Norwegian University of
Science and Technology

Random Testing versus Partition Testing

Kristian Oftedal

Master of Science in Informatics
Submission date: May 2011
Supervisor: Tor Stålhane, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

In the Software Engineering discipline of testing it has for the past 30 years been a discussion on the effectiveness of random testing compared to the effectiveness of partition testing. A new article by D. Rombach et al. claims that partition testing is more efficient, while there is a statistical model that shows the two strategies to be equal. The assignment consists of experimenting with the two test strategies on a program with known errors and analyze the results.

Abstract

The difference between Partition Testing and Random Testing has been thoroughly investigated theoretically. In this thesis we present a practical study of the differences between random testing and partition testing. The study is performed on the open-source project Buddi with JUnit and Randoop as test tools. The comparison is made with respect to coverage rate and fault rate. The results are discussed and analyzed. The observed differences are statistically significant at the 10% level with respect to coverage rate, in favour of partition testing, and not statistically significant at the 10% level with respect to the fault rate.

Contents

1 Introduction and Problem Statement **1**
1.1 Introduction 1
1.2 Problem Definition: 1
1.3 Outline 2

2 State of the Art **3**
2.1 State of the Art Overview 3
2.2 Test Techniques 3
2.3 Partition Testing versus Random Testing 14
2.4 Testing Tools 15

3 Experiment **23**
3.1 Experimental Subject 23
3.2 Test Metrics 27
3.3 Manual Category-Partition testing 28
3.4 Randoop Testing 29

4 Test results **33**
4.1 Overview of Test Results 33
4.2 Coverage Results 33
4.3 Failures Found 36

5 Data Analysis **43**
5.1 Paired T Test Coverage Results 43
5.2 Paired T Test Fault Results 44
5.3 Comparison with Another Study 46

5.4 Threats to Validity 47
5.5 Assumptions of Theoretical Research 48

6 Conclusion **49**

A Buddi Specifications **55**
A.1 Specifications 55

B Primes **61**

C Test Specifications Buddi ModelImpl Package **63**
C.1 Account 63
C.2 Account Type 65
C.3 BudgetCategories 66
C.4 Day & Time 68
C.5 DocumentImpl 69
C.6 FilteredLists 75
C.7 ModelFactory 79
C.8 ScheduledTransactionImpl 82
C.9 SourceImpl 84
C.10 Transaction 84

1 Introduction and Problem Statement

1.1 Introduction

Software Testing is performed to give confidence that the software under test (SUT) behaves as expected and to assess the SUT's quality by detecting defects and failures. The impact of faulty software may be the loss of lives, as in the Therac-25 case [22], or an accidental financial loss [27]. "Testing shows the presence, not the absence of bugs." [26] to quote Dijkstra opinion about testing. But proving the correctness of a program may be costly if not impossible [37]. Therefore, the most common testing activity is to write tests that verify a certain level of quality in the software. The research community has come up with many testing strategies and tools to perform testing. A significant question is if one strategy is more effective than another. Particular, many theoretical and statistical comparisons have been performed between the Random test strategy and the Partition test strategy, but the question if random testing is more efficient than partition testing have not been tested empirically yet. In this thesis I will compare random testing with partition testing implemented in testing tools. I have chosen to test tools for the object-oriented language Java and I will test an open-source project. Bacchelli et. al. [5] has done a similar study, but my focus will be the nature of the testing strategy instead of automatic testing versus manual testing.

1.2 Problem Definition:

How effective is random testing compared to partition testing with respect to coverage rate and fault rate?

1.3 Outline

The outline of this thesis is the following : Chapter 2 contains a summary of a discussion from the research community regarding the effectiveness of random testing versus the effectiveness of partition testing. A brief discussion of the two testing methods is also provided. The chapter also contains a discussion of available testing tools. Chapter 3 discuss the experiment. Results are analyzed in chapter 4 and the conclusion is provide in chapter 5

2

State of the Art

2.1 State of the Art Overview

In this chapter a survey of related works from the software engineering literature is provided to give a picture of how this experiment is related to previous works. First a description of software testing techniques will be given before a summary of the partition testing versus random testing discussion. Relevant tools to the testing techniques and the experiment are also discussed.

2.2 Test Techniques

2.2.1 Black-Box Testing

In Black-box testing, test cases are designed based on external properties of the SUT that the testers can find without taking the SUT's implementation into account. This can either be through inspection of the specifications belonging to the SUT [12] or through execution of the functions followed by an analysis of their input and output data, as in functional testing [18]. The advantage of black-box testing is that the program is tested according to how the SUT is expected to behave. For example, if a function called addInteger(int a, int b) was black-box tested, the tester could input the pair (1,2) and validate that the output to be 3.

Examples of black-box testing include partition testing [37], which can also be a white-box technique, category-partition testing [31], random testing, and boundary value analysis [25]. This thesis will be concerned with category-partition testing and random testing. A description of category-partition and random testing will be given below.

Figure 2.1: SUT treated like a black-box

2.2.2 White-Box Testing

White-box testing is the opposite of black-box testing. Instead of taking the specifications into consideration and treating the SUT as a black-box, the goal of white box testing is to examine the structure of the SUT and design the test cases based on this information. The white-box test cases are constructed to test the SUT exhaustively by executing all branches, paths and statements at least once. An advantage with structural testing is that you will know which part of the code that fails, which is not possible with black-box testing. Examples of white-box testing techniques are control flow testing, path analysis [19] and coverage testing (path coverage, statement coverage, statement coverage). Coverage testing is a technique where we identify which parts of a SUT a test suite actually executes. This thesis will use statement coverage as a test metric.

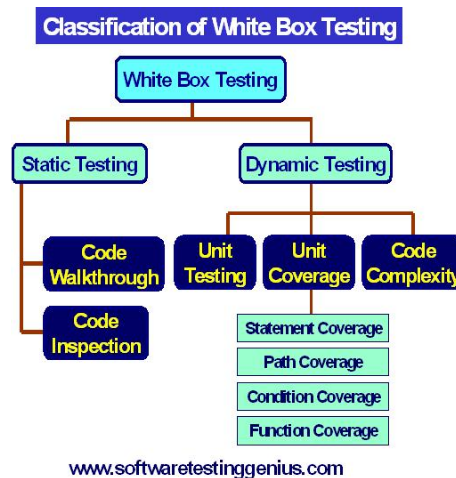


Figure 2.2: Overview of White-box testing

2.2.3 Partition Testing

In partition testing, which is considered both a white-box and black-box technique, the SUT's input domain is divided into classes or partitions. Ideally these partitions are mathematically disjoint. To define these partitions a tester will have to utilize all available information about the SUT, as done in the structural testing technique [14]. Partition testing was first introduced by Weyuker and Ostrand as a "revealing subset of the input" [37]. They proved that if an input from a revealing subset produced an incorrect output of the program, then any input from that distinct subset would produce an incorrect input. For example, in branch testing the input of the program is divided into subdomains, were a particular subdomain executes a branch of the SUT. [36].

2.2.4 Category-Partition Testing

Category-partition testing provides a systematic method to generate tests and can be best summarized by the abstract of the original article [31]:

> *"A method for creating functional test suites has been developed in which a test engineer analyzes the system specifications, writes a series of formal test specifications, and then uses a generator tool to produce test descriptions from which test scripts are written. . . "*

The following are the main steps of category-partition testing [31]:

Analyze Analyze the specifications. Identify functional units. Determine the parameters of the functional units.

Categorize Find characteristics of the parameters and possible environmental biases, which will be the categories.

Partition Determine boundary values and possible situations that might occur and partition these into choices that represents values of equivalence classes.

Constraints Decide how the choices interacts and if the choices could restrict each other.

Test specification Generate a formal test specification based on the categories, choices and constraints. P Test frames is then produced
based on the test specifications.

Evaluate Determine if the tests produced are sufficient or if some changes should be applied. For example, the absence of a constraint or a too larger number of test were produced

Test Cases Convert the test frames into test cases and produce test scripts that covers all the choices.

Illustration:

Below is a basic specification for a function called find. Find takes as input a pattern and a file and will search for instances of the pattern in the provided file [31].

Specification:

Command
find

Syntax
find <pattern> <file>

Function
The find command is used to locate one or more instances of a given pattern in a text file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, regardless of the number of times the pattern occurs.

The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes (“). To include a quotation mark in the pattern, two quotes in a row(“”) must be used.

Example Input

```
find john myfile
```

displays lines in the file **myfile** which contains **john**.

```
find "john doe" myfile
```

displays lines in the file **myfile** which contains **john doe**.

```
find "john ""doe"" myfile
```

displays lines in the file **myfile** which contains **john “doe”**.

The following properties can be identify about the input parameter [31] :

- pattern length;
- pattern enclosed in quotes;
- embedded blanks in pattern;
- embedded quotes in pattern.

In this example, file is the environment and the following characteristics have been identified:

- number of occurrences of pattern in file;

- number of occurrences of pattern in a target line, which is the line that contains the pattern;
- maximum line length in the file.

A test specification is then developed based on the characteristics. The test specification below contains properties, constraints and selection expressions within the [] notation. This ensures that no test frame will have contradictory properties [31].

Formal Test Specification for the find command:

Parameters:

Pattern size:

empty	[property empty]
single character	[property NonEmpty]
many characters	[property NonEmpty]
longer than any line in the file	[property NonEmpty]

Quoting:

pattern is quoted	[property Quoted]
pattern is not quoted	[if NonEmpty]
pattern is improperly quoted	[error]

Embedded blanks:

no embedded quotes	[if NonEmpty]
one embedded quote	[if NonEmpty & Quoted]
several embedded quotes	[if NonEmpty & Quoted]

File name:

valid file name	
no file with this name	[error]
omitted	[error]

Environments:

Number of occurrences of pattern in file:

none	[if NonEmpty] [single]
exactly one	[if NonEmpty] [property Match]
more than one	[if NonEmpty] [property Match]

Pattern occurrences on target line:

Assumes line contains the pattern

one	[if Match]
more than one	[if Match] [single]

Below is a potential test frame that includes choices from the test specification. Since the choices *NonEmpty* and *Quoted* have been selected, the choices *several embedded blanks* and *one embedded quote* are allowed. In the file environment this allows for the choices *exactly one* which means that the pattern also occurs on a line.

Test Frame:

Test Case 28: (Key = 3.1.3.2.1.2.1.)

```
Pattern size : many characters
Quoting : pattern is quoted
Embedded blanks : several embedded blanks
Embedded quotes : one embedded quote
File name : valid file name
Number of occurrences of pattern in file : exactly one
Pattern occurrences on target line : one
```

find command to perform test:

```
find "has " " one quote" testfile
```

2.2.5 Random Testing

Random testing is a test technique where test cases are chosen at random from the input domain of the program. As explained above, random testing uses a minimum of knowledge about the SUT and is therefore considered to be a black-box testing technique. Sometimes an unsystematic approach to selecting test data is appealing [16]. It could be the case that a systematic plan is hard to develop [15] or too expensive to perform functional or structural testing and therefore a random test approach is preferred.

To increase its relevancy, Random Testing may choose the input according to an operational profile. An operational profile is a list of operations and the probability of the operation taking place. The operational profile concept was introduced by Musa et. al. [24] in 1987 to make the testing emphasize realistic usage patterns of the SUT. This is important because most programs do not have an uniform distribution of its input and usage. A classic example of why testing should be done according to an operational profile is the Ariane 5 accident [23]. The Ariane 5 rocket exploded 39 seconds into its flight. An operand error occurred because of a 64-bit floating point to a 16 bit integer conversion was done with an unexpected high value. Much of the software used in the design was taken from the previous Ariane 4 rocket, but the early trajectory of Ariane 5 differed from that of Ariane 4 [23]. One of the findings of the investigation committee was that the software had not been tested with the actual

trajectory of Ariane 5 [23]. If the software had been tested with a operation profile of Ariane 5, the accident could probably have been avoided.

Figure 2.3: The Ariane 5 explosion

It has been in pointed out several times in the literature that obtaining an operational profile often is infeasible. But most of these papers are from the mid 90's and much have happened in software engineering since then. The traditional waterfall method has in many cases been replaced by agile methodologies [1], where small incremental parts of a particular system are released as often as every week. This means that the tester have a much higher chance of obtaining usage patterns and provide an extensive operational profile towards the end of the project.

2.2.6 Example of Path Coverage Testing

To give an example of the partition testing strategy path coverage, consider the following program: The program "midval" in figure 2.4 takes as input three integers and then computes the middle value of the triple. The program has four different execution paths and table 2.1 gives an example of four test cases that will cover the four execution paths. Figure 2.5 shows the paths of the program.

```
public int midval(int x, y, z) {
    if (x > y)
        if (y > z)
            return y;
        else
            return x;
    else
        if (x > z)
            return z;
        else
            return y;
}
```

Figure 2.4: Implementation of midval

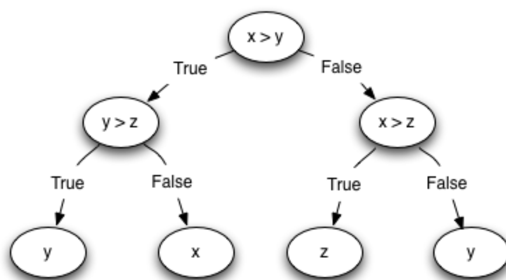


Figure 2.5: Paths of midval

```
1. int x=3, y=2, z=1
2. int x=3, y=2, z=4
3. int x=2, y=3, z=2
4. int x=2, y=3, z=4
```

Table 2.1: Possible test cases

In the third test case the program will encounter an error, as it will compute y to be the middle value of the triple. This error was not difficult to detect with path coverage testing. If we on the other hand would have tried to perform random testing on the program the probability of detecting this error would be low. In fact, if no operational profile was available and no other knowledge about the program would be taken into account the efficiency would be dependent on the intervals used to generated random inputs. Given that the range is set to [1, 100], the probability to generate three numbers with the same properties as test case three :

$$\{x \mid x >= 1 \text{ and } x <= 100, y \mid y >= 1\ y <= 100 \text{ and } y \neq x, z \mid z >= 1 \text{ and } z <= 100 \text{ and } z = x\}$$

would be 0.0099.

