). This data allows us to establish a relation between the `set()` method and the `testSet()` test.

In general, a test is related to a software building block (i.e. classes, methods, instance variables) if at least one of the changes that were performed to create the test depends on the change creating the building block. We will come back to the dependencies in Section 6 in which we will exploit them for optimization purposes. But first, we explain how we envision pro-active debugging by means of the change objects.

5. PRO-ACTIVE DEBUGGING

The idea behind pro-active debugging is to retrieve bugs as soon as they are inserted in the software system. We are aware that avoiding all bugs in a pro-active way is a utopia. Nevertheless, in this paper we strive towards this by detecting some avoidable bugs in a pro-active way. The main idea behind our technique is to run all the relevant tests, whenever a small set of changes are applied to the software system. In order to demonstrate how ChOP can assist in doing that, we go back to the `Buffer` from Figure 1 and its corresponding tests from Figure 2. Consider that, due to changing requirements, the buffer needs to be extended with an *undo* functionality, which should allow a buffer user to undo the effect of a `set()` method, and consequently take the buffer back to its previous value. In order to implement this requirement, the developer adapts the buffer and obtains the

[2]The assertEquals method verifies that its two parameters are equal in value. If they are not, the test fails.

version in Figure 4.

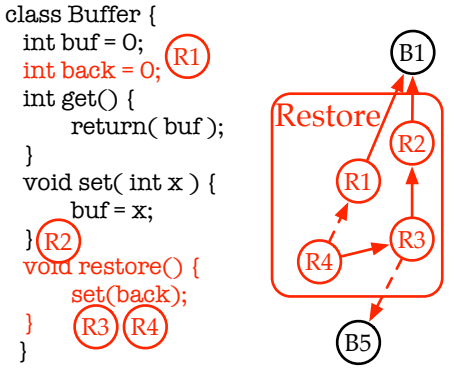

Figure 4: Restore: source code (left), change objects (right)

Every time a set of changes related to the implementation of the undo (e.g., $\{R1\}$, $\{R2\}$ and $\{R3, R4\}$) is produced, the existing tests (from Figure 2) are executed. None of them fails at any point. As the buffer was extended with a new method, however, the test set has to be adapted accordingly: It has to be extended with a `testRestore()` test, which tests the `restore()` method. It verifies that when calling the `restore()` method after the `set()` method, the value of `buf` is changed to the value that it was originally (see Figure 5 for the code and Figure 6 for the changes).

Some programming styles claim that the tests should be written before the program itself [1]. This does not pose a conceptual problem for our approach, but still requires an adaptation on the implementation level.

```
class BufferTest extends TestCase {
  Buffer buffer;
  void setUp(){
      buffer = new Buffer();
  }
  void tearDown(){}
  void testGet(){
      buffer.buf = 1337;
      assertEquals(buffer.get(), 1337);
  }
  void testSet(){
      buffer.set(7331);
      assertEquals(buffer.buf, 7331);
  }
S1 void testRestore(){
      buffer.buf= 1337; S2 S3
S4 S5 buffer.set(1338);
      buffer.restore(); S6 S7
   S8 assertEquals(buffer.buf, 1337); S9 S10
  }
}
```

Figure 5: Annotated source code for the testRestore method.

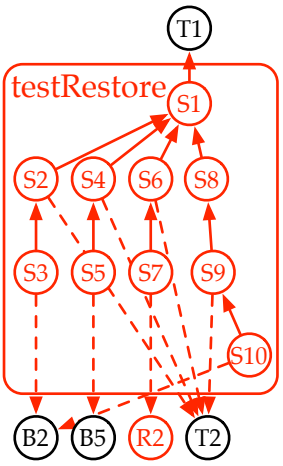

Figure 6: Changes required to create the testRestore method.

While implementing a test in a ChOP way, the tests are executed as well. Consequently, upon the application of $\{S8, S9, S10\}$ (the last set of changes changes for the `testRestore()`), the tests are also executed. At that point, the `testRestore()` fails, as the implemented behavior of the `restore()` does not correspond to the desired behavior: Instead of restoring the attribute `buf` to its original value, it is set to 0. Thanks to the dependencies between the changes related to the failing test and the changes related to the program code, we can point the developer to both the source code of the failing test and the program code related to that test. This is interesting, as a failing test might be caused by a defect within the program code or the test code.

Let us go back to the buffer example, and demonstrate how we find the interesting "spots" in both the test code and the program code upon the failing `testRestore()`. We start of from the annotated source code of the failing test and find all the changes related to the implementation of that test $\{S1, S2, S3, ..., S10\}$. We can find these as the annotations are explicitly kept in the source code. Afterwards, we use the dependencies to establish the transitive closure of all changes on which the changes in this set depend. In the buffer case, this process results in the set $\{S1, S2, S3, ..., S10, T1, T2, B2, B5, R2, B1\}$. The spots in the source code that relate to these changes are the possible targets for fixing the failing test. In our case we find that the failing test is related to the methods `set()` (which is created by $B5$), `restore()` (which is created by $R2$) and `testRestore()` (created by $S1$). We can therefore deduce that the bug that causes the test to fail is probably located in one of these methods. By adding a statement to the `set()` method, we fix our mistake and obtain the version in Figure 7. Upon the execution of $R6$, the test suite is executed again and all tests pass.

There are two advantages brought along by our approach. First, there is the time at which we can detect a bug. Running the tests (in the background) after the application of a small set of changes allows us to notify the developer of a failing test while he is coding. At that moment, the devel-

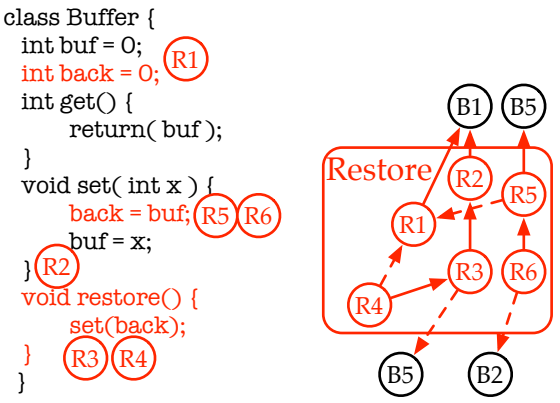

Figure 7: Corrected version of restore feature: source code (left), change objects (right)

oper is still in the right frame of mind, knows where he is programming and why he is programming that code. Second, there is the fine-grained information we can provide upon the failure of a test. This information (which is based on the fine-grained change objects and the dependencies between them) might unveil non-trivial relations within the program and test code, which can be used to detect the most interesting spots in both the test and program code for locating the bug. Both advantages relate to the detection of bugs, which we believe to be speeded up significantly thanks to our approach.

A disadvantage of this approach relates to the scalability. As the test suites and program code grow bigger, it becomes an intensive task to run all tests for every modification of the code. In order to overcome this problem, we now propose a way to optimize our pro-active debugging process.

6. OPTIMIZATION

The execution of the entire test suite upon the application of a simple set of changes might come with an unacceptable overhead (especially when the change sets and the test suites get bigger). In order to avoid such overhead, we propose to run only the relevant tests (= the tests that relate to the recent changes) instead of all of them. We can use the dependencies that exist between the change objects to retrieve all those tests. In the buffer example for instance, the $R6$ change of Figure 7 does not require the `testGet()` test to be re-evaluated. In general, we can say that we only require the tests to be evaluated that were affected by a change that depends on the building block that is affected by the applied changes.

To calculate which tests need to be run, we execute the following steps:

Step 1. Calculate the transitive closure of the changes on which the last applied changes hierarchically depend.

Step 2. Of this set, take the changes that added a method.

Step 3. Look for the changes that non-hierarchically depend on these changes.

Step 4. Calculate the transitive closure of the changes on which these changes hierarchically depend.

Step 5. For each change that adds a test method, add this test to those that need to be run.

Step 6. For each change that adds a method that is not a test, repeat Step 3. through Step 6.

In the buffer example, this principle boils down to the evaluation of the `testRestore()` and `testSet()` tests when the changes $R5$ and $R6$ are applied. First we calculate the transitive closure of the changes on which $R5$ and $R6$ depend and obtain $\{R5, R6, B5, B1\}$ (Step 1). The change $B5$ is the change that added the `set()` method (Step 2). For this change we look for the changes that non-hierarchically depend on it, by following the dashed lines "in the opposite direction". We find that the changes $T11$, $S5$ and $R3$ depend on $B5$ (Step 3). Next we calculate the transitive closure of the changes on which this set of changes depends and obtain $\{T11, T10, T9, T1, S5, S4, S1, R3, R2, B1\}$ (Step 4). In this set we have three changes that add a method: $T9$ adds the method `testSet()`, $S1$ adds the method `testRestore()` and $R2$ adds the method `restore()`. Thus we have found that we need to run the `testSet()` and `testRestore()` tests (Step 5). The algorithm however is not finished as we need to redo steps 3 to 6 for $R2$, the change that added the `restore()` method. We find that the change that adds an invocation to this method is $S7$ (Step 3). When we look for the changes on which $S7$ depends we find the set $\{S7, S6, S1, T1\}$ (Step 4). In this set there is only one change that added a method: $S1$, which is the change that added the `testRestore()` method. We find that the testRestore() test needs to be run.

This process follows a linear algorithm for retrieving the relevant tests, which takes longer if the change set grows. The detection process, however, supposedly takes less time than the time needed to actually run the non-relevant tests. We expect this profit to increase for growing change sets.

7. RELATED WORK

This work relates to several fields in the software development research domain: software testing, change reification, development assistance and debugging. While the former two fields incorporate technologies on which our approach is based, the latter two present related approaches that aim for improving the quality of the software. In this section, we explain the state-of-the-art in those fields and relate it with our work.

Software testing is the process of validating and verifying that a software program works as expected. We use software testing in order to detect defects (bugs) in the software [5]. More specifically, we use a unit testing technique [2], in which every test verifies the functionality of a small particle of the software. Most main-stream programming languages and IDEs have a corresponding unit testing framework which allows to apply unit testing.

Another testing technique that aims at pro-active debugging is that of *regression testing*. The goal of regression testing

is to validate that recently applied modifications did not introduce new errors in code that has previously been tested. As such it provides confidence that the modifications are correct [3, Chapter 15]. Running all regression tests can be a time-consuming process, therefore techniques have been proposed to select only the relevant tests as a way to reduce the run-time. These techniques are related to our optimization in Section 6. Especially *"Retest Within Firewall"*, which selects the tests to run by analyzing dependencies between parts of the system.

The idea of running the tests in the background while coding the system was previously proposed by Saff and Ernst in [**?**]. They developed a framework for continuous testing in Eclipse which uses the excess cycles of a developers workstation to run regression tests in the background every time a developer saves his changes. While similar on first sight, their approach is conceptually different from our approach as we test the software incrementally (upon the application of a change), while in their approach, the software is tested in snapshots (when the adapted source code is saved).

Another technology on which our approach is based, is the *reification of change* into first-class objects. Other uses for such reification has already been pointed out by other researchers. In [20] Robbes shows that the information from the change objects provides a lot more information about the evolution of a software system than the central code repositories. They later used this approach to detect the logical coupling between program entities [21]. In [7] Denker shows that first-class changes can be used to define a scope for dynamic execution and that they can consequently be used to adapt running software systems. In this section, we first explain a model of first-class changes and then show how to specify and compose features as sets of first-class changes. Both change models of Robbes and Denker are similar to our model. Our model, however, incorporates changes within the statement level, while that level is not supported by those models. This level of granularity is required, as it provides the detailed information we use to relate tests to source code and vice versa.

Our approach provides *development assistance* by revealing related parts of the software that are non-trivial to retrieve. Other approaches exist that have the same purpose. For instance Zimmermann et al. mine the version repositories of a system to suggest which parts of the code to adapt when similar modifications are applied (i.e., When files a,b, and c are often revised together, adapting files a and c might suggest that a modification to file b is also necessary.) [22]. The *TeamTracks* tool by DeLine et al. is another tool that tries to unveil hidden relations within the software building blocks. It tracks a programmer's navigation path on a system in order to suggest to other developers which parts of the program may be of interest based on their own navigation path [6]. Our approach seems complementary to both approaches. The comparison of Rastkar and Murphy already shows that there is no direct relationship between both, suggesting that they capture different aspects of unveiling hidden relations [17].

Our pro-active debugging technique is related to other approaches that target the debugging of software systems, such as *omniscient* debugging, *query-based* debugging, *delta* debugging and *rewind-based* debugging. Omniscient debuggers make it possible to navigate backwards in time in a program execution trace, targeting the task of debugging complex applications [13]. Query-based debugging consists of identifying events that match a query expressed in some high-level language [14]. Delta debugging is used to find the cause of a regression fault by looking at the differences between the old and the new code [**?**]. Rewind-based debugging can be achieved by taking the program back to a version which did not include the concerned bug. All four techniques seem to be usable in combination with our technique as they all target different facets of the debugging process.

8. FUTURE WORK

The approach presented in this paper is not implemented yet. Consequently, as a first track of future work, we envision the implementation of a proof-of-concept implementation, which supports our approach. This implementation will be based on a combination of sUnit and ChEOPS (Change and Evolution Oriented Programming Support). sUnit is a unit testing framework for Smalltalk, which is incorporated in the VisualWorks for Smalltalk development environment. ChEOPS is a tool that supports ChOP within the same development environment. Moreover, it provides the logging capabilities which we require and produces fine-grained first-class change objects that represent the development actions taken by the developer [8].

Once an implementation is available, we can start the evaluation of our approach. The first step in the evaluation consists in finding out whether or not the overhead for running the tests every time a set of changes are applied is acceptable. If that is not the case, we plan to implement the optimization sketched in Section 6. Afterwards, we need to evaluate that the optimization does not rule out relevant tests in practice, that the speed gain is worthwhile and that it makes the approach "workable".

The third track of future work consists in setting up a controlled experiment in which we test how our approach behaves in practice. What we want to know is whether or not the developers experience the development assistance we provide as useful. We envision an empirical study in which we have two groups of developers (all computer science students with the same formation) which have to evolve a software system that contains some bugs. While one group may use our approach throughout the development process, the second group may not. A comparison of the quality of the resulting software and the time used to obtain it, will answer questions related to the benefits of our approach with respect to software quality and development speed respectively.

A final track of future works consists of comparing and combining our approach to and with other related approaches that provide development assistance. We strongly believe that a combination of our approach with rewind-based debugging seems promising. The symbiosis between the change objects and the undo mechanism brings along a very fine level of granularity of rewind steps: The changes can be undone one by one until a version is obtained that does not include the bug. We expect this to speed up the retrieval of

the development action that introduced the bug.

9. CONCLUSIONS

The sooner a bug is detected, the better. Software testing is a technique that is used to speed up bug detection, and consequently lower the costs of fixing bugs. When all the tests are run, however, it is often difficult to locate the precise error that causes a test to fail. In this paper, we outline a technique that uses software testing to warn the software developer upon the introduction of a bug in the software system. It allows what we call *pro-active debugging*: detecting a bug before it actually becomes apparent in the program code. Pro-active debugging aims at minimizing the presence of bugs at a very early stage in the development process.

Our technique is based on change-oriented programming (ChOP): a programming paradigm that centralizes change as the main development entity. In ChOP a programmer does not need to write large streams of source code, but rather uses the integrated development environment to *apply changes* to his source code. Upon the application of a small set of changes, we propose to run the tests of a test suite. A failing test is an indication that the just-applied changes are causing a bug. As this technique allows us to notify the developer right away when a development action makes a test fail, we expect it to (a) decrease the probability of introducing bugs and (b) increase the speed of detecting bugs.

In this paper, we sketch how we plan to implement pro-active debugging and present a possible optimization technique which avoids the execution of non-relevant tests. These tests can be filtered away by using the dependencies between the change objects that were applied for obtaining the program code and the change objects that were applied for obtaining the test code. The same dependencies also allow us to provide the developer with fine-grained information about the interesting spots in both the program and test code that are related to a failing test.

10. REFERENCES

[1] Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[2] K. Beck. Simple smalltalk testing: With patterns. *The Smalltalk Report*, 4(2):16–18, October 1994.

[3] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[4] J. Brant and D. Roberts. Refactoring browser. Technical report, http://wiki.cs.uiuc.edu/RefactoringBrowser, 1999.

[5] L. Crispin and J. Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2008.

[6] R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 241–248, Washington, DC, USA, 2005. IEEE Computer Society.

[7] M. Denker, T. Gîrba, A. Lienhard, O. Nierstrasz, L. Renggli, and P. Zumkehr. Encapsulating and exploiting change with changeboxes. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 25–49, New York, NY, USA, 2007. ACM.

[8] P. Ebraert. First-class change objects for feature-oriented programming. In *15th Working Conference on Reverse Engineering proceedings*, pages 319–323. IEEE Computer Society, 2008.

[9] P. Ebraert. *A bottom-up approach to program variation*. PhD thesis, Vrije Universiteit Brussel, 2009.

[10] P. Ebraert, E. Van Paesschen, and T. D'Hondt. Change-oriented round-trip engineering. Technical report, Vrije Universiteit Brussel, 2007.

[11] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[12] P. Hamill. *Unit Test Frameworks*. O'Reilly, 2004.

[13] B. Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging*, 2003.

[14] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *ACM SIGPLAN Notices*, pages 365–383, 2005.

[15] S. McConnell. *Code complete*. Microsoft Press Redmond, WA, USA, 2004.

[16] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, 6 edition, 2004.

[17] S. Rastkar and G. C. Murphy. On what basis to recommend: Changesets or interactions? In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 155–158, Washington, DC, USA, 2009. IEEE Computer Society.

[18] C. Rich and R. Waters. *The programmer's apprentice*. ACM Press, 1990.

[19] C. Rich and R. C. Waters. The programmer's apprentice project: A research overview. 1987.

[20] R. Robbes and M. Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, pages 93–109, 2007.

[21] R. Robbes, D. Pollet, and M. Lanza. Logical coupling based on fine-grained change information. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 42–46, Washington, DC, USA, 2008. IEEE Computer Society.

[22] 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*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.