INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Investigating the Use of Analysis Contracts to Improve the Testability of Object Oriented Code

By

Hong Sun, B. Eng.

**A thesis submitted to the Faculty of Graduate Studies and
Research in partial fulfillment of the requirements of the
degree of
Masters of Engineering TTM**

**Telecommunication Technology Management
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, K1S 5B6
Canada**

March 2002

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-72085-3

Canada

The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

**Investigating the Use of Analysis Contracts to
Improve the Testability of Object Oriented Code**

Submitted by

Hong Sun, B. Eng.

in partial fulfillment of the requirements for
the degree of Master of Engineering

Chair, Professor Rafik A. Goubran

Thesis Co-Supervisor, Professor L. Briand

Thesis Co-Supervisor, Professor Y. Labiche

Carleton University

May 2002

ABSTRACT

A number of activities involved in testing software are known to be difficult and time consuming. Among them is the definition and coding of test oracles and the isolation of faults once failures have been detected. Through a thorough and rigorous empirical study, we investigate how the instrumentation of contracts could address both issues. Contracts are known to be a useful technique to specify the precondition and postcondition of operations and class invariants, thus making the definition of object-oriented analysis or design elements more precise. It is one of the reasons the Object Constraint Language (OCL) was made part of UML. Our aim in this paper is to reuse and instrument contracts to ease testing. A thorough case study is run where we define OCL contracts, instrument them using a commercial tool, and assess the benefits and limitations of doing so to support the automated detection of failures and the isolation of faults. As contracts can be defined at various levels of details, we also investigate the cost and benefit of using contracts at different levels of precision. We then draw practical conclusions regarding the applicability of the approach and its limitations.

Acknowledgments

I would like to thank my supervisors, Lionel Briand and Yvan Labiche, for their understanding, guidance, support, and feedback.

I would like also to thank my family for their continued support.

Table of Contents

CHAPTER 1	INTRODUCTION	1
CHAPTER 2	RELATED WORKS	3
CHAPTER 3	OBSERVABILITY AND DIAGNOSABILITY	7
3.1	OBSERVABILITY	7
3.2	DIAGNOSABILITY	11
CHAPTER 4	THE DEFINITION OF CONTRACTS	13
4.1	CONTRACT DEFINITION RULES.....	13
4.2	ISSUES	15
4.2.1	<i>Level of detail in postconditions.....</i>	<i>15</i>
4.2.2	<i>Delegation and postconditions.....</i>	<i>17</i>
4.3	CONTRACT EXAMPLES FROM THE ATM CASE STUDY	18
4.4	SELECTION OF AN INSTRUMENTATION TOOL AND IMPLEMENTATION RULES20	
4.4.1	<i>Selection of an instrumentation tool.....</i>	<i>21</i>
4.4.2	<i>Instrumentation guidelines.....</i>	<i>23</i>
CHAPTER 5	THE ANALYSIS OF CONTRACTS AS A TOOL FOR TESTABILITY	30
5.1	CONTRACTS AS A SUBSTITUTE TO TEST ORACLES.....	30
5.2	CONTRACTS AS A TOOL TO IMPROVE DIAGNOSABILITY	32
CHAPTER 6	CASE STUDY DESCRIPTION	37
6.1	THE ATM SYSTEM.....	37
6.2	SEEDING OF FAULTS.....	38
6.3	TEST CASES DESIGN.....	42
6.4	EXPERIMENTAL PROCEDURE	43
CHAPTER 7	ANALYSIS RESULTS	45
7.1	THE DETECTION OF FAILURES.....	45
7.2	THE DIAGNOSABILITY OF FAULTS.....	47
CHAPTER 8	VALIDITY	51
CHAPTER 9	AUTOMATION TOOL	53
9.1	REQUIREMENTS.....	53
9.2	DESIGN.....	54
9.3	IMPLEMENTATION	56
9.4	USAGE.....	56
9.5	SUMMARY	58
CHAPTER 10	CONCLUSIONS	60
APPENDIX A:	EXAMPLES OF MUTANTS NOT KILLED BY CONTRACTS ...	64