However, it would be more complex to generate a random input if the input to a function was for instance a data structure. Consider, for example, generating an input to the function:

```
Graph shortestPathAlgorithm(Graph graph, Vertex source) {
    for each vertex v in Graph:
            dist[v] := infinity ;
            previous[v] := undefined ;
    end for ;
    dist[source] := 0 ;
    .
    .
}
```

A possible approach would be to generate random binary random representations, for example:
Graph as 0001110011110101110011001110110110100101010100001111011110101 and Vertex as 1011011011101001100110011 and hope that the representations are legal instances of the inputs Graph and Vertex and that Vertex indeed exists in Graph. Another would be to identify properties of the data structure that could be generated randomly, but this would give no guarantee of a uniform sampling of the input domain.

2.2.7 Example of Random Testing

To give an example of random testing, consider the following program: The program "MortageCalc" computes the expected mortgage to a person based on the gender, age and salary of that particular person. Table 2.2 contains the specification to the program and Figure 2.6 contains a possible Java implementation:

Category	**Male**	**Female**
Young	(18-35 years) 75	(18-30 years) 70
Middle	(36-45 years) 55	(31-40 years) 50
Old	(46-55 years) 30	(41-50 years) 35

Table 2.2: Mortgage program specification

```
public int MortgageCalc(boolean gender, int age, int salary) {
    if (gender) {
        if (18 <= age && age < 35)
            return 75 * salary;
        else if (36 <= age && age < 45)
            return 55 * salary;
        else if (46 <= age && age < 55)
            return 30 * salary;
    } else {
        if (18 <= age && age < 30)
            return 75 * salary;
        else if (31 <= age && age < 40)
            return 50 * salary;
        else if (41 <= age && age < 50)
            return 35 * salary;
    }
    return 0;
}
```

Figure 2.6: Mortgage program implementation

At first this seems to be a situation where partition testing would be the ideal test strategy. But in this case, selecting boundaries could be more difficult. For age you could try 17, 18, 34 and 35 for instance. Salary boundary values of positive integers within a reasonable range could be suggestions to define an artificial boundary. With gender there are just two different values to choose between. However, since there is a dependency between age and gender a more fine grain partition would be necessary. So we have identified four candidates for the two first inputs. In addition we should include boundary values for the Middle and Old categories in the specification. If we also include some nominal values, the total of test cases would be the results in table 2.3, but it might not be necessary to cover all the paths:

Parameter	**Categories**	**Cases per category**	**Total**
Age	3	4	12
Salary	3	4	12
Gender	2	24	48

Table 2.3: Test cases needed for proper coverage

Because of the dependencies between the parameters and combinations available at least 48 test cases would be necessary to test all the boundaries. To manually design 48 different test cases would not be efficient and a random generation of the input

values would be better. It is assumed that the generator would allow you to specify an upper bound, as in Java Random nextInt(n). The results would then have to be validated by the 'Specification Oracle'. The Specification Oracle determines whether a test is correct or not. In the literature it is always assumed that an oracle is available. However, in practice this is often not the case [16].The only program that would produce the correct output that program is in fact the program itself. There exists the possibility of a manual inspection the results, but this would compromise the original advantage of automated generation of test data versus a manual generation [16]. In the case described above, however, a manual inspection would not be overwhelming.

If we continue the example with the partition testing we will notice that it will fall short, because it only tests code that is already part of a program. But what if there is a branch missing or a statement missing or the branches are wrong [28]? You will not be able to test for this because you are not looking for it. In the example above, there is not a path to cover ages that does not fit with one of the categories. Instead they will be wrongly assigned the value of the Old category.

Customer			
Sex	Age	Salary	Occurrence Probability
Male	16-35	100k-200k	0.08
Female	16-30	100k-200k	0.09
Male	16-35	200k-400k	0.13
Female	18-30	200k-400k	0.12
Male	36-45	200k-400k	0.23
Female	31-40	200k-400k	0.21
Male	46-60	200k-400k	0.07
Female	41-55	200k-400k	0.06

Table 2.4: Potential operational profile

Table 2.4 shows a potential operation profile for MortageCalc. It can be stated that MortageCalc has constant failure rate θ. We can than compute that MortageCalc will succeed with $1-\theta$ probability. If tested N independent times with respect to the operation profile the success probability will be $(1-\theta)^N$ and the probability of at least one observed failure would be $p = 1-(1-\theta)^N$[16]. In $1/\theta$ runs, the confidence probability is that at least $1-\theta$ failures will be observed. This is also known as the mean time to failure (MTTF)[16] and it solves to:

$$\frac{1}{\theta} \geq \frac{1}{1-(1-p)^{1/N}}$$

To assess confidence $1-p$ the number of tests required with the MTTF is

$$\frac{\log(1-p)}{\log(1-\theta)} \approx \frac{p}{\theta} \text{ for small values of } p \text{ and } \theta$$

If for example N = 1000 and MTTF is 300, it can be computed that $1-p \approx .98$.

2.3 Partition Testing versus Random Testing

In the Software Engineering discipline of testing it has for the past 30 years been a discussion on the effectiveness of random testing compared to the effectiveness of partition testing. In his book, "The Art of Software testing" Myers [25] claims that random testing is the least effective testing methodology, because you will not able to select an optimal set of test data [25]. This statement was followed by a number of papers during the 80's favoring random testing to partition testing. In 1984 Duran and Ntafos showed that "random testing can find a lot of subtle errors without a great deal efforts" [10]. The effort required to generate test data can be high with partition testing [10]. A critique of partition testing also came from Hamlet [14], who through a combination of statistical analysis and theoretical proofs showed that random testing could be more efficient than partition testing given that the domains with high failure rate could not be identified [14]. Similar results were also shown by Weyuker and Jeng [36]. Proportional partition testing [7] were suggested to overcome some of the problems shown with partition testing and in 1994 T. Y. Chen and Y. T. Yu found that partition testings worst case could be as good as random testing if the number of test cases picked within a domain were proportional to its size [7]. A problem with this strategy was shown by Ntafos [29], since the number of test cases needed to maintain the proportion could reach an infeasible size [29]. Gutjahr [13] went over Weyuker and Jeng's model and used their approach to show that partition testing would always be a better choice than random testing [13]. Endres and Rombach [11] then stated this as a law in their book "A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories" [11]. The efficiency of random testing has also been revisited as late as 2010 [4].

A common property shared by all of these papers is that they are theoretical research with a basis in theoretical and statistical analysis. For example Duran and Ntafos "consider programs in which the domains of error do not intersect" and assume that the operational profile of the program under test is available[10]. Further Duran and Ntafos assume that the ideal partition of choice would include one error in each sub-domain [10]. Overall, a lot of assumptions that makes the theoretical research easier to perform is done. Often it will be difficult to implement these in practice. Chan, Chen, Mak and Yu[6] points out that the results found in Weyuker and Jeng's paper[36] could be difficult to apply in practice because a program's input is not often dividable into equal-sized partitions [6]. As mentioned in Wohlin [38], software engineering is subject to human influences and therefore you cannot find any formal rules or laws. At the end of the experiment I will assess if the SUT inhabits any of the assumptions made in the literature.

2.4 Testing Tools

2.4.1 JUnit

JUnit is a unit testing framework used to write regression tests. Regression testing is performed to determine that changes done to a system does not introduce new errors [25].. The programmer specifies classes to test, input of the test and the expected output. The actual output is then usually compared to the expected output and the success of the test is determined. Many of the tools and frameworks described below implements JUnit in some way. Figure 2.7 shows an example of JUnit test script and 2.8 shows the test validation output.

```
public void testGetBalanceDate() {
    Account a = new Account()("Test", AccountType at);
    Date startDate = DateUtil.getDate(2007, Calendar.JUNE, 3);
    a.setStartDate(startDate);
    assertEquals(startDate, a.getStartDate());
}
```

Figure 2.7: Example of JUnit test script

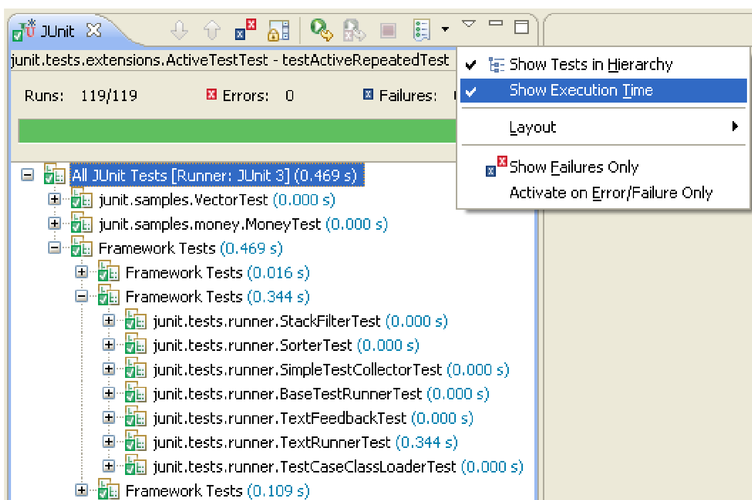


Figure 2.8: Example of JUnit test validation output

2.4.2 JCrasher

JCrasher [9] provides automated random testing of Java classes. JCrasher takes as input the Java class under test in byte code and produces possible test suites. The byte code is analyzed using the java reflection API. JCrasher analyses the class, methods and identifies the method parameter types and return types. When JCrasher has determined the number of test cases to generate, JCrasher writes the test cases as JUnit tests. Illegal inputs are collected at runtime by JCrasher and are Exceptions that the test cases throw. Thus, JCrasher is a robustness tester that will try to crash the program under test. The fact that JCrasher only recognizes exceptions as illegal inputs is a weakness of the testing approach since it will not be able to detect illegal outputs according to the specification of the SUT. Also JCrasher does not support regression test, as all of its test are generated at random and therefore repeating a test suite is very unlikely [9].

2.4.3 Randoop

Randoop [32] is an automated testing tool that performs Feedback-directed random test generation. In Feedback-directed random test generation new tests are generated based on the feedback given from previously computed values to ensure that the next tests to be produced contains new and legal inputs and states of the class [33]. Randoop generates sequences of random method calls and test them against exceptions and contracts that the tester optionally can implement [32]. Contracts are specified with a special contract interface called @CheckRep. @CheckRep allows you to specify a property that should hold for a class under test such as a post-condition or an object invariant [33]. This is implemented as method, boolean check(Object o), that will return true or false. For example, for a sorted list you could check that the list actually was sorted[32]. Pacheco et. al. proved that Feedback-directed random testing outperformed systematic and undirected random testing [33]. Randoop scales well and achieves the same coverage as systematic approaches [32]. Randoop provides two different type of test generation. The first is regression testing. This means in Randoop case that once you have generated a set of regression tests you can keep them. The other type is fail testing. In fail testing, Randoop will test potential @CheckRep methods mentioned above and a set of default contracts. Table 2.5 contains the default contracts that Randoop will check for.

Data type	Values				
`byte`	-1	0	1	10	100
short	-1	0	1	10	100
int	-1	0	1	10	100
long	-1	0	1	10	100
float	-1	0	1	10	100
double	-1	0	1	10	100
char:		’#’	’ ’	’4’	’a’
java.lang.String				””	”hi!”

Table 2.6: Overview of Randoop primitive values pool

Name	**Expected behaviour**
Equals to null	o.equals(null) should return false
Reflexity of equality	o.equals(o) should return true
Symmerty of equality	o1.equals(o2) implies o2.equals(o1)
Equals-hashcode	o1.equals(o2) == true, then o2.equals(o1) == true
No null pointer exceptions	No NullPointerException is thrown if no null pointers are used in test

Table 2.5: Contracts of Randoop

To make the most of Randoop, an operation-profile must be implemented in the classes. In Randoop this is called contracts [32]. A contract specifies a property which is either an object invariant or a post-condition to a method [32]. Also Randoop is “pseudo random” in its true nature. Unless specified, Randoop will choose values from a pool. Most of these values are boundary values. Table For example the int values are -1, 0, 1, 10, & 100.

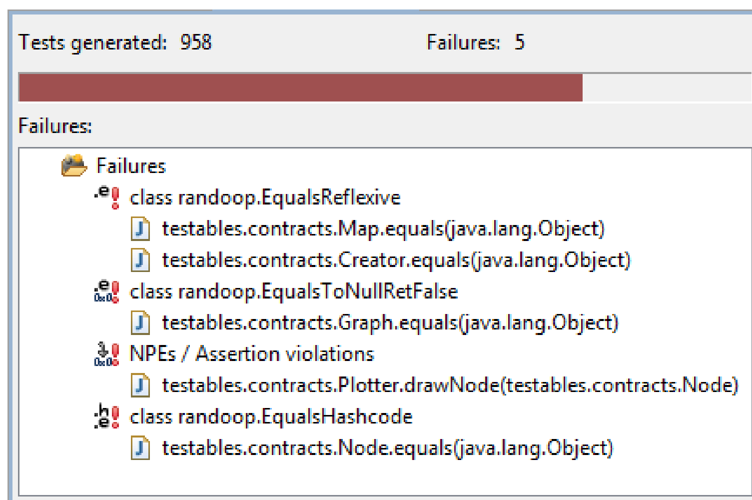


Figure 2.9: Example of Randoop output

2.4.4 JWalk (Extended)

JWalk (Extended) is an automatic category partition tool [20]. It is based on JWalk [35], which is a lazy semi-automatic systematic testing tool. The JWalk extension includes a category partition technique that partitions Java methods according to arguments of the method and chooses appropriate values for the arguments. JWalk also makes use of the Java Reflection API to analyze the code during execution. However it will not output any JUnit tests.

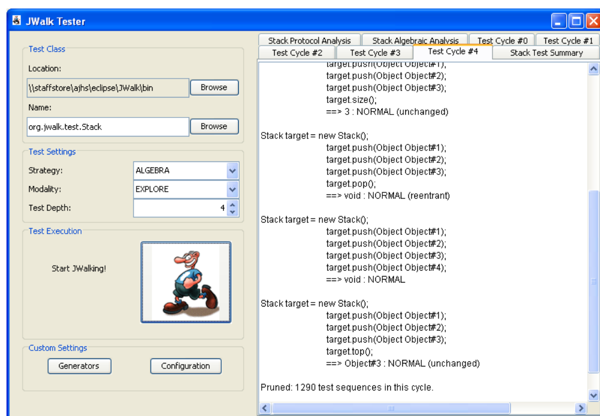


Figure 2.10: Example of JWalk output

2.4.5 EclEmma

EclEmma is an Eclipse plug-in of the open-source coverage testing tool Emma. Emma measures the statement, branch, loop and basic block of the supplied jUnit tests [8]. It will instrument .class files and output the results in a report [8]. Instrumentation is the process of manupulating code by injecting reporting code a certain positions of the code you are testing. Basic block coverage is an aggregation of a sequence of non-branching statements.

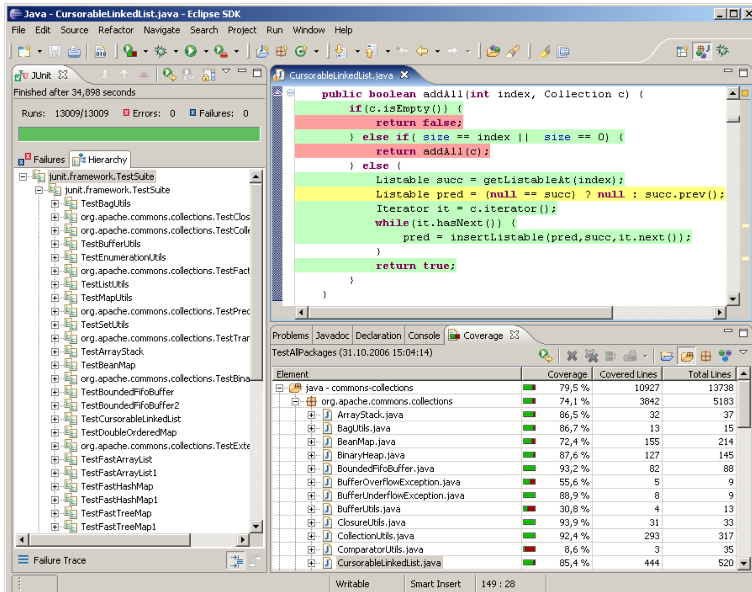


Figure 2.11: Example of EclEmma output

2.4.6 FindBugs

FindBugs [17] is a static analysis tool that inspects software for bug patterns. Hovemeyer defines bug patterns as "*error-prone coding practices that arise from the use of erroneus design patterns, misunderstanding of language semantics, or simple and common mistakes.*". Static analysis is the process of reviewing code without actually executing it. Techniques include data flow analysis which is the tracing of data through the methods and classes and model checking. Through what is called a *bug pattern detector*, FindBugs looks for bug patterns suggested by books [3], bug patterns observed and bug patterns suggested by the users [17]. FindBugs could be used to assess bugs located in the SUT.

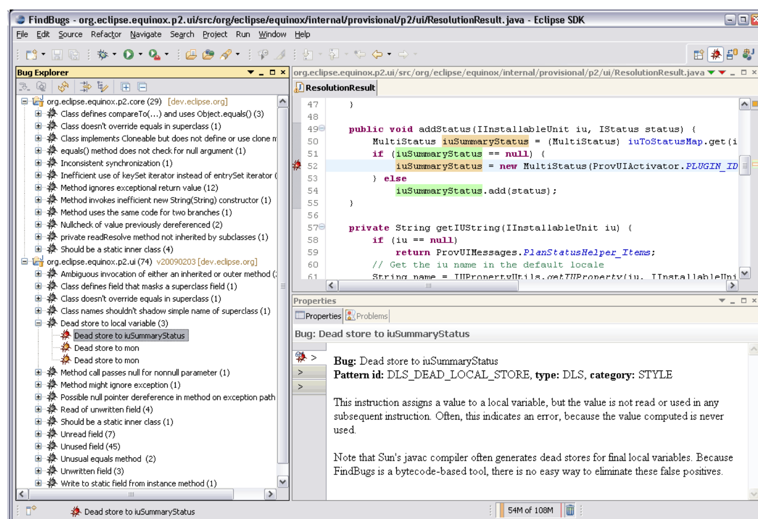


Figure 2.12: Example of FindBugs output

2.4.7 MuJava

MuJava [39] is an automated object oriented software mutation testing tool. It takes Java class files as input and mutates them. MuJava will mutate on the intra-method level, which will inject faults in the implementation of functionality in methods, and the inter-class level, which is injection of faults in the integration of classes. Traditionally Mutation testing is performed to assess the quality of the test cases, or how many mutant it kill. The potential use of MuJava in the experiment is to seed faults into the program.

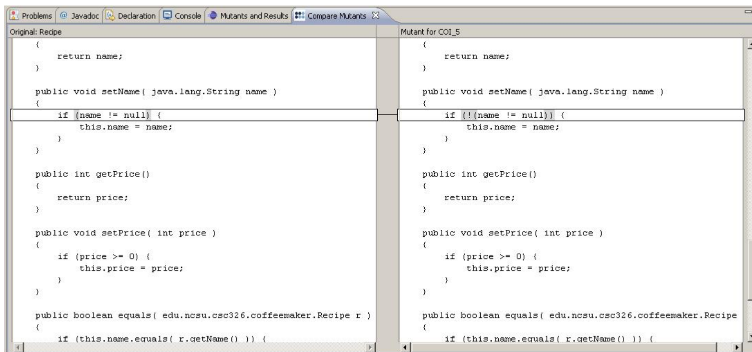


Figure 2.13: Example of mutant created by MuJava

3 Experiment

3.1 Experimental Subject

"Buddi is a personal finance and budgeting program, aimed at those who have little or no financial background" [30] and it is released under the GNU General Public License [30]. Buddi won the about.com readers' choice award, Best Mac Personal Finance Software 2011 [2]. The main properties of Buddi are the following:

- Allow user to create accounts to store money. The sum of money in accounts is the net worth.
- Associate accounts with Budget Categories to track the money flow and what you plan to spend. Categorize the money flow as income or expenses and specify the budget category the flow belongs to.
- Record the source and sink of the money flow with transactions. For example, specifying which account you payed a bill with.
- Define repeating transactions with scheduled transactions. For example, defining the day of the month you receive your salary.
- Transactions and account balances may be saved and loaded for later use.

This open source product was chosen for a number of reasons. First of all with its 20k LOC the project is of a medium size, which meant it would reduce the spent time studying the source code. Buddi is also a popular application with around one million downloaders and the development community is active. The bug tracker latest update was provided the same day as the decision to select the project (08.12.10) was taken. In appendix A the following documentation about Buddi is provided: an UML model, data model and some basic requirements of the program is provided. The consists

of 28 packages. I have therefore narrowed the scope of the experiment down to the model implementation package. The reason for this is that the classes contained in this package is decoupled from the rest of the system, which will help isolating the functions and test them. It is also the largest package with 3k LOC. Table 3.1 shows the classes of the package. Figures 3.1, 3.2 and 3.3 shows the main screens of Buddi.

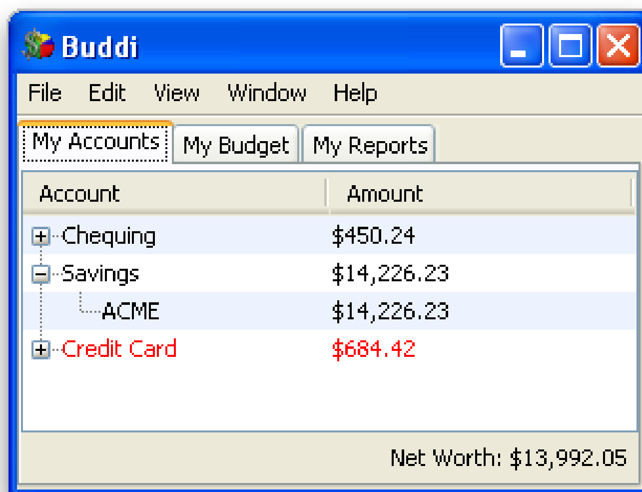


Figure 3.1: Overview of accounts

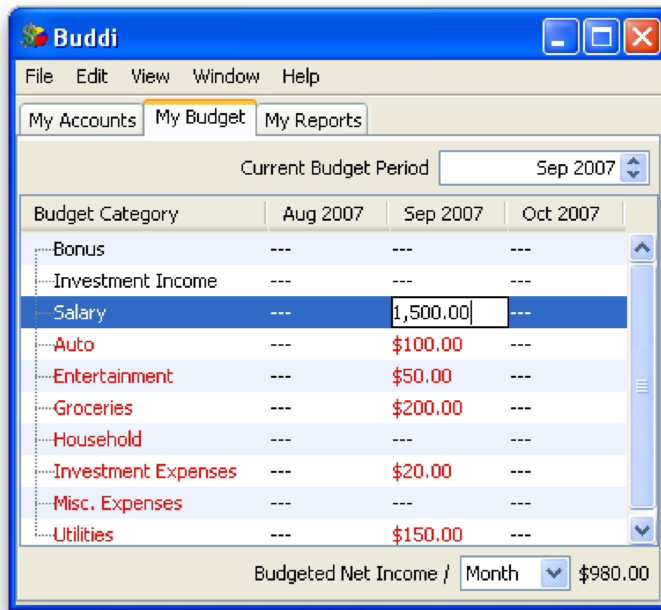


Figure 3.2: Overview of budget categories

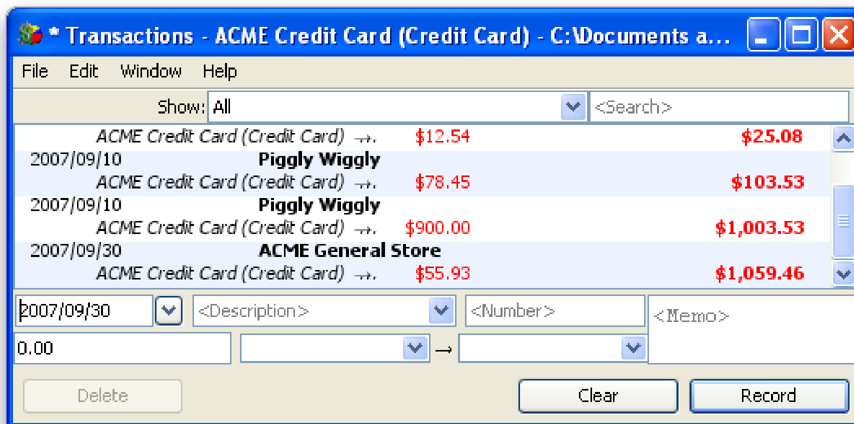


Figure 3.3: Overview of transactions

Name
AccountImpl
AccountTypeImpl
BudgetCategoryImpl
BudgetCategoryTypeMonthly
BudgetCategoryTypeQuarterly
BudgetCategoryTypeSemiMonthly
BudgetCategoryTypeSemiYearly
BudgetCategoryTypeWeekly
BudgetCategoryTypeYearly
ConcurrentSaveException
Day
DocumentImpl
FilteredLists
ModelFactory
ModelObjectImpl
ScheduledTransactionImpl
SourceImpl
SplitImpl
Time
TransactionImpl
TransactionSplitImpl

Table 3.1: Classes in the model implementation package

3.2 Test Metrics

3.2.1 Independent Variables

- Random Testing
- Partition Testing

3.2.2 Dependent Variables

- Coverage rate
- Fault rate

3.2.3 Failure Definition

Failures have many definitions in the literature, but I will settle for the definition given by Laprie: "A system failure occurs when the delivered service deviates from the specified service, where the service specification is an agreed description of the expected service." [21]

3.2.4 Statement Coverage Rate Calculation

The most common formula to calculate statement coverage for component i is the following:

Number of executable statements executed = n_i
Total number of executable statements = t_i

$$\text{Coverage } c_i = \frac{\mathrm{n}_i}{\mathrm{t}_i} \times 100$$

Which means that:

$$n_i = \frac{\mathrm{t}_i \times c_i}{100}$$

In this experiment I will use the following formula to calculate the total statement coverage.

$$c = \frac{\sum t_i c_i / 100}{\sum t_i} \times 100 = \frac{\sum t_i c_i}{\sum t_i}$$

3.2.5 Environment

The experiment has been executed on a Mac OS X 10.6.7 64-bit operating system with one Intel Core i5 CPU @ 2.53 GHz processor with two cores and RAM size of 4.00GB, L2 cache size of 256k KB per core, Java 6 SE Update 25, Eclipse Helio IDE 3.6.2, JUnit 4.8.2, EclEmma 1.5.3 and Randoop 1.3.2.

3.2.6 Tool Selection

After the review of potential tools in chapter 2, we narrowed the selection down to Randoop and JUnit. FindBugs were ruled out because it would involve a comparison between random testing, partition testing and Statistic Analysis, which would be outside the scope of the thesis. We decided to not use JWalk because it does not produce any JUnit cases, which makes it impossible to measure the coverage rate of the tests. JCrasher only provokes exceptions and is therefor not suited for the experiment. We also decided to leave out fault injection with MuJava because the tracker of Buddi was active with many open bugs (25).

3.2.7 Hypothesis

Based on the problem statement given in 1.2 we have formulated the following hypothesis: The null hypothesis is that there is no difference in the number of failures found by random testing and partition testing and that the two achieves the same amount of coverage. The alternate hypothesis is that partition testing achieve a higher coverage rate and reveals a higher number of failures than random testing. Stated:

$$H_0 : \mu_{random} = \mu_{partition}$$
$$H_1 : \mu_{random} < \mu_{partition}$$

A paired t test will be performed since two paired samples will be the results of the experiment. The criterion is $H_1 : \mu_{random} < \mu_{partition}$: rejects H_0 if $|t_0 > t_{\alpha/2,n-1}$. Alpha(α) has been set to = 0.1. This level has been chosen to reflect that the experiment performed on a open-source program.

3.3 Manual Category-Partition testing

The first testing activity we did was manual category-partition testing. This would ensure that we would early on gain a good overview of the code and how it worked. Another reason for choosing to perform the experiment in this order was to avoid potential bias.

The steps explained in chapter 2 was followed. In the analysis phase, info on the Buddi web page [30] and the UML diagrams provided in appendix A was studied. We discovered that the documentation provided was not adequate. For example, in the UML diagram over the system, the class FilteredLists was not included. To

compensate the lack of documentation we borrowed the concept of path domain from Equivalence testing [34]. A path is considered to be a "...sequence of statements through the implementation" [34] and the path domain is the set of paths. In the categorize phase it is allowed to take the code into consideration[31]. Since we did not have a test generator available, I generated the test cases by hand. Appendix C contains the formal test specifications of Buddi produced with category-partition.

3.4 Randoop Testing

Because of Randoop's random basis we experimented with the test generation prior to conducting the experiment. Randoop takes as input a given amount of time to generate test cases or a number of test cases to generate. Narrowing the input range on values that would provide maximum coverage would optimize the execution of the experiment. Figure 3.4 show the input screen with the available parameters.

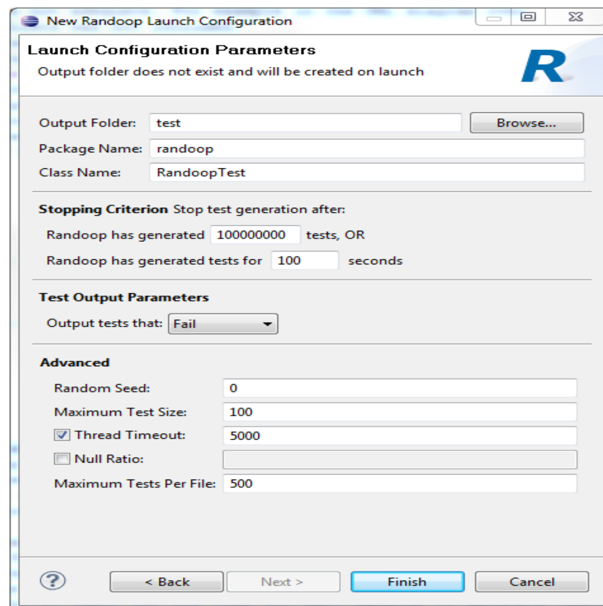


Figure 3.4: Overview of Randoop input

I experimented with different seeds chosen randomly from a list of primes, which is provided in appenidx B, to make sure that all the test sequences generated was not the same. We also experimented with the amount of test cases needed to reach the maximum possible width coverage of the class under test (CUT), but found it more useful to input the amount of seconds you want Randoop to generate tests. I also experimented with the @CheckRep annotation (contract interface mentioned in 2) to

write contracts. We did not find this option valuable as most of the classes in Buddi already had the proper logic to check that they their state was legal.

3.4.1 Code Changes

The following changes have been made to make the code to fit Randoop. Without these changes Randoop was not able to test the clone method or instantiate the objects properly.

- The tag **public** has been added to the method clone() to the following classes: AccountImpl, AccountTypeImpl, BudgetCategoryImpl, DocumentImpl, ModelObjectImpl, ScheduledTransactionImpl, SplitImpl, TransactionImpl and TransactionSplitImpl.
- A constructor has been made to replace ModelFactory instantiation of the following classes: AccountImpl, AccountTypeImpl, BudgetCategoryImpl, DocumentImpl, ScheduledTransactionImpl, SplitImpl, TransactionImpl and TransactionSplitImpl.

3.4.2 Randoop Regression Testing

The steps below explains how regression testing was performed in this experiment with Randoop Testing

- The package of the classes under test was targeted with test type pass. (ModelImpl Package)
- A seed was chosen randomly from the list in appendix B.
- Execution time was set to 180 seconds.
- EclEmma was executed to measure the coverage provided by the generated test cases.
- This was repeated three times, and the results was averaged class wise.

3.4.3 Randoop failure testing

The steps below explains how fail testing was performed in this experiment with Randoop Testing:

- The package of the classes under test was targeted with test type fail. (ModelImpl Package)
- A seed was chosen randomly from the list in appendix B.
- Execution time was set to 180 seconds.

- Output report was examined and reported errors were checked to validate.
- This was repeated three times, and the results was averaged class wise.

4 Test results

4.1 Overview of Test Results

Table 4.1 contains the results found with random testing as test strategy. It contains both the coverage rate and fault rate. Table 4.2 contains the results found with partition testing as testing strategy.

4.2 Coverage Results

The component coverage rate for partition testing is better than the component coverage rate of random testing. With random testing, three classes in table 4.1 stand out with less than half the coverage of the other classes. A common property shared by these three classes are methods that take as input a file in a special format the program uses to save a current session for later use. The issue with random testing and complex input types were discussed in section 2.2.5 and in DocumentImpl for instance, a total of 98 LOC is missed because they are related to a file format which Randoop cannot instantiate.

4.2.1 Complexity

In most of the other cases 100% coverage is not reached because of complex combinations of conditions. Figure 4.1 is an example of this. It shows the logic performed to determine if an instance of the Transaction Class is valid. The code snippet below is not performed by TransactionImpl. Instead it resides within DocumentImpl. Nine different paths must be executed by the test suite to make sure that all illegal states will be discovered.

Many of the paths interact with the classes Splits and BudgetCategory, which

Name	Coverage	Number of Bugs
AccountImpl	65.5%	0
AccountTypeImpl	94.4%	1
BudgetCategoryImpl	80.4%	2
BudgetCategoryTypeMonthly	100%	0
BudgetCategoryTypeQuarterly	100%	0
BudgetCategoryTypeSemiMonthly	100%	0
BudgetCategoryTypeSemiYearly	100%	0
BudgetCategoryTypeWeekly	100%	0
BudgetCategoryTypeYearly	100%	0
ConcurrentSaveException	100%	0
Day & Time	100%	0
DocumentImpl	43.5%	2
FilteredLists	31.5%	0
ModelFactory	39.3%	0
ModelObjectImpl	100%	1
ScheduledTransactionImpl	91.1%	0
SourceImpl	100%	0
SplitImpl	88.9%	0
Time	100%	0
TransactionImpl	67.7%	0
TransactionSplitImpl	88.4%	0
WrapperLists	88.9%	0
Component Coverage rate	57.8%	6

Table 4.1: Random testing statistics

add additional environmental conditions. Category-partition testing has properties to handle situations like these, but it is up to the tester to identify them. With random testing, the internal workings of Randoop dictates the probability of generating an environment with properties that allows the paths to be executed. To raise the probability multiple test runs should be performed, but no guarantees of execution can be provided.

Also in the class DocumentImpl, 124 LOC is missed due to logic regarding updating of Scheduled Transactions. A scheduled transaction works as this:

- A start date and end date is given to indicate the period of time this transaction will be scheduled.
- Three integers specify day, month and/or week when the scheduled transaction will be committed.
- Which integer DocumentImpl takes into account is dependent on which one of

Name	Coverage	Number of bugs
AccountImpl	90.3%	1
AccountTypeImpl	90.9%	0
BudgetCategoryImpl	91.6%	4
BudgetCategoryTypeMonthly	100%	0
BudgetCategoryTypeQuarterly	100%	0
BudgetCategoryTypeSemiMonthly	93.9%	0
BudgetCategoryTypeSemiYearly	90.9%	0
BudgetCategoryTypeWeekly	100%	0
BudgetCategoryTypeYearly	100%	0
ConcurrentSaveException	100%	0
Day & Time	100%	0
DocumentImpl	75.4%	3
FilteredLists	86.2%	3
ModelFactory	61.2%	0
ModelObjectImpl	87.5%	0
ScheduledTransactionImpl	93.4%	0
SourceImpl	100%	0
SplitImpl	88.9%	0
Time	100%	0
TransactionImpl	89.2%	0
TransactionSplitImpl	88%	0
WrapperLists	88.9%	0
Component Coverage rate	81.7%	10

Table 4.2: Category Partition test statistics

nine possible filters that the scheduled transaction specified.

- If a date provided as parameter to the updateScheduledTransactions(Date date) method matches a scheduled transaction, then it is updated.

In random testing this means that four different randomly generated parameters must match. The probability of this condition taking place with the default integer generator of Randoop, listed in 2.6, is 0.0. Assuming an integer generator that generates numbers in the range [1, 2011], so this year may be represented, and instantiation is done with the method DateUtil.getDate(int year, int month, int day), the probability of generating two tuples of equal numbers where the day and month numbers are valid equals to:

$$\left(\frac{1}{2011} \times \frac{30}{2011} \times 1\right) \times \left(\frac{1}{2011} \times \frac{1}{2011} \times \frac{1}{2011}\right) \approx 0.0000000441$$

Class	Method	Description
Accounts	getStartDate()	Nullpointer Exception
BudgetCategories	getChildren & getAllChildren	Nullpointer Exception
DocumentImpl	removeBudgetCategory()	Nullpointer Exception

Table 4.3: Intersection of failures

To complicate Scheduled Transaction further, the nine filters mentioned above are defined as enums, but the class ScheduledTransactions takes the Enum input as a string, and as you can see in figure 4.2, the comparison is done with the enum as a string. Given a random string generator that generates random strings with length in the range [0,40] and that the characters in the string is generated from the last 96 symbols of the ASCII table, the string representation of a filter:
"SCHEDULE_FREQUENCY_MONTHLY_BY_DATE"
has the following probability of being generated at random:

$$\frac{1}{40} \times (\frac{1}{96})^{34} \approx 4.006 * 10^{-68}$$

So the probability of generating a match with date and filter is:

$$0.0000000441 \times 4.006 * 10^{-68} \approx 1.76 \times 10^{-75}$$

Figure 4.2 and 4.3 shows the implementation.

Another shared property of DocumentImpl, FilteredLists and ModelFactory is that some methods have been omitted by the random generator. Figure 4.4, 4.5 and 4.6 shows example outputs from the test runs. Since the generator used for this experiment is feedback based, a possible explanation could be that a false positive were given as feedback and the generator skipped the methods. This is also backed up by the fact that the same classes were missed when the test run was repeated with another seed. Another possible explanation is that the generator ran out of time slated for that class and skipped the method. Since Randoop does not output any information about the test run these qualified guesses are all I can provide. In table 5.7 it can also be seen that some classes tested with Randoop stands out with notable lower coverage rate than the other classes.

ModelFactory is also subject to the special file format and an auto save property it has. The method createDocument(File file) misses 118 LOC because of this. The whole class is 196 LOC. Most of these lines are parsing of the file, which will contain a possible state of the program.

4.3 Failures Found

With category-partition testing 10 failures were revealed, while random testing revealed 6 failures (tables 4.2 and 4.1. The failures revealed with category-partition

testing are concentrated in the classes FilteredLists, DocumentImpl and Budget Category. The faults found in FilteredLists is related to objects that should be or not in the list. These were not found by Randoop, because it will not check for it. Manual testing has the advantage of discovering non-computational errors such as faulty design practices, inconsistencies and interface errors. Such a failure is for instance in Accounts. If you set the date to be null you will be allowed to do so. If you then call the getDate() function it will not return null, but today's date. If you are not allowed to set date to null then it would be the setters task to determine a legal instance and not the getter. Figure 4.7 shows a failure in DocumentImpl that both strategies found. Source s is set to null if there exists no budget categories or accounts, which makes sense, because these are objects that are sources. However, when concatenating the string *message*, s.getFullName() will of course give a nullpointer exception.

4.3.1 Java API Failure Found

Figure 4.8 shows a snippet from a Randoop test case showing two odd failures found in the Java API. It violates the Java contract *Symmetry of Equality* & the java contract *Equals-hashcode*. The pre-condition for this to happen is that both lists are empty. If this could affect the program is highly unlikely since Hashcodes are not used and lists are not compared to each other. According to the failure definition given in 3 this should not be considered a failure in Buddi, but rather in the Java API, which is not under test in this experiment. It seems that the contracts of Randoop does not fit every program.

```
if (object instanceof Transaction){
    Transaction t = (Transaction) object;
    if (t.getFrom() instanceof Split && t.getFromSplits() != ↩
        null){
        long splitSum = 0;
        for (TransactionSplit split : t.getFromSplits()) {
            splitSum += split.getAmount();
            if (split.getSource() == null)
                throw new ModelException("...");
            if (split.getAmount() == 0)
                throw new ModelException("...");
            if (split.getSource() instanceof BudgetCategory
                    && !((BudgetCategory) split.getSource()).↩
                        isIncome())
                throw new ModelException("...");
        }
        if (splitSum != t.getAmount())
            throw new ModelException("...");
        else
            if (t.getFrom() instanceof BudgetCategory
                    && !((BudgetCategory) t.getFrom()).isIncome↩
                        ())
                throw new ModelException("...");
        if (t.getTo() instanceof Split && t.getToSplits() != ↩
            null){
            long splitSum = 0;
            for (TransactionSplit split : t.getToSplits()) {
                splitSum += split.getAmount();
                if (split.getSource() == null)
                    throw new ModelException("...");
                if (split.getAmount() == 0)
                    throw new ModelException("...");
                if (split.getSource() instanceof BudgetCategory
                && ((BudgetCategory) split.getSource()).isIncome↩
                    ())
                    throw new ModelException("...");
            }
            if (splitSum != t.getAmount())
                throw new ModelException("...");
        } else {
            if (t.getTo() instanceof BudgetCategory
                    && ((BudgetCategory) t.getTo()).isIncome())
                throw new ModelException("...");
```

Figure 4.1: Transaction validation check

```
if(s.getFrequencyType().equals(ScheduleFrequency.↩
    SCHEDULE_FREQUENCY_MONTHLY_BY_DATE.toString())
         && (s.getScheduleDay() == tempCal.get(Calendar.↩
              DAY_OF_MONTH) ||
         (s.getScheduleDay() == 32 && tempCal.get(Calendar.↩
              DAY_OF_MONTH)
         == tempCal.getActualMaximum(Calendar.DAY_OF_MONTH)))) {
    todayIsTheDay = true;
} else if
(s.getFrequencyType().equals(ScheduleFrequency.↩
    SCHEDULE_FREQUENCY_MONTHLY_BY_DAY_OF_WEEK.toString()) && s.↩
    getScheduleDay() + 1 == tempCal.get(Calendar.DAY_OF_WEEK) &&
         tempCal.get(Calendar.DAY_OF_MONTH) <= 7) {
    todayIsTheDay = true; }else
if(s.getFrequencyType().equals(ScheduleFrequency.↩
    SCHEDULE_FREQUENCY_WEEKLY.toString())
         && s.getScheduleDay() + 1 == tempCal.get(Calendar.↩
              DAY_OF_WEEK)) {
    todayIsTheDay = true; }
else if(s.getFrequencyType().equals(ScheduleFrequency.↩
    SCHEDULE_FREQUENCY_BIWEEKLY.toString()) && s.getScheduleDay()↩
     + 1 == tempCal.get(Calendar.DAY_OF_WEEK) && ((DateUtil.↩
    getDaysBetween(lastDayCreated, tempDate, false) >= 13= || ↩
    isNewTransaction)){ todayIsTheDay = true; lastDayCreated = (↩
    Date) tempDate.clone(); if(isNewTransaction) isNewTransaction↩
    =false; } else if
(s.getFrequencyType().equals(ScheduleFrequency.↩
    SCHEDULE_FREQUENCY_EVERY_X_DAYS.toString()) && DateUtil.↩
    getDaysBetween(lastDayCreated, tempDate, false) >=
         s.getScheduleDay() ){
    todayIsTheDay = true;
    lastDayCreated = (Date) tempDate.clone();
}
else if
(s.getFrequencyType().equals(ScheduleFrequency.↩
    SCHEDULE_FREQUENCY_EVERY_DAY.toString())){
    todayIsTheDay = true;
}
```

Figure 4.2: Scheduled Transaction Update Method part 1.

```
else if (s.getFrequencyType().equals(ScheduleFrequency.↩
    SCHEDULE_FREQUENCY_EVERY_WEEKDAY.toString())
        && (tempCal.get(Calendar.DAY_OF_WEEK) < Calendar.↩
            SATURDAY)
        && (tempCal.get(Calendar.DAY_OF_WEEK) > Calendar.SUNDAY)↩
            ){
    todayIsTheDay = true;
}else if(s.getFrequencyType().equals(ScheduleFrequency.↩
    SCHEDULE_FREQUENCY_MULTIPLE_WEEKS_EVERY_MONTH.toString()) && ↩
    s.getScheduleDay() + 1 == tempCal.get(Calendar.DAY_OF_WEEK)) ↩
    {
    int week = s.getScheduleWeek();
    int weekNumber = tempCal.get(Calendar.DAY_OF_WEEK_IN_MONTH) ↩
        - 1;
    int weekMask = (int) Math.pow(2, weekNumber);
    if ((week & weekMask) != 0)
        todayIsTheDay = true;
}else if(s.getFrequencyType().equals(ScheduleFrequency.↩
    SCHEDULE_FREQUENCY_MULTIPLE_MONTHS_EVERY_YEAR.toString()) && ↩
    s.getScheduleDay() == tempCal.get(Calendar.DAY_OF_MONTH)) {
    int months = s.getScheduleMonth();
    int monthMask = (int) Math.pow(2, tempCal.get(Calendar.MONTH↩
        ));
    if ((months & monthMask) != 0)
        todayIsTheDay = true;
}
if (todayIsTheDay){
    for (Transaction t : getTransactions(tempDate, tempDate)) {
        if (DateUtil.isSameDay(t.getDate(), tempDate) && t.↩
            isScheduled()
                && t.getFrom().equals(s.getFrom()) && t.getTo()↩
                    .equals(s.getTo())
                && t.getDescription().equals(s.getDescription())↩
                    ) {
        todayIsTheDay = false;
```

Figure 4.3: Scheduled Transaction update method part 2.

Element	Coverage	Covered Lines	Missed Lines	Total Lines
refreshUidMap()	100.0 %	17	0	17
registerObjectInUidMap(ModelObj	100.0 %	2	0	2
removeAccount(Account)	55.6 %	5	4	9
removeAccountType(AccountType	66.7 %	4	2	6
removeBudgetCategory(BudgetCat	0.0 %	0	4	4
removeScheduledTransaction(Sche	0.0 %	0	3	3
removeTransaction(Transaction)	0.0 %	0	3	3
save()	100.0 %	2	0	2
saveAs(File)	0.0 %	0	2	2
saveAuto(File)	0.0 %	0	22	22
saveToStream(OutputStream)	0.0 %	0	36	36
saveWrapper(File, boolean)	43.2 %	19	25	44
setAccounts(List<Account>)	100.0 %	3	0	3
setAccountTypes(List<AccountTyp	100.0 %	3	0	3
setBudgetCategories(List<Budget(	100.0 %	3	0	3

Figure 4.4: Test output of DocumentImpl

Element	Coverage	Covered Lines	Missed Lines	Total Lines
FilteredLists.java	29.6 %	58	138	196
FilteredLists	29.6 %	58	138	196
AccountListFilteredByDeleted	80.0 %	4	1	5
AccountListFilteredByDeleted(D	100.0 %	2	0	2
AccountListFilteredByType	83.3 %	5	1	6
AccountListFilteredByType(Doc	100.0 %	3	0	3
BuddiFilteredList<T>	100.0 %	8	0	8
BuddiFilteredList(Document, Lis	100.0 %	4	0	4
updateFilteredList()	100.0 %	4	0	4
BudgetCategoryListFilteredByChild	66.7 %	6	3	9
BudgetCategoryListFilteredByCh	100.0 %	3	0	3
BudgetCategoryListFilteredByDelet	80.0 %	4	1	5
BudgetCategoryListFilteredByD	100.0 %	2	0	2
BudgetCategoryListFilteredByParer	77.8 %	7	2	9
BudgetCategoryListFilteredByPa	100.0 %	3	0	3
BudgetCategoryListFilteredByPerio	85.7 %	6	1	7
BudgetCategoryListFilteredByPe	100.0 %	3	0	3
ScheduledTransactionListFilteredB	21.4 %	3	11	14
ScheduledTransactionListFiltere	100.0 %	3	0	3
TransactionListFilteredByDate	50.0 %	4	4	8
TransactionListFilteredByDate(D	100.0 %	4	0	4
TransactionListFilteredBySearch	0.0 %	0	93	93
TransactionListFilteredBySearch	0.0 %	0	4	4
acceptCleared(Transaction)	0.0 %	0	17	17
acceptDate(Transaction)	0.0 %	0	29	29
acceptReconciled(Transaction)	0.0 %	0	17	17
acceptText(Transaction)	0.0 %	0	13	13
isFiltered()	0.0 %	0	1	1
setClearedFilter(TransactionCle	0.0 %	0	2	2
setDateFilter(TransactionDateFi	0.0 %	0	2	2
setReconciledFilter(Transaction	0.0 %	0	2	2
setSearchText(String)	0.0 %	0	2	2
TransactionListFilteredBySource	20.0 %	3	12	15
TransactionListFilteredBySource	100.0 %	3	0	3
TypeListFilteredByAccounts	80.0 %	8	2	10
TypeListFilteredByAccounts(Do	100.0 %	3	0	3
getTransactionsBySearch(Documer	0.0 %	0	4	4
FilteredLists()	0.0 %	0	1	1

Figure 4.5: Test output of FilteredLists

Element	Coverage	Covered Lines	Missed Lines	Total Lines
ModelFactory	31.6 %	62	134	196
createAccount(String, AccountTyp	100.0 %	4	0	4
createAccountType(String, boolear	100.0 %	4	0	4
createBudgetCategory(String, Bud	100.0 %	5	0	5
createDocument()	50.0 %	24	24	48
createDocument(File)	0.0 %	0	86	86
createScheduledTransaction(String	0.0 %	0	15	15
createSplit()	100.0 %	1	0	1
createTransaction(Date, String, lor	100.0 %	7	0	7
createTransactionSplit(Source, lon	100.0 %	4	0	4
getAutoSaveLocation(File)	20.0 %	2	8	10
getBudgetCategoryType(BudgetCa	0.0 %	0	1	1
getBudgetCategoryType(String)	100.0 %	9	0	9

Figure 4.6: Test output of ModelFactoryImpl

```
.
.
if (t.getTo() == null){
    Source s = null;
    if (getBudgetCategories().size() > 0)
        s = getBudgetCategories().get(0);
    else if (getAccounts().size() > 0)
        s = getAccounts().get(0);
    else
        s = null;
    String message = "Transaction with description '" +
    t.getDescription() + "' of amount '"+ t.getAmount()
    + "' on date '" + t.getDate()
    + "' does not have a To source defined.
    Setting this to '" + s.getFullName();
```

Figure 4.7: Example of failure both strategies found

```
.
.
DocumentImpl var1 = new DocumentImpl();
java.util.List var2 = var1.getTransactions(...);
java.util.List var3 = var1.getAccountTypes();
assertTrue(var3.equals(var2) ? var3.hashCode() == var2.hashCode↩
    () : true);
assertTrue(var3.equals(var2) ? var2.equals(var3) : true);
```

Figure 4.8: Code revealing Java.util.AbstractList failure

5 Data Analysis

5.1 Paired T Test Coverage Results

Based on the two samples shown in table 5.1 the results from the experiment have been compared with a paired t test. Table 5.2 contains a review data and table 5.3 contains intermediate values used in the calculations.

Group	Random	Partition
Mean	85.4182	91.6500
SD	21.8821	9.3643
SEM	4.6653	1.9965
N	22	22

Table 5.2: Summary of data

$$\begin{aligned} t_0 &= 1.8619 \\ df &= 21 \\ t_{0.05,21} &= 1.721 \end{aligned}$$

Table 5.3: Intermediate values used in calculations

5.1.1 P Value and Statistical Significance

The two-tailed P value equals 0.0767. By conventional criteria, this difference is not considered to be quite statistically significant. $t_0 > t_{0.05,21}$ means that the null hypothesis can be rejected at the 0.1 level.

Test Strategy	
Random Testing	Partition Testing
65.5	90.3
94.4	90.9
80.4	91.6
100	100
100	100
100	93.9
100	90.9
100	100
100	100
100	100
100	100
43.5	75.4
31.1	86.2
39.3	61.2
100	87.5
91.1	93.4
100	100
88.9	88.9
100	100
67.7	89.2
88.4	88
88.9	88.9

Table 5.1: Input paired t test coverage rate

5.2 Paired T Test Fault Results

Based on the two samples shown in table 5.4 the results from the experiment have been compared with a paired t test. Table 5.5 contains a review data and table 5.6 contains intermediate values used in the calculations.

Test Strategy	
Random Testing	Partition Testing
1	1
2	0
0	4
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
2	3
0	3
0	0
1	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0

Table 5.4: Input paired t test fault rate

Group	Random	Partition
Mean	0.2727	0.50
SD	0.6311	1.8529
SEM	0.1345	0.2527
N	22	22

Table 5.5: Summary of data

$$\begin{aligned} t_0 &= 1.22 \\ df &= 21 \\ t_{0.05,21} &= 1.721 \end{aligned}$$

Table 5.6: Intermediate values used in calculations

5.2.1 P Value and Statistical Significance

The two-tailed P value equals 0.2336. By conventional criteria, this difference is considered to be statistically significant. $t_0 < t_{0.05,21}$ means that the null hypothesis cannot be rejected at the 0.1 level. This makes it impossible to reject the null hypotheses.

5.3 Comparison with Another Study

Table 5.7 shows results from the study done by Bacchelli et. al. mentioned in the introduction [5]. Bacchelli et. al. does not mention specifically how the manual JUnit tests have been written, but states that the tests has been based on the white-box testing techniques of boundary-value analysis, data structure analysis, control path execution, error management path execution, mock objects and environments generation[5]. Many of these can be defined as partition testing. How the coverage has been calculated is also not mentioned. I have performed a paired t test on the coverage rate. Data showing failures per class have not been available but manual testing found a total 14 failures, while Randoop found 7 failures. Table 5.7 shows the input samples, table 5.8 reviews data of the samples and table 5.9 shows the intermediate values used in the calculation.

Coverage rate		
Name	Manual	Randoop
Base64	79.5	93.2
BitArray	71	87.7
HTMLDecoder	55.9	26.7
HTMLEncoder	71.1	85.5
HTMLNode	96.96	80.9
HexUtil	73.9	68.3
LRUHashtable	83	74.2
LRUQueue	83	88.9
Multivalue	84.4	73.9
SimpleField	53.8	64.6
SizeUtil	82.6	53.4
TimeUtil	94.8	55.5
URIPreCode	78.7	46.1
URLDecoder	67.5	46.6
URLEncoder	85.7	88.8
Average	77.46	68.95

Table 5.7: Bacchelli et. al results

Group	Partition	Random
Mean	77.4573	68.9533
SD	12.2742	19.7288
SEM	3.1692	5.0940
N	15	15

Table 5.8: Summary of data

$$\begin{aligned} t_0 &= 1.7475 \\ df &= 14 \\ t_{0.05,14} &= 1.761 \end{aligned}$$

Table 5.9: Intermediate values used in calculations

5.3.1 P Value and Statistical Significance

The two-tailed P value equals 0.1024. By conventional criteria, this difference is not considered to be statistically significant. $t_0 < t_{0.05,14}$ means that the null hypothesis cannot be rejected at the 10% level.

5.4 Threats to Validity

I will now give a brief discussion about the internal and external threats to the validity of this experiment

5.4.1 Internal Validity

Maturation

Prior to this experiment I had no knowledge of category-partition testing and this may have effected the first test cases produced. To reduce this threat I began with the smallest and least classes.

5.4.2 External Validity

Random Test Tool

The main strength of using Random testing is that it can provide a statistical predication of the significance to a successful random test. [15]. However, Randoop do not allow input of usage statistics. As Hamlet states: "...without a profile, reliability is not a meaningful idea" [15]. Instead reliability is offered through regression tests. An oracle was also not provided. This may have affected the the results and at least

they are valid for Randoop. Therefore a random test tool with the characteristics identified below should be built to elaborate the validity to random testing.

- Operational Profile provided.
- Oracle available.

Generalizability

This study was performed on one open-source program with one M.Sc student. The tests were also written at a late stadium of the development of the program. The results are therefore at least valid for testing performed by an M.Sc. on an open-source project. The tester risk have been reduced by comparing the results with another similar study. If the study is valid for an industry program should be investigated.

5.5 Assumptions of Theoretical Research

As mentioned earlier, Duran and Ntafos [10] assumed that failures do not intersect the defined partitions. How to define this in practice is difficult. For example, Buddi, contained an failure in the method getChildren() that belongs to the class BudgetCategoryImpl. If no child exists a nullpointer exception will occur that is not handled. This behavior will affect the method removeBudgetCatory(BudgetCategory) in the class DocumentImpl. When DocumentImpl attempts to remove a budget category it will check if it contains any children and a nullpointer exception will occur if the budget category contains no children and DocumentImpl will not be able to remove the budget category from its list. This failure clearly intersects between the partitions made in this experiment.

The assumption of uniform distribution of failures does not fit with the program tested in this experiment. This is showed in table 5.4. Instead failures seems clustered in classes with complex logic and a high number of LOCS. For example, four failures reside in the most complex class, DocumentImpl, while smaller classes such as Day and Time do not have any failures. The only assumption that seems to hold is that a program contains at least one failure.

6 Conclusion

Partition testing out-performed random testing 81% versus 57.8% with respect to the coverage rate and 10 versus 6 with respect to the fault rate. However the statistical difference was only able to reject the null hypothesis with respect to coverage rate at the 10% level. The data from the study performed by Bacchelli rejected the null hypothesis at the 10% level and partially supports the conclusion. The coverage rate was rejected due to a difference of 0.014 between t_0 and $t_{0.05,14}$. This was rather unexpected as the random test tool did not have all the properties suggested as necessary for Random Testing to be efficient [16], since an operational profile and an oracle is not provided. Another reason for the difference in total coverage of the component is that the classes that Random Testing missed, were the ones that contains the most LOC. This will not be detectable when calculating the arithmetic mean for the paired t test.

Even though Randoop only will check for a reduced set of faults, no statistical difference between partition testing and random testing was proved regarding the failures found, but partition testing found more bugs. As expected, Randoop discovered some odd failures, such as the Java API related failure.

Randoop allows you to produce vast amounts of test cases for many classes within a short period of times and is time efficient compared to manual writing of test cases. A drawback is that the tests should be reviewed for quality assurance as well as to pinpoint the cause and location of potential failures.

Handling of dates and matching of input parameters seems to be a weakness of random testing. Another weakness is when a data type is used to represent something else like the string for enum and int for day, month and week because, at least Randoop, will not let you specify three different integer generators.

A weakness of category-partition testing is that the quality of the tests is dependent on the accuracy of the specification belonging to the SUT. In this case this was not accurate and the code had to be taken into consideration. Improvisation of the test

specification was also necessary because the complexity, especially in DocumentImpl, made it difficult to specify all environmental biases in the test specification.

The strength of category-partition testing is the produced test specification that allows you to specify characteristics of both the input domain as well as the environment. By combining these a thorough test suite is produced. This is indicated by the average component coverage rate of 91.65 % that category-partition achieved.

Bibliography

[1] Manifesto for Agile Software Development. http://agilemanifesto.org, 2001. Last visited 9.11.10.

[2] About.com. Best personal finance software and tax software. http://financialsoft.about.com/od/reviewsfinancesoftware/ss/Best-Personal-Finance-Software-Best-Tax-Software_3.htm, 2011. Last visited 18.05.11.

[3] Eric Allen. *Title*. APress, 2002.

[4] Andrea Arcuri, Muhammad Zohaib Z. Iqbal, and Lionel C. Briand. Formal analysis of the effectiveness and predictability of random testing. In Paolo Tonella and Alessandro Orso, editors, *ISSTA*, pages 219–230. ACM, 2010.

[5] Alberto Bacchelli, Paolo Ciancarini, and Davide Rossi. On the effectiveness of manual and automatic unit test generation. In *Proceedings of the 2008 The Third International Conference on Software Engineering Advances*, pages 252–257, Washington, DC, USA, 2008. IEEE Computer Society.

[6] F.T. Chan, T.Y. Chen, I.K. Mak, and Y.T. Yu. Proportional sampling strategy: guidelines for software practitioners. *Information and Software Technology 38*, 1996.

[7] T. Y. Chen and Y. T. Yu. On the relationship between partition testing and random testing. *IEEE Transactions on Software Engineering, Vol. 20, No. 12*, 1999.

[8] Mountainminds GmbH & Co. Eclemma. http://www.eclemma.org/, 2006. Last visited 15.05.11.

[9] Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, 2004.

[10] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, SE-10(4):438 –444, 1984.

[11] Albert Endres and Dieter Rombach. *A Handbook of System and Software Engineering: Empirical Observations, Laws and Theories*. Addison-Wesley, 2003.

[12] D. Gelperin and B. Hetzel. The growth of software testing. *Commun. ACM*, 31:687–695, June 1988.

[13] Walter J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering, Vol. 25, No. 5*, 1999.

[14] D. Hamlet and R. Taylor. Partition testing does not inspire confidence [program testing]. *Software Engineering, IEEE Transactions on*, 16(12):1402 –1411, 1990.

[15] Dick Hamlet. When only random testing will do. In *When Only Random Testing Will Do*, 2006.

[16] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[17] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39:92–106, December 2004.

[18] William Howden. Functional program testing. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-6, NO. 2, MARCH 1980*, 1980.

[19] William E. Howden. Reliability of the path analysis testing strategy. *IEEE Trans. Software Eng.*, 2(3):208–215, 1976.

[20] Peter Lamb. Extending the jwalk testing tool by addition of ory partition testing. Master's thesis, University of Sheffield, 2007.

[21] J-C. Laprie. Dependable computing and fault tolerance : Concepts and terminology. In *Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., Twenty-Fifth International Symposium on*, page 2, jun 1995.

[22] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26:18–41, July 1993.

[23] J. L. Lions. Ariane 5 flight 501 failure. Technical report, Independent Inquire Board, 1996.

[24] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software reliability: measurement, prediction, application*. McGraw-Hill, Inc., New York, NY, USA, 1987.

[25] Glenford J. Myers. *The Art of Software Testing, Second Edititon.* John Wiley & Sons, Inc., 2004.

[26] NATO. Software engineering techniques. In *SOFTWARE ENGINEERING TECHNIQUES*, 1969.

[27] Peter G. Neumann. Accidental financial losses. *Commun. ACM*, 35:194–, September 1992.

[28] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, 14:868–874, 1988.

[29] Simeon C. Ntafos. On comparisons of random, partition, and proportional partition testing. *IEEE Transactions on Software Engineering, Vol. 27, No. 10*, 2001.

[30] Wyatt Olson. Buddi. `http://buddi.digitalcave.ca`, 2007. Last visited 15.05.11.

[31] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31:676–686, June 1988.

[32] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada.* ACM, October 2007.

[33] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[34] Debra J. Richardson and Lori A. Clarke. A partition analysis method to increase program reliability. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 244–253, Piscataway, NJ, USA, 1981. IEEE Press.

[35] Anthony Simons. Jwalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. *Automated Software Engineering*, 14:369–418, 2007.

[36] E.J. Weyuker and B. Jeng. Analyzing partition testing strategies. *Software Engineering, IEEE Transactions on*, 17(7):703 –711, July 1991.

[37] E.J. Weyuker and T.J. Ostrand. Theories of program testing and the application of revealing subdomains. *Software Engineering, IEEE Transactions on*, SE-6(3):236 – 246, 1980.

[38] Claes Wohlin, Per Runeson, Martin HÃűst, Magnus C. Ohlsson, BjÃűrn Regnell, and Anders WesslÃl'n. *Experimentation in Software Engineering.* Kluwer Academic Publishers, 2003.

[39] Yu-Seung, Jeff Offutt, and Yong Rae Kwon. Mujava : An automated class mutation system. *Software Testing, Verification and Reliability, 15(2):97-133*, 2005.

Buddi Specifications

This appendix contains information about the Buddi implementation[30].

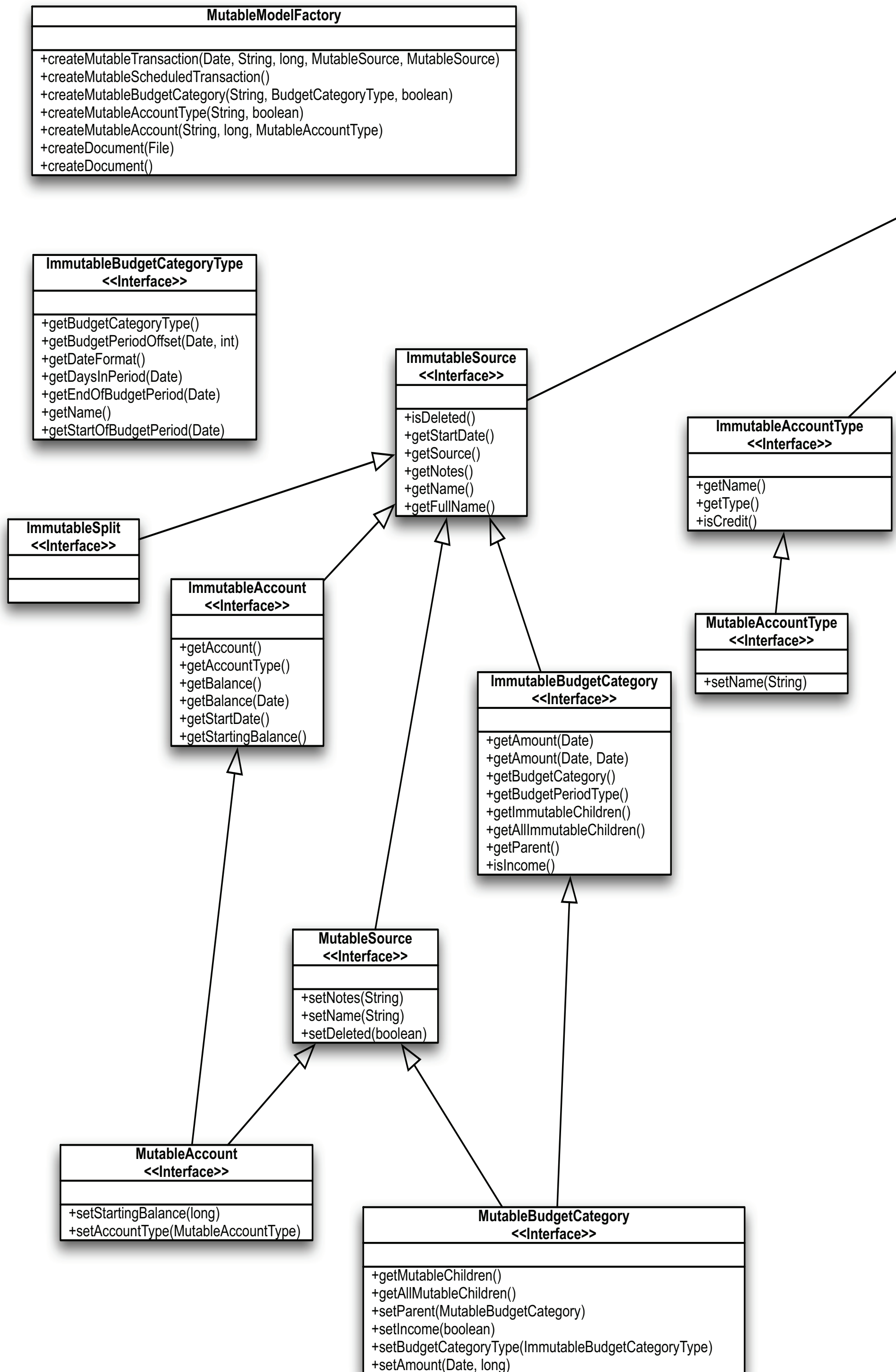
A.1 Specifications

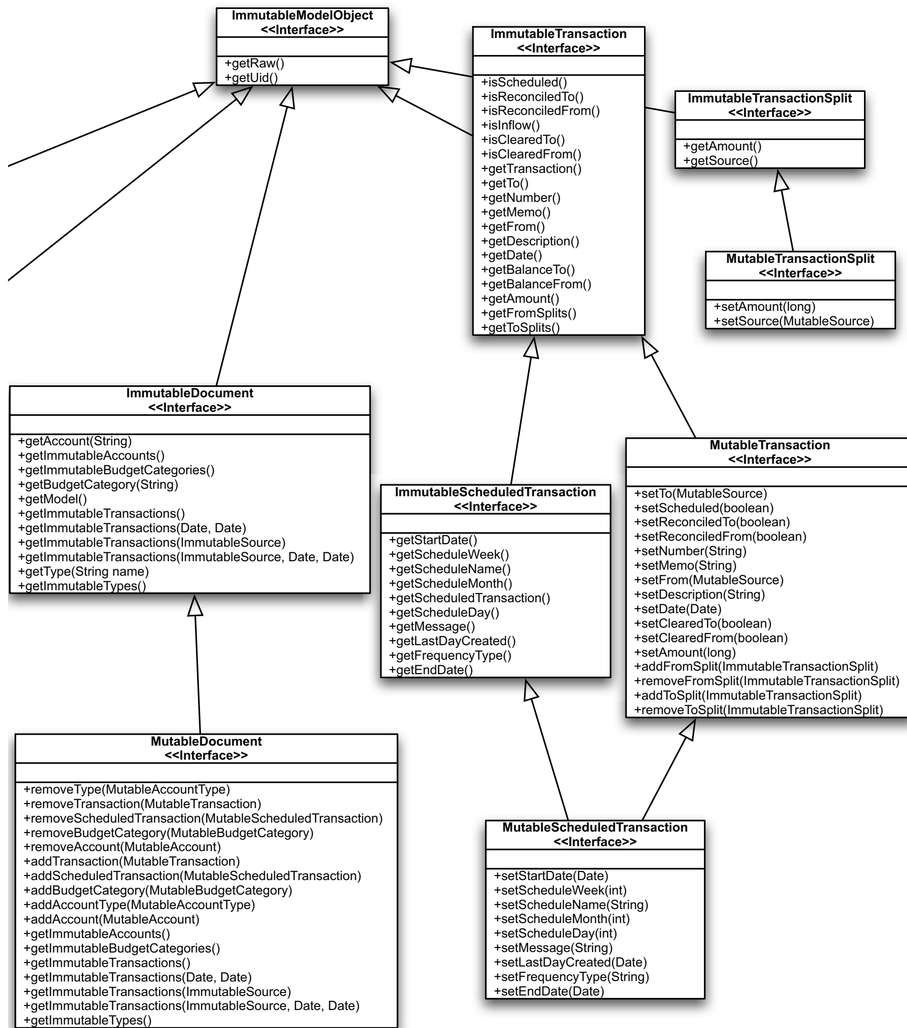
A.1.1 Accounts

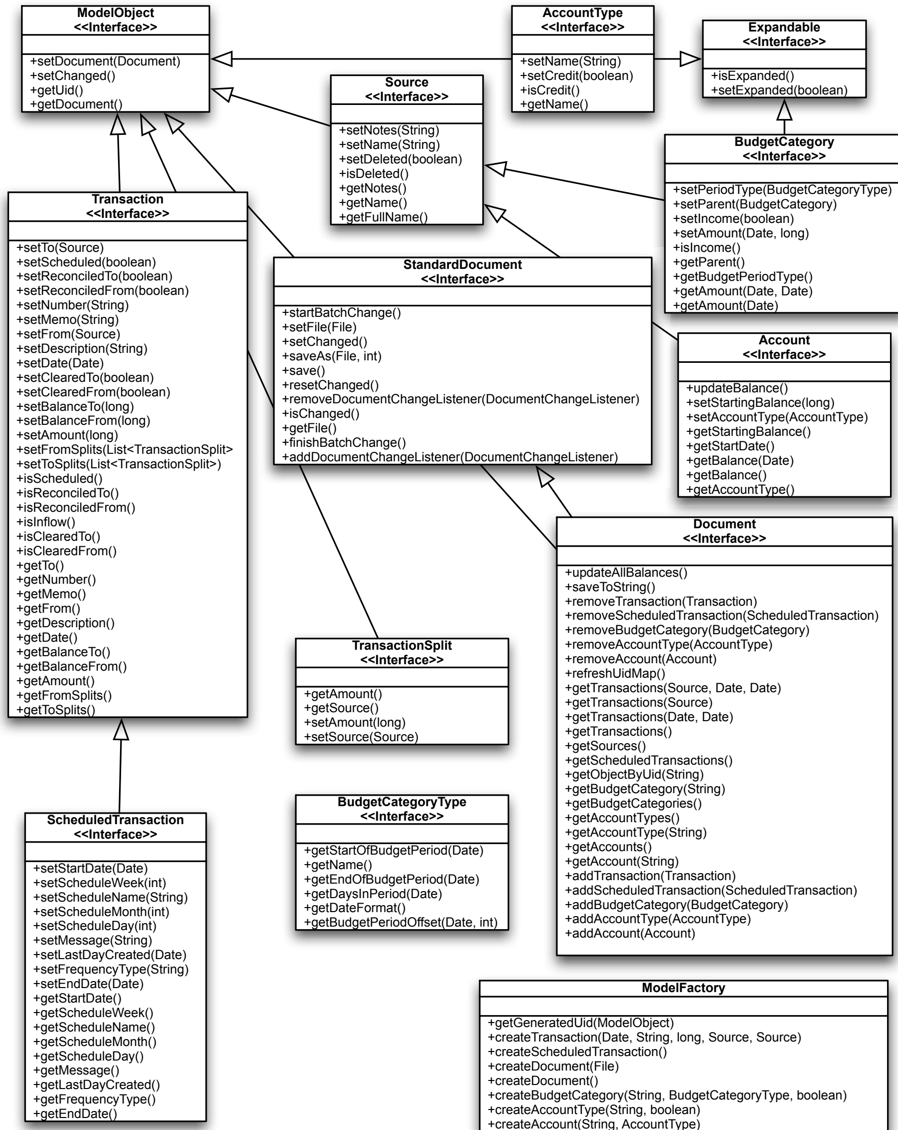
How the balance should be displayed according to account type and value

Debit		Credit
Value >= 0	Black	
Value < 0	Red, Negative	
Value > 0		Black, Negative
Value <= 0		Red

Table A.1: Accounts specification categories


MutableModelFactory
+createMutableTransaction(Date, String, long, MutableSource, MutableSource)
+createMutableScheduledTransaction()
+createMutableBudgetCategory(String, BudgetCategoryType, boolean)
+createMutableAccountType(String, boolean)
+createMutableAccount(String, long, MutableAccountType)
+createDocument(File)
+createDocument()
ImmutableBudgetCategoryType
<<Interface>>
+getBudgetCategoryType()
+getBudgetPeriodOffset(Date, int)
+getDateFormat()
+getDaysInPeriod(Date)
+getEndOfBudgetPeriod(Date)
+getName()
+getStartOfBudgetPeriod(Date)
ImmutableSource
<<Interface>>
+isDeleted()
+getStartDate()
+getSource()
+getNotes()
+getName()
+getFullName()
ImmutableAccountType
<<Interface>>
+getName()
+getType()
+isCredit()
ImmutableSplit
<<Interface>>
ImmutableAccount
<<Interface>>
+getAccount()
+getAccountType()
+getBalance()
+getBalance(Date)
+getStartDate()
+getStartingBalance()
MutableAccountType
<<Interface>>
+setName(String)
ImmutableBudgetCategory
<<Interface>>
+getAmount(Date)
+getAmount(Date, Date)
+getBudgetCategory()
+getBudgetPeriodType()
+getImmutableChildren()
+getAllImmutableChildren()
+getParent()
+isIncome()
MutableSource
<<Interface>>
+setNotes(String)
+setName(String)
+setDeleted(boolean)
MutableAccount
<<Interface>>
+setStartingBalance(long)
+setAccountType(MutableAccountType)
MutableBudgetCategory
<<Interface>>
+getMutableChildren()
+getAllMutableChildren()
+setParent(MutableBudgetCategory)
+setIncome(boolean)
+setBudgetCategoryType(ImmutableBudgetCategoryType)
+setAmount(Date, long)


ImmutableModelObject
<<Interface>>
+getRaw()
+getUid()
ImmutableTransaction
<<Interface>>
+isScheduled()
+isReconciledTo()
+isReconciledFrom()
+isInflow()
+isClearedTo()
+isClearedFrom()
+getTransaction()
+getTo()
+getNumber()
+getMemo()
+getFrom()
+getDescription()
+getDate()
+getBalanceTo()
+getBalanceFrom()
+getAmount()
+getFromSplits()
+getToSplits()
ImmutableTransactionSplit
<<Interface>>
+getAmount()
+getSource()
MutableTransactionSplit
<<Interface>>
+setAmount(long)
+setSource(MutableSource)
ImmutableDocument
<<Interface>>
+getAccount(String)
+getImmutableAccounts()
+getImmutableBudgetCategories()
+getBudgetCategory(String)
+getModel()
+getImmutableTransactions()
+getImmutableTransactions(Date, Date)
+getImmutableTransactions(ImmutableSource)
+getImmutableTransactions(ImmutableSource, Date, Date)
+getType(String name)
+getImmutableTypes()
ImmutableScheduledTransaction
<<Interface>>
+getStartDate()
+getScheduleWeek()
+getScheduleName()
+getScheduleMonth()
+getScheduledTransaction()
+getScheduleDay()
+getMessage()
+getLastDayCreated()
+getFrequencyType()
+getEndDate()
MutableTransaction
<<Interface>>
+setTo(MutableSource)
+setScheduled(boolean)
+setReconciledTo(boolean)
+setReconciledFrom(boolean)
+setNumber(String)
+setMemo(String)
+setFrom(MutableSource)
+setDescription(String)
+setDate(Date)
+setClearedTo(boolean)
+setClearedFrom(boolean)
+setAmount(long)
+addFromSplit(ImmutableTransactionSplit)
+removeFromSplit(ImmutableTransactionSplit)
+addToSplit(ImmutableTransactionSplit)
+removeToSplit(ImmutableTransactionSplit)
MutableDocument
<<Interface>>
+removeType(MutableAccountType)
+removeTransaction(MutableTransaction)
+removeScheduledTransaction(MutableScheduledTransaction)
+removeBudgetCategory(MutableBudgetCategory)
+removeAccount(MutableAccount)
+addTransaction(MutableTransaction)
+addScheduledTransaction(MutableScheduledTransaction)
+addBudgetCategory(MutableBudgetCategory)
+addAccountType(MutableAccountType)
+addAccount(MutableAccount)
+getImmutableAccounts()
+getImmutableBudgetCategories()
+getImmutableTransactions()
+getImmutableTransactions(Date, Date)
+getImmutableTransactions(ImmutableSource)
+getImmutableTransactions(ImmutableSource, Date, Date)
+getImmutableTypes()
MutableScheduledTransaction
<<Interface>>
+setStartDate(Date)
+setScheduleWeek(int)
+setScheduleName(String)
+setScheduleMonth(int)
+setScheduleDay(int)
+setMessage(String)
+setLastDayCreated(Date)
+setFrequencyType(String)
+setEndDate(Date)


ModelObject
<<Interface>>
+setDocument(Document)
+setChanged()
+getUid()
+getDocument()
AccountType
<<Interface>>
+setName(String)
+setCredit(boolean)
+isCredit()
+getName()
Expandable
<<Interface>>
+isExpanded()
+setExpanded(boolean)
Source
<<Interface>>
+setNotes(String)
+setName(String)
+setDeleted(boolean)
+isDeleted()
+getNotes()
+getName()
+getFullName()
BudgetCategory
<<Interface>>
+setPeriodType(BudgetCategoryType)
+setParent(BudgetCategory)
+setIncome(boolean)
+setAmount(Date, long)
+isIncome()
+getParent()
+getBudgetPeriodType()
+getAmount(Date, Date)
+getAmount(Date)
Transaction
<<Interface>>
+setTo(Source)
+setScheduled(boolean)
+setReconciledTo(boolean)
+setReconciledFrom(boolean)
+setNumber(String)
+setMemo(String)
+setFrom(Source)
+setDescription(String)
+setDate(Date)
+setClearedTo(boolean)
+setClearedFrom(boolean)
+setBalanceTo(long)
+setBalanceFrom(long)
+setAmount(long)
+setFromSplits(List<TransactionSplit>
+setToSplits(List<TransactionSplit>)
+isScheduled()
+isReconciledTo()
+isReconciledFrom()
+isInflow()
+isClearedTo()
+isClearedFrom()
+getTo()
+getNumber()
+getMemo()
+getFrom()
+getDescription()
+getDate()
+getBalanceTo()
+getBalanceFrom()
+getAmount()
+getFromSplits()
+getToSplits()
StandardDocument
<<Interface>>
+startBatchChange()
+setFile(File)
+setChanged()
+saveAs(File, int)
+save()
+resetChanged()
+removeDocumentChangeListener(DocumentChangeListener)
+isChanged()
+getFile()
+finishBatchChange()
+addDocumentChangeListener(DocumentChangeListener)
Account
<<Interface>>
+updateBalance()
+setStartingBalance(long)
+setAccountType(AccountType)
+getStartingBalance()
+getStartDate()
+getBalance(Date)
+getBalance()
+getAccountType()
Document
<<Interface>>
+updateAllBalances()
+saveToString()
+removeTransaction(Transaction)
+removeScheduledTransaction(ScheduledTransaction)
+removeBudgetCategory(BudgetCategory)
+removeAccountType(AccountType)
+removeAccount(Account)
+refreshUidMap()
+getTransactions(Source, Date, Date)
+getTransactions(Source)
+getTransactions(Date, Date)
+getTransactions()
+getSources()
+getScheduledTransactions()
+getObjectByUid(String)
+getBudgetCategory(String)
+getBudgetCategories()
+getAccountTypes()
+getAccountType(String)
+getAccounts()
+getAccount(String)
+addTransaction(Transaction)
+addScheduledTransaction(ScheduledTransaction)
+addBudgetCategory(BudgetCategory)
+addAccountType(AccountType)
+addAccount(Account)
TransactionSplit
<<Interface>>
+getAmount()
+getSource()
+setAmount(long)
+setSource(Source)
ScheduledTransaction
<<Interface>>
+setStartDate(Date)
+setScheduleWeek(int)
+setScheduleName(String)
+setScheduleMonth(int)
+setScheduleDay(int)
+setMessage(String)
+setLastDayCreated(Date)
+setFrequencyType(String)
+setEndDate(Date)
+getStartDate()
+getScheduleWeek()
+getScheduleName()
+getScheduleMonth()
+getScheduleDay()
+getMessage()
+getLastDayCreated()
+getFrequencyType()
+getEndDate()
BudgetCategoryType
<<Interface>>
+getStartOfBudgetPeriod(Date)
+getName()
+getEndOfBudgetPeriod(Date)
+getDaysInPeriod(Date)
+getDateFormat()
+getBudgetPeriodOffset(Date, int)
ModelFactory
+getGeneratedUid(ModelObject)
+createTransaction(Date, String, long, Source, Source)
+createScheduledTransaction()
+createDocument(File)
+createDocument()
+createBudgetCategory(String, BudgetCategoryType, boolean)
+createAccountType(String, boolean)
+createAccount(String, AccountType)

Income		Expense
Value >= 0	Black	Red, Negative
Value < 0	Red, Negative	Black

Table A.2: Categories specification

]

From	To	+/−	Color
A	A	+	Black
A	A	-	Red
A	C	+	Red
A	C	-	Black
C	A	+	Black
C	A	-	Red

Table A.3: Implementation wise: if ((To == Category AND Value >= 0) OR (To == Account AND Value < 0)) then Color = Red else Color = Black

Primes

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223	227	229
233	239	241	251	257	263	269	271	277	281
283	293	307	311	313	317	331	337	347	349
353	359	367	373	379	383	389	397	401	409
419	421	431	433	439	443	449	457	461	463
467	479	487	491	499	503	509	521	523	541
547	557	563	569	571	577	587	593	599	601
607	613	617	619	631	641	643	647	653	659
661	673	677	683	691	701	709	719	727	733
739	743	751	757	761	769	773	787	797	809
811	821	823	827	829	839	853	857	859	863
877	881	883	887	907	911	919	929	937	941
947	953	967	971	977	983	991	997	1009	1013
1019	1021	1031	1033	1039	1049	1051	1061	1063	1069
1087	1091	1093	1097	1103	1109	1117	1123	1129	1151
1153	1163	1171	1181	1187	1193	1201	1213	1217	1223
1229	1231	1237	1249	1259	1277	1279	1283	1289	1291
1297	1301	1303	1307	1319	1321	1327	1361	1367	1373
1381	1399	1409	1423	1427	1429	1433	1439	1447	1451
1453	1459	1471	1481	1483	1487	1489	1493	1499	1511
1523	1531	1543	1549	1553	1559	1567	1571	1579	1583

1597	1601	1607	1609	1613	1619	1621	1627	1637	1657
1663	1667	1669	1693	1697	1699	1709	1721	1723	1733
1741	1747	1753	1759	1777	1783	1787	1789	1801	1811
1823	1831	1847	1861	1867	1871	1873	1877	1879	1889
1901	1907	1913	1931	1933	1949	1951	1973	1979	1987
1993	1997	1999	2003	2011	2017	2027	2029	2039	2053
2063	2069	2081	2083	2087	2089	2099	2111	2113	2129
2131	2137	2141	2143	2153	2161	2179	2203	2207	2213
2221	2237	2239	2243	2251	2267	2269	2273	2281	2287
2293	2297	2309	2311	2333	2339	2341	2347	2351	2357
2371	2377	2381	2383	2389	2393	2399	2411	2417	2423
2437	2441	2447	2459	2467	2473	2477	2503	2521	2531
2539	2543	2549	2551	2557	2579	2591	2593	2609	2617
2621	2633	2647	2657	2659	2663	2671	2677	2683	2687
2689	2693	2699	2707	2711	2713	2719	2729	2731	2741
2749	2753	2767	2777	2789	2791	2797	2801	2803	2819
2833	2837	2843	2851	2857	2861	2879	2887	2897	2903
2909	2917	2927	2939	2953	2957	2963	2969	2971	2999
3001	3011	3019	3023	3037	3041	3049	3061	3067	3079

Test Specifications Buddi ModelImpl Package

C.1 Account

C.1.1 Starting Balance

Parameters:

Starting balance:

Negative
Positive
Empty

C.1.2 Start Date

Parameters:

Date:

Valid	[property valid]
null	

Environments:

Related Transactions:

None	[single]
Exactly one	[if valid]
More than one	[if valid]

C.1.3 Update Balance

Environments:

Related Transactions:

None	[single]
Exactly one	[if valid]
More than one	[if valid]

Movement of Transactions:

To account	[if valid]
From account	[if valid]

C.1.4 Get Balance

Parameters:

Date:

Valid	[property valid]
null	

Environments:

Related Transactions:

None	[single]
Exactly one	[if valid]
More than one	[if valid]

C.1.5 Overdraft Credit Limit

Parameters:

Limit:

Negative	[error]
Positive	
Empty	

C.1.6 Interest rate

Parameters:

Rate:

Negative	[error]
Positive	
Over 100%	[error]

C.1.7 Account Type

Parameters:

AccountType:

Credit	
Cash	
Other	
Empty	[error]

C.1.8 Compare To

Parameters:

Account's account tyep:

Credit set	[property credit]
Credit not set	[property nocredit]
null	[error]

ModelObject:

UidSet	[property uid]
null	[error]

Environments:

This account's account type:

Credit set	[if credit]
Credit set	[if nocredit]
Different name	[if uid]
Same name	[if uid]

C.2 Account Type

C.2.1 Account Type

Parameters:

Name:

Valid String	
Empty	[error]

Environments:

Document:

Set	[property doc]
Not set	

Other Account Type Names:

None	[if doc]
one or more with same name	[if doc][error]

C.2.2 Credit

Parameters:

Credit:

True
False

C.2.3 Expanded

Parameters:

Expanded:

True
False

C.2.4 Compare To

Parameters:

Account type:

Credit set	[property credit]
Credit not set	[property nocredit]
null	[error]

ModelObject:

UidSet	[property uid]
null	[error]

Environments:

This account type:

Credit set	[if credit]
Credit set	[if nocredit]
Different name	[if uid]
Same name	[if uid]

C.3 BudgetCategories

C.3.1 Get Amount

Parameters:

Date:

Valid date	[property valid]
Start date before end date	[property sequenced]
End date before end date	[property nonsequenced][single]
Start date in same budget period as end date	[property period]
Start date and end date area overlap two periods	[property overlap]
Not Valid	[error]

Environments:

Number of amounts related to date:

none	[if valid and sequenced]
one or more	[if valid and sequenced]
one or more	[if valid and period]
one or more	[if valid and overlap]

C.3.2 Income

Parameters:

Income:

True
False

C.3.3 Expanded

Parameters:

Expanded:

True
False

C.3.4 Budget Category Type

Parameters:

Type:

Monthly
Quarterly
SemiMonthly
SemiYearly
Weekly
Yearly

C.3.5 Family

Environments:

Number of children:

none
one or more

C.3.6 Get Budget Period Type

Parameters:

Start Date and End Date:

Valid date	[property valid]
Start date before end date	[property sequenced]
End date before end date	[property nonsequenced][single]

Environments:

Number of amounts related to date:

none	[if valid and sequenced]
one or more	[if valid and sequenced]

C.3.7 Delete

Environments:

Number of children:

none
one or more

C.4 Day & Time

C.4.1 Constructor

Parameters:

Date:

As integers
Date format
Null

Time:

As long
Date format
Zero

C.5 DocumentImpl

C.5.1 Remove Transaction:

Parameters:

Transaction:

Valid	[property valid]
null	[error]

Environments:

Number of transactions in transaction list:

none	[if valid][singel]
one	[if valid]

C.5.2 Remove ScheduledTransaction:

Parameters:

ScheduledTransaction:

Valid	[property valid]
null	[error]

Environments:

Number of scheduled transactions in scheduled transaction list:

none	[if valid][singel]
one	[if valid]

C.5.3 Remove BudgetCategory:

Parameters:

BudgetCategory:

Valid	[property valid]
null	[error]

Environments:

Number of budget categories in budget category list:

none	[if valid][singel]
one	[if valid]

Referring transactions:

none	[if valid][singel]
one or more	[error]

Containing scheduled transactions:

none [if valid][singel]
one ore more [error]

C.5.4 Remove Account:

Parameters:
Account:
Valid [property valid]
null [error]

Environments:
Number of accounts in account list:
none [if valid][singel]
one [if valid]
more than one [if valid]

Contains Transactions:
none [if valid][singel]
one or more [error]

Contains Scheduled Transactions:
none [if valid][singel]
one or more [error]

C.5.5 Remove AccountType:

Parameters:
AccountType:
Valid [property valid]
null [error]

Environments:
Number of account types in account type list:
none [if valid][singel]
one [if valid]

Environments:
Referring accounts:
none [if valid][singel]
one or more [error]

C.5.6 Get Transactions:

Parameters:

Source:

Valid	[property valid]
Account	[property account]
BudgetCategory	[property budgetcategory]
null	[error]
none	[property nosource]

Start Date:

null	[error]
none	[property nostart]
Before End date	[property start]
After End date	[error]

End Date:

null	[error]
none	[property noend]
After Start date	[property end]
Before Start date	[error]

Environments:

Number of transactions in transaction list:

none	[if valid and start and end][singel]
one	[if account and valid]
one	[if budget and valid]
one	[if account and valid and start and end and]
one	[if budget and valid and start and end]
more than one	[if account and valid]
more than one	[if budget and valid]
more than one	[if account and valid and start and end]
more than one	[if budget and valid and start and and end]

C.5.7 Get Sources:

Environments:

Number of sources in sources list:

Valid	
Account	
BudgetCategory	
null	[error]
none	

C.5.8 Get Accounts:

Environments:

Accounts:

Initialized
un-initialized

C.5.9 Get Account Types:

Environments:

Accounts:

Initialized
un-initialized

C.5.10 Get Budget Categories:

not applicable

C.5.11 Get Scheduled Transactions:

Environments:

Scheduled Transactions:

Initialized
un-initialized

C.5.12 Get Transactions:

Environments:

Transactions:

Initialized
un-initialized

C.5.13 Get Account:

Parameters:

Name:

Empty	[property empty]
one or more characters	[property nonempty]

Environments:

Number of occuring accounts:

none	[error]
one	[if nonempty]

C.5.14 Get Account Type:

Parameters:

Name:

Empty	[property empty]
one	[property nonempty]

Environments:

Number of occuring account types:

none	[error]
one or more	[if nonempty]

C.5.15 Get Budget Category:

Parameters:

Name:

Empty	[property empty]
one or more characters	[property nonempty]

Environments:

Number of occuring budget categories:

none	[error]
one or more	[if nonempty]

C.5.16 Update All Balances

Environments:

Number of accounts:

one or more

C.5.17 Add Transaction

Parameters:

Transaction:

From Split	[property from]
To split	[property to]
Not Valid	[error]

Environments:

Properties of the Transaction:

split source is null	[if from][error]
split amount equals zero	[if from][error]
split source is not an income budget category	[if from][error]
from split sum differs transaction sum	[if from][error]
split source is null	[if to][error]
split amount equals zero	[if to][error]
split source is not an income budget category	[if to][error]
to split sum differs transaction sum	[if to][error]

C.5.18 Add Scheduled Transaction

Parameters:

Scheduled Transaction:

Valid [property valid]

Environments:

Scheduled Transactions Already in list:

same name [if valid][error]

C.5.19 Add Budget Category

Parameters:

Budget Category:

Valid [property valid]

C.5.20 Add Account Type

Parameters:

Account Type:

Valid [property valid]

Environments:

Account Types Already in list:

same name [if valid][error]

C.5.21 Add Account

Parameters:

Account:

Valid [property valid]

Environments:
Accounts Already in list:
same name [if valid][error]

C.5.22 Save

Parameters:
File:
none [property nofile]
one [property file]

Environments:
File match:
one [if valid]

C.6 FilteredLists

C.6.1 Account List Filtered By Delete:

Parameters:
Account List:
One [property nonempty]
Many [property nonempty]
Empty [property empty]

Environments:
Deleted accounts in list:
none [if nonempty]
one or more [if nonempty]

C.6.2 Account List Filtered By Type:

Parameters:
Account Type List:
One [property nonempty]
Many [property nonempty]
Empty [property empty]

Environments:
Type of accounts in list:
none [if nonempty]
one or more [if nonempty]

C.6.3 Budget Category List Filtered By Children:

Parameters:

Budget Category List:

One	[property nonempty]
Many	[property nonempty]
Empty	[property empty]

Environments:

Budget Categories with parents in list:

none	[if nonempty]
one or more	[if nonempty]

C.6.4 Budget Category List Filtered By Parent:

Parameters:

Budget Category List:

One	[property nonempty]
Many	[property nonempty]
Empty	[property empty]

Environments:

Budget Categories with children in list:

none	[if nonempty]
one or more	[if nonempty]

C.6.5 Budget Category List Filtered By Delete:

Parameters:

Budget Category List:

One	[property nonempty]
Many	[property nonempty]
Empty	[property empty]

Environments:

Deleted budget categories in list:

none	[if nonempty]
one or more	[if nonempty]

C.6.6 Budget Category List Filtered By Type:

Parameters:

Budget Category Type List:

One	[property nonempty]
Many	[property nonempty]
Empty	[property empty]

Environments:

Type of budget categories in list:

none	[if nonempty]
one or more	[if nonempty]

C.6.7 Transaction List Filtered By Source:

Parameters:

Transaction List:

One	[property nonempty]
Many	[property nonempty]
Empty	[property empty]

Environments:

Matches in list:

none	[if nonempty]
one or more	[if nonempty]

C.6.8 Transaction List Filtered By Date:

Parameters:

Transaction List:

One	[property nonempty]
Many	[property nonempty]
Empty	[property empty]

Parameters:

Start date and end date:

Start date before end date	[property date]
end date before start date	[error]

Environments:

Matches in list:

none	[if nonempty and date]
one or more	[if nonempty and date]

C.6.9 Transaction List Filtered By Search:

Parameters:

Transaction List:

One	[property nonempty]
Many	[property nonempty]
Empty	[property empty]

Parameters:

Cleared Filter Key:

All cleared	[if nonempty]
cleared	[if nonempty]
not cleared	[if nonempty]

Parameters:

Reconciled Filter Key:

All reconciled	[if nonempty]
reconciled	[if nonempty]
not reconciled	[if nonempty]

Parameters:

Search Text:

good text	
empty	
null	[error]

Parameters:

Date Filter:

All dates	[if nonempty]
today	[if nonempty]
yesterday	[if nonempty]
this week	[if nonempty]
this semi month	[if nonempty]
last semi month	[if nonempty]
this month	[if nonempty]
last month	[if nonempty]
this quarter	[if nonempty]
last quarter	[if nonempty]
this year	[if nonempty]
last year	[if nonempty]

Environments:

Matches in list:

none	[if nonempty]
one or more	[if nonempty]

C.6.10 Scheduled Transaction List Filtered By Before today:

Parameters:

Scheduled Transaction List:

One	[property nonempty]
Many	[property nonempty]
Empty	[property empty]

Environments:

Matches in list:

none	[if nonempty]
one or more	[if nonempty]

C.7 ModelFactory

C.7.1 Create Document:

Parameters:

File:

good file name	[property name]
omitted	[property empty]

Environments:

Matching fields:

none	[if name]
one or more	[if name]

Auto save:

true	[if name]
false	[if name]

C.7.2 Create Transaction:

Parameters:

Date:

Valid	
not valid	[error]

Description:

one or more characters
omitted

Amount:

Positive
Negative

To:
Budget Category
Account

From:
Budget Category
Account

C.7.3 ModelObjectImpl

C.7.4 Create Scheduled Transaction:

Parameters:

Name:
Valid
not valid [error]

Message:
one or more characters
omitted

Start date:
Valid
not valid [error]

End Date:
Valid
not valid [error]

Frequency:
Positive
Negative [error]

Schedule day:
In range
out of range [error]

Schedule Week:
In range
out of range [error]

Schedule Month:
In range
out of range [error]

Description:
one or more characters
omitted

Amount:
Positive
Negative

To:
Budget Category
Account

From:
Budget Category
Account

C.7.5 Create Budget Category:

Parameters:
Name:
Valid
not valid [error]

Budget Category Type:
Valid
not valid [error]

Income:
true
false

C.7.6 Create Account Type:

Parameters:
Name:
Valid
not valid [error]

Credit:

true
false

C.7.7 Create Account:

Parameters:
Name:
Valid
not valid [error]

Type:
Valid
not valid [error]

C.8 ScheduledTransactionImpl

C.8.1 Name

Parameters:
Name:
Valid
not valid [error]

C.8.2 Message

Parameters:
Message:
one or more characters
omitted

C.8.3 Start Date

Parameters:
Date:
Valid
not valid [error]

C.8.4 End Date

Parameters:
End Date:
Valid
not valid [error]

C.8.5 Frequency

Parameters:
    Frequency:
        Positive
        Negative [error]

C.8.6 Schedule Day

Parameters:
    Schedule day:
        In range
        out of range [error]

,

C.8.7 Schedule Week

Parameters:
    Schedule Week:
        In range
        out of range [error]

C.8.8 Schedule Month

Parameters:
    Schedule Month:
        In range
        out of range [error]

C.8.9 Description

Parameters:
    Description:
        one or more characters
        omitted

C.8.10 Amount

Parameters:
    Amount:
        Positive
        Negative

C.8.11 To

Parameters:

To:

Budget Category
Account

C.8.12 From

Parameters:

From:

Budget Category
Account

C.9 SourceImpl

C.9.1 Name

Parameters:

Name:

Valid
not valid [error]

C.9.2 Notes

Parameters:

Name:

Valid
not valid [error]

C.9.3 Deleted

Parameters:

Deleted:

true
false

C.10 Transaction

C.10.1 Create Transaction:

Parameters:

Date:

Valid
not valid [error]

Description:
one or more characters
omitted

Amount:
Positive
Negative

To:
Budget Category
Account

From:
Budget Category
Account