APPENDIX B: CONTRACTS AT DIFFERENT LEVELS OF PRECISION.....	69
APPENDIX C: CONTRACT FOR ATM SYSTEM.....	72
APPENDIX D: ROOT CAUSE OF MUTANTS NOT KILLED BY THE CONTRACTS.....	116
APPENDIX E: TEST CASES DESIGNED FOR ATM SYSTEM.....	120
APPENDIX F: MUTANTS DESIGNED FOR ATM SYSTEM CASE STUDY	143
APPENDIX G: DIAGNOSABILITY OF INDIVIDUAL MUTANT AND ANALYSIS.....	150

List of Figures

Figure 1 Sensitivity analysis for Observability.....	11
Figure 2 Instrumentation of an invariant, pre- and postcondition with Jcontract	22
Figure 3 Instrumentation of a postcondition using the @assert tag with Jcontract	26
Figure 4 Example where the postcondition cannot be fully checked.....	28
Figure 5 Diagnosability measure: example.....	35
Figure 6 Class diagram (simplified) for the ATM system	38
Figure 7 Number of faults seeded per mutation operator.....	41
Figure 8 Number of faults seeded per class	42
Figure 9 Average Diagnosability	48
Figure 10 Diagnosability Distribution.....	49
Figure 11 Class Diagram of MIAT	54
Figure 12 Directory Structure of Mutant programs	55
Figure 13 GUI of Mutant Insertion	56
Figure 14 GUI of Testing Driver.....	57
Figure 15 Mutant program not killed: example of Situation 1.....	64
Figure 16 Mutant program not killed: example of Situation 2.....	65
Figure 17 Mutant program not killed: example of Situation 3.....	66
Figure 18 Mutant program not killed: example of Situation 4.....	67
Figure 19 Contracts for method Session::doFailedTransactionExtension()	67
Figure 20 Mutant program not killed: example of Situation 5.....	68
Figure 21 Postcondition for Bank: initiateWithdrawl() at the highest precision level.....	69
Figure 22 Postcondition for Bank: initiateWithdrawl() at the intermediate precision level.....	70
Figure 23 Postcondition for Bank: initiateWithdrawl() at the lowest precision level.....	71
Figure 24 Sequence Diagram of Mutant 27	152
Figure 25 Sequence Diagram of Mutant 51	153
Figure 26 Sequence Diagram of Mutant 78	155
Figure 27 Sequence Diagram of Mutant 93	156
Figure 28 Sequence Diagram of Mutant 101	157

List of Tables

Table 1 Involvement of attributes, parameters and return values in contracts.....	14
Table 2 Level of detail in postconditions (example).....	16
Table 3 Contract examples for the ATM	20
Table 4 Comparason of Three Contract Instrumentation Tools.....	21
Table 5 Mutant operator examples.....	40
Table 6 Category Designed for ATM System Withdrawl Transaction.....	43
Table 7 Mutant Detection Results for the ATM system	45
Table 8 Parameter, Category and Choices for Use Case 001.....	120
Table 9 Test Cases for Test Suite 001	121
Table 10 Parameter, Category and Choices for Use Case 002.....	123
Table 11 Test Cases for Test Suite 002.....	123
Table 12 Parameter, Category and Choices for Use Case 003.....	126
Table 13 Test Cases for Test Suite 003.....	126
Table 14 Parameter, Category and Choices for Use Case 004.....	129
Table 15 Test Cases for Test Suite 004.....	129
Table 16 Parameter, Category and Choices for Use Case 005.....	132
Table 17 Test Cases for Test Suite 005.....	132
Table 18 Parameter, Category and Choices for Use Case 006.....	134
Table 19 Test Cases for Test Suite 006.....	135
Table 20 Parameter, Category and Choices for Use Case 007.....	136
Table 21 Test Cases for Test Suite 007.....	136
Table 22 Parameter, Category and Choices for Use Case 008.....	138
Table 23 Test Cases for Test Suite 008.....	139
Table 24 Test Cases for Test Suite 009.....	140
Table 25 Mutants Designed for ATM System	143
Table 26 Diagnosability of Individual Mutant.....	150

Chapter 1 Introduction

One important issue in testing software systems is to develop test oracles that can automatically determine, within the test driver, whether a test case is successful or not. Developing such test oracles manually is expensive and represents a major cost when writing test drivers. Furthermore, there is no general solution or methodology for their construction, thus making them frequently complex and error-prone [Beizer 1990]. Another issue, which is especially critical in object-oriented systems, is to perform a diagnosis of a failure in order to determine the fault(s) that caused it. Again, such a debugging activity is extremely time-consuming as it is not uncommon for developers to spend days or weeks isolating faults in a large system. This problem is made more acute in object-oriented designs where functionality is spread across large numbers of objects.

A common practice, first introduced in the Fusion methodology [Coleman 1994] during object-oriented Analysis, is to define contracts for all public methods and classes in the Analysis class diagram, i.e., pre- and postconditions for methods, class invariants for classes [Meyer 1992]. The goal is to make the Analysis model more precise and complete. In the context of the Unified Modeling Language (UML), such contracts can be expressed in the Object Constraint Language [Warmer 1999]. The question is then whether we can reuse such contracts to address both the test oracle and diagnosability issues. In short, can we leverage Analysis contracts for the purpose of supporting testing as well? More precisely, we want to investigate the following questions:

- Can instrumented contracts be used as a substitute to hard-coded test oracles in the test drivers?

- Can instrumented contracts be used to significantly lower the effort of locating faults after the detection of a failure?

Furthermore, because contracts can be defined at various levels of precision (and therefore complexity), we need to investigate whether defining and instrumenting very precise contracts brings real benefits considering the additional cost they entail.

The paper first discusses related works and motivates our study. Then section 4 provides detailed information about the way contracts were defined. Chapter 5 describes the overall strategy we followed to investigate the above questions. Chapter 6 reports on the detailed setting and procedures of our case study. Chapter 7 provides detailed results. The validity of our study is discussed in Chapter 8. Chapter 9 gives a brief introduction on the case study automation tool and conclusions are drawn in Chapter 10.

Chapter 2 Related Works

A number of authors have discussed the use of executable assertions to improve testability and practitioners have been using them for decades¹. Several directions have been investigated with respect to the definition and usage of assertions. Rosenblum [Rosenblum 1995] reports on several experiments that led to a classification of the assertions that were most effective at detecting faults in C programs. An automatic process for the identification of assertion insertions is described in [Voas 1995a], the objective being to find statements where faults are likely to hide and then specify an assertion for these statements. In this pioneering work on assertions, Voas proposes a technique that is based on mutation testing and does not require an oracle. However it requires very large numbers of executions, even for small programs. As an example, [Voas 1991] reports on a program that computes a solution to the quadratic equation ax^2+bx+c and requires tens of thousands of executions. In [Cousot 2000], to the opposite end of the spectrum, the authors investigate abstract testing, an extension of program static analysis aiming at verifying user-provided specifications by abstract interpretation of the program semantics. This strategy does not require program execution (only static analysis) and can detect errors such as readings of non-initialized variables, buffer overflows, and invalid arithmetic operations (e.g., division by zero). However, functional failures that do not result from these kinds of errors cannot be detected.

In the context of object-oriented systems, Binder [Binder 1999] discusses the advantages of so-called Built-In Test (BIT) oracles, based for example on class invariants and responsibility-driven executable assertions. One issue that he raises is that although

¹ In [Rosenblum 1995], Rosenblum briefly explains how assertions have been used as an aid to software development for more than 30 years.

an assertion can detect out-of-range and some in-range – but incorrect – inputs, it is not clear whether the majority of incorrect results would necessarily cause a run-time assertion violation. In other words, Binder indirectly poses the question of whether assertions, for example based on contracts, can be used as effective test oracles. In such a case, the advantage would be that the oracles could be derived from the contracts in the Analysis documents and instrumented using an assertion tool supporting contracts. The investigation of failure detection effectiveness of instrumented contracts is one important objective of my thesis.

A strategy for implementing BIT support was already proposed in [Wang 1997], showing how class test drivers and oracles could be embedded in the class itself. The basic idea developed in [Jezequel 2001] is similar but they go further by suggesting a procedure to use Design by Contract [Meyer 1992] to implement more effective BIT support by instrumenting contracts as assertions. In order to clearly identify the benefits of instrumenting contracts, [Baudry 2001] proposes definitions of measures for two quality factors, namely robustness and diagnosability, though the definition of the latter one is not precise and operational. Robustness captures the degree to which the software is able to recover from internal faults that would otherwise have provoked a failure, and diagnosability captures the fault isolation effort after the occurrence of a failure. We further discuss the definitions and measurements of robustness and diagnosability in the following section.

Several case studies are briefly reported in [Baudry 2001]. Based on the three systems analyzed, the authors suggest that robustness is quickly enhanced by the introduction of contracts. It is however unclear how this was measured and on which

quantitative basis these conclusions are drawn. What is the rationale followed for the definition of contracts? Are contracts exclusively based on analysis information or also included design and implementation information? It is therefore important to perform a well-designed, precise study that looks into the matter. Furthermore, measuring improvements in robustness due to contracts is important but not sufficient. We need to better characterize the mechanisms by which contract assertions fail to detect failures and determine whether some strategies could be devised to instrument contract assertions in a more effective way.

As a side issue, we believe that Robustness is a misnomer as the only thing we consider here is the probability of detecting a failure through contract violation. A better term would be *observability*, one of the two components of testability [Binder 1999], that we will use in the remainder of this text.

In this thesis, we perform a carefully designed and reported case study that furthers existing work in five ways:

- It provides a precise, operational measure for diagnosability;
- It investigates the effectiveness of contract assertions in improving observability and diagnosability in object-oriented systems by following a precise contract definition procedure, exclusively based on Analysis information;
- It assesses the impact of simplifying contracts on observability and diagnosability, for the sake of easing instrumentation;

- It investigates the mechanisms by which failures remain undetected (not observed) or are detected far away from their corresponding fault location (low diagnosability);
- It analyzes the best ways to instrument contracts with a particular emphasis on Java systems.

Chapter 3 Observability and Diagnosability

In the following subsections, we go more in depth in discussing the two testability factors we are assessing. We review the work of [Baudry 2001], perform a sensitivity analysis of an observability model, and propose a new rationale to define diagnosability.

3.1 Observability

A mathematical model of Observability is provided in [Baudry 2001]. The Observability of a system (called global Observability) composed of a set of interconnected components, is defined as the probability that an internal fault is detected by any one of the components. The *weakness* of a system is the opposite probability: the probability that a fault is not detected. Let $P'_i(S)$ being the probability that a fault in component i , part of system S , is not detected in S . $P'_i(S)$ is the weakness of system S on faults in i . The mathematical expression of $P'_i(S)$ then relies on the following observation: an internal fault in a component plugged into a system can be detected either by the component itself or by one of its clients (or children). For simplicity, the authors only consider faults that are detected by a component directly dependent on the faulty one, even though dependences are transitive. Then, $P'_i(S)$ is the probability that the fault is not detected by component i , nor by its direct clients and children: $P'_i(S) = P'_{ii} * \prod_j P'_{ij}$ where P'_{ij} is the probability that a fault in component i is *not* detected by the direct client j of i . Then the weakness of system S is $P'(S) = \sum_i P_{fail}(i) * P'_i(S)$ where $P_{fail}(i)$ is the probability a failure comes from executing a fault in component i . Let $P(S)$ being the

Observability of system S , and P_{ij} being the probability that a fault in component i is detected by a direct client j of i . The previous expression can be transformed as follows:

$$1 - P(S) = \sum_i P_{fail}(i) * (1 - P_{ii}) * \prod_j (1 - P_{ij}) \quad (1)$$

In [Baudry 2001], the authors investigated their Observability measures (P_{ij} and P_{ii}) on three different software systems. In these three systems, P_{ij} was observed to be proportional to P_{ii} : $P_{ij} = c * P_{ii}$, where c is the proportionality coefficient between the fault detection effectiveness of a class and the fault detection effectiveness of one of its server classes where a fault is assumed to be located. It is assumed that a client class is less effective than its server classes in detecting faults that are in these server classes, thus leading to $0 \leq c < 1$. Based on fault seeding and empirical analysis, Baudry et al estimated c to average at 0.8 (c values are expected to vary across pairs of client/server classes).

As a simplification, assuming that (i) the probability the failure comes from component i is the same for all the other components ($P_{fail}(i) = 1/n, \forall i$, where n is the number of components), and (ii) the probability that a fault in component i is discovered by contracts in i is the same for all the components ($P_{ii} = P_{jj}, \forall i, j$), the authors were able to demonstrate, as expected, a significant increase in Observability of the three systems with the improvement in Observability of their components: $P(S)$ is a function in P_{ii} .

Given assumptions (i) and (ii), the results on the three different systems show that not using contracts results in systems that are not robust, but that adding simple contracts

improves significantly their Observability². In addition, it is noted that the three Observability curves for the three systems are similar and the most plausible explanation seems to be that similar dependency densities between classes can be found across the systems. It was also shown on one system, that a change in c (e.g., 0.2 instead of 0.8) had a small impact on the system Observability $P(S)$.

Though a model was proposed, no systematic sensitivity analysis of $P(S)$ against these two factors (coefficient c and class dependency density k) was performed. Based on the above simplifying assumptions, we derive from (1) the following formula:

$$P(S) = 1 - (1 - P_{ii}) * (1 - c * P_{ii})^k \quad (2)$$

where k denotes the number of clients of class i (class dependency density).

Based on (2), Figure 1 shows how $P(S)$ relates to P_{ii} for varying values of c and k . In Figure 1 (left), c is constant and k takes values 1, 3, and 6³. In Figure 1(right), k is constant and c takes values 0.2, 0.5, and 0.8. From these figures, it is clear that $P(S)$ is very sensitive to these two factors. One important conclusion we can draw from the sensitivity analysis is that P_{ii} does not need to approach 1 for contracts to ensure a high probability of failure detection. After a certain threshold, in many (c, k) configurations, $P(S)$ reaches a plateau close to 1 for values of P_{ii} that are significantly below 1. For example, when $c=0.8$ and $k=3$, we reach a plateau when $P_{ii} \sim 0.75$. This suggests that contracts do not need to be highly precise or complete to be useful. Considering that k

² A definition of 'simple' contracts is not provided though.

³ It is reasonable to expect classes to show, on average, at least one dependency and at most six. The average class dependency density in the three cases studies described in [Baudry 2001] are 1.9, 2.6, and 2.8, respectively.

may vary significantly from system to system and that c is very difficult to estimate⁴, we can conclude that sensitivity analysis gives us little insight into the potential effectiveness of instrumented contracts. Furthermore, it is difficult to assess the impact of the simplifying assumptions we had to make above on the validity of the results we obtained. Therefore, empirical investigations are crucial if we want to answer any question about contracts and observability.

Based on the analysis of three systems, [Baudry 2001] suggests that diagnosability is quickly enhanced by the introduction of contracts. For instance, they improve contracts for one of their systems and observe an increase in observability from 58.5% to 87.5%. Similar results are also reported in [Baudry 2000]. However, the authors do not provide the rationale for the determination of the first set of contracts nor the procedure applied for their improvement. It is also unclear whether contracts were exclusively based on analysis information or also included design and implementation information. It is therefore important to perform a well-designed, precise study that looks into the matter. Section 5.1 describes how we measure observability in the context of our case study.

⁴ It depends, to a large extent, on how precise the contracts are, the type of dependencies between clients and servers, the functionality of the classes.

Figure 1 Sensitivity analysis for Observability

3.2 Diagnosability

Regarding diagnosability, [Baudry 2001] does not provide a mathematical model similar to that of Observability, but only briefly comment the main ideas and results. Once a failure has been revealed diagnosability relates to the ease with which the corresponding fault(s) can be isolated. With this definition in mind, diagnosability can be measured as the effort involved in isolating faults. Since this is in practice difficult to measure and is subject to random fluctuations depending on the type and location of faults, diagnosability can indirectly and objectively be estimated based on the maximum size of the code to be analyzed to ensure the fault will be detected. This is a reasonable indicator as the isolation effort is related to the size of the sets of potentially faulty methods or statements.

[Baudry 2001] proposes to base their estimation of diagnosability on the concept of *indistinguishability set*. This is the set composed of statements bounded by two consecutive contracts in an execution flow. The intuitive idea is that, when a failure is observed due to a contract violation, the statements to be investigated (which cannot be distinguished in terms of their likelihood to contain a fault) are between the last contract

to execute successfully and the contract violated. The larger the indistinguishability set, the more expensive the fault diagnosis.

Since [Baudry 2001] only provides the main ideas and results, some elements of the approach remain to be clarified. For instance, it is not clear whether the definition of indistinguishability set is adequate to measure diagnosability. The statements in this set may turn out not to be faulty as the fault(s) may have been executed before the last successfully executed contract. In other words, the incorrect internal state of the system can be due to a fault outside of the indistinguishability set, as contracts cannot be expected to be perfect test oracles. Measuring diagnosability that way is at best a rough approximation. Measuring diagnosability must involve an estimation of the statements to be analyzed based on the location of the fault(s). Furthermore, it has to be measured based on an actual set of failures and corresponding faults, not just based on contract information.

In this thesis, we will perform an in-depth investigation of the impact of contract executable assertions (i.e., instrumentation of contracts in code) on diagnosability. As defined in Section 5.2, we are to measure diagnosability in a way that will enable systematic comparisons between programs instrumented with contracts and programs that exclusively use classical test oracles. This measure is based on an estimation of the size of the diagnosis work to be performed by measuring a 'distance' between the location of failure detection and the location of the fault that caused the failure.

Chapter 4 The Definition of Contracts

We are investigating the use of contracts to support testing. But what do we mean by contract? Though the notion of design by contract [Meyer 1992] is well established, the level of detail of those contracts and the way they are defined needs to be specified as it may vary a great deal in practice. Therefore, assessing the impact of contracts on testing is not meaningful if we do not define precisely how those contracts are devised. In other words we need to define precisely what we are evaluating. Based on experience, we define a set of practical rules we have been using and refining to define our contracts. We also provide some representative examples, and explain what have been the implementation issues and procedures to instrument those contracts. Recall that we consider contracts to be defined during Analysis, in our case by using the Object Constraint Language (OCL) in the context of the UML notation [Warmer 1999, OMG 2001]. This choice was made because UML is now a de-facto standard notation for object-oriented Analysis and OCL is the most natural way to precisely define contracts in that context⁵. At the same time, most of the tools used in instrumenting the contracts are designed according to the OCL notions. Defining contracts during Analysis entails that our contracts must be based exclusively on application domain knowledge and are instrumented in the application domain classes of the system and their public methods.

4.1 Contract Definition Rules

In order to be systematic and complete, it is recommended that contracts be defined by following some guidelines that can be formalized as rules. It is particularly important in the context of this paper to precisely define the conditions under which we

obtained the results presented in Chapter 7. The rules defined below apply in the context where OCL contracts are reused from Analysis documents. Contracts have three types of components (class invariant, method pre-condition, method postcondition) and may involve five kinds of elements: class attributes, class associations, method input or output parameters, and method return values. We systematically identify what kind of elements may be needed for a given type of contract, as described in Table 1.

Types of elements	Contracts		
	Invariant	Pre-condition	Postcondition
Attribute	X	X	X
Input parameter		X	X
Output parameter			X
Return value			X
Association		X	X

Table 1 Involvement of attributes, parameters and return values in contracts

Class attributes are systematically involved in class invariants, pre- and postconditions as checking the value of attributes and their relationships is a key element of contracts. To be more precise, the usual rules followed to define contracts based on attributes and parameters are:

- (1) Check whether attributes and input and output parameters are within the allowed domain of values;
- (2) Check whether expected relationships among attributes' and input and output parameters' values hold.

⁵ This also addresses the difficult challenge, mentioned in [Voas 1995b], of finding/designing a language in

Furthermore, postconditions typically describe the addition or destruction of association links whereas pre-conditions verify that certain links are present or absent so as to enable the correct execution of a method. Associations are not involved in class invariants as constraints on class associations are already modeled with multiplicities in class diagrams. Therefore multiplicities are not involved in our invariants, even though it would be good practice to use multiplicities in class diagrams to augment code assertions checking invariants.

Output parameters and return values are part of the methods' behavior to be modeled by postconditions. Since neither class invariants nor pre-conditions involve the modeling of method effects, they do not use output parameters or return values.

4.2 Issues

In addition to the above general rules, several remarks should be made regarding the level of detail in postconditions and the use of delegation in methods.

4.2.1 Level of detail in postconditions

A postcondition may involve returned values, a number of modifications of attributes or output parameters, and these may vary according to different conditions under which the method is executing. For instance, suppose the specification of method `int Result (int x, int y)` indicates that: when $x > y$, the return value is 1; when $x < y$, the return value is 2, and when $x = y$, the return value is 3. In that situation, like in many others, we can define the corresponding postcondition at different levels of precision, as illustrated in Table 2. It is clear that these postconditions show very different

which pre and post-conditions are specified by the user.

levels of detail and complexity, but both are correct in the sense that they make a true statement about the effect of the `Result()` method.

```
if (x>y) then result = 1           result >=1 and result <=3
else if (x<y) then result = 2
else      result = 3
endif
endif
```

(Note that this is close to the code structure)

Table 2 Level of detail in postconditions (example)

It is difficult to come up with general guidelines for selecting an appropriate level of precision for specifying postconditions. Nevertheless, we can consider the following issues:

- A weak postcondition may not be able to detect an illegal system state (as defined by the states of its objects and their associations);
- Specifying in great detail a postcondition will help detect failures and locate faults but more effort is required to define the postcondition, check its correctness (the postcondition is more likely to contain faults due to its complex logic), and instrument it. Furthermore, since we do not know how many faults are present and can potentially be caught, the cost-effectiveness of investing in a more precise postcondition is not known beforehand.

Only experience and practice, through case studies such as the one presented here, will tell us about the potential gains of using more detailed contracts. In our study, in order to *systematically* investigate the impact of contract precision on observability and diagnosability, we defined contracts at three levels of precision:

- The highest level of precision, keeping in mind we define Analysis contracts. Here every distinct condition, possibly resulting from a different set of inputs or system state, is distinguished in the postcondition. The motivation in this case is to assess the *maximum*, potential benefit of contracts.
- An intermediate level of precision that only distinguishes conditions for the standard situation (expected execution) from exceptional situations that are also addressed by the method.
- A low level of precision that just defines the ranges/enumerations of values expected as resulting from executing the method.

We provide examples of contract definition at the three levels of precision in Appendix B. The above definitions allow us to work at consistent levels of precision across the system and therefore be systematic in our investigation.

4.2.2 Delegation and postconditions

When a (caller) method returns a value that merely transmits the return value of another (callee) method, one may ask whether a postcondition, which is the exact copy of the callee's postcondition, is to be defined for the caller. In other words, in the following code chunk, is it necessary to check the return value's validity in the postcondition of `valueOfBalance()` since this was checked in `getBalance()`?

```
public int valueOfBalance () {
    return currentAccount.getBalance();
}
```

Three answers can be identified, i.e., the postcondition is defined only for the callee, only for the caller, or both the callee and caller, and we discuss them below according to two criterion, i.e., instrumentation cost and fault detection:

- If the return value's validity is only checked in the callee we avoid redundancy in the checks. However, we cannot detect errors in the caller, e.g., a wrong method call, which would result in a valid or invalid return value. We then rely on a side effect to detect the error.
- If the return value's validity is only checked in the caller we also avoid, to some extent, redundancy in the checks. However, we have to make sure that any caller has this check. Diagnosability is easier than in the previous case because if the return value is invalid, it because of an error in the call or an error in the caller.
- If the return value's validity is checked in both the callee and caller(s), we have redundant checks. However, diagnosability is easier because if a contract is violated, we exactly know whether it is in the caller or callee.

We chose the last solution, considering that adding some redundant checks is not so important compared to all the checks performed in such an experiment. It also provides the best fault detection and diagnosability capabilities. We also discuss below the problems introduced by delegation when *instrumenting* contracts in the source code (Section 4.4.2).

4.3 Contract Examples from the ATM case study

As mentioned above, we use the Object Constraint Language (OCL) to define our contracts. We provide here, in Table 3, some contract examples from the ATM case study (at the highest level of precision). For the complete contracts defined for ATM case study please refers to Appendix C. As a writing convention for contracts, we introduce first the *context* (a class in the case of an invariant, plus a method in the case of pre- and

postconditions) as well as a *label* for the contract component being defined: *inv* for invariant, *pre* for pre-condition, and *post* for postcondition. From these examples (and Appendix B), we can see that the case study involves contracts that are far from being trivial and this is something we have to expect in actual systems. We need instrumentation tools that support OCL to the maximum extent possible and well-defined implementation rules to instrument such contracts systematically in the context of the selected tool. The further the tool's instrumentation language is from OCL, the higher the cost overhead of using contracts as implementation rules. This issue is discussed in the next section.

The meaning of the four example contracts in Table 3 is provided by order of appearance:

- Contract 1: A `Bank` object must satisfy the following: For the current transaction, the card and account must exist and the available balance must be less or equal than the current balance.
- Contract 2: Method `Bank::finishDeposit(...)` requires that the `ATMnumber` exists and that the `serialNumber` be positive. Then, the method ensures that, provided that the `parameter succeeded` is true, the account is credited with the amount of the transaction.
- Contract 3: The method ensures that, according to the value entered with the keyboard, the correct transaction is initiated (deposit, transfer, ...).
- Contract 4: Method `ATM::checkIfCashAvailable(...)` verifies whether the requested amount is less than the amount available in the ATM (it asks the `cashDispenser`).

context: Bank
inv: self._currentTransactionCard->notEmpty() and self._currentTransactionAccount->notEmpty() and self._currentTransactionAccount._availableBalance.getValue()<= self._currentTransactionAccount._currentBalance.getValue()
context: Bank::finishDeposit(ATMnumber:int,serialNumber:int, successful:Boolean) : void
pre: self.atmMachineProfile->exists(_ATMNumber = ATMnumber) and serialNumber >0
post: succeeded implies self._currentTransactionAccount._currentBalance.getValue()= self._currentTransactionAccount._currentBalance.getValue@pre+ _currentTransactionAmount.getValue()
context: Transaction::chooseTransaction(session:Session, atm:ATM, bank:Bank): Transaction
post: let option : integer = atm.getMenuChoice ("Please choose a transaction type:", 4, TransactionMenu) (option = 1 and result oclIsTypeOf WithdrawalTransaction) or (option = 2 and result oclIsTypeOf DepositTransaction) or (option = 3 and result oclIsTypeOf TransferTransaction) or (option = 4 and result oclIsTypeOf InquiryTransaction)
context: ATM::checkIfCashAvailable(amount: Money):Boolean
post: result = not (self.cashDispenser.currentCash().less(amount))

Table 3 Contract examples for the ATM

4.4 Selection of an instrumentation tool and implementation rules

The use of an instrumentation tool has a significant impact on the cost-effectiveness of using contracts to support testing. Its selection is therefore important as well as its effective usage following instrumentation guidelines.

4.4.1 Selection of an instrumentation tool

A tool named Jcontract [JContract], from Parasoft, was selected to instrument the contracts after the comparison (see Table 4) of the available instrumentation tools, for example iContract [iContract], Jass [Jass] and JMSAssert [JMSAssert]. JContract was selected for its ability to handle local variables within contracts (see discussions below) and there is no any bug found after the pilot. Compared to the other three tools, JContract is easier to use and offers more flexibility in terms of control structures in contracts (see Table 4).

	Pro.	Con.
iContract	<ul style="list-style-type: none">• most OCL-like language, especially offers the following operations, implies, forall and exists	<ul style="list-style-type: none">• the generated code is faulty.• many options to deal with, quite complicated to use.• can not use the if-then-else control structure in contracts.
JMSAssert	<ul style="list-style-type: none">• source code not instrumented, but Java Virtual Machine extended.	<ul style="list-style-type: none">• complicated to use.• can not use the if-then-else control structure in contracts
Jass	<ul style="list-style-type: none">• offers three operators to manipulate sets(forall, exists, changeonly), but not the “implies” operation• easy to use.• best documentation.	<ul style="list-style-type: none">• Java source files must be renamed with extension .jass• the source code needs to implement the Cloneable Interface and define the clone() operation for the @pre function.• can not use the if-then-else control structure

Table 4 Comparason of Three Contract Instrumentation Tools

Jcontract provides a tool that instruments and compiles Java code so that extra byte code is added to check pre-conditions and class invariants at method entry and postconditions and class invariants at method exit during run time. If a pre-condition, postcondition, or class invariant is violated then Jcontract may be set up so that a message may be sent to a monitor managed by the tool, an exception can be raised, and/or the

program may be terminated. The language used to instrument contracts (DbC) is similar to OCL⁶ and provides specific Javadoc tags that can be used to specify all required aspects of contracts in the code without side effects. We will use @pre, @post, @invariant, @assert tags as shown in Figure 2 and Figure 3. The information provided by the Javadoc comments is then used by Jcontract to instrument the code. The tag @assert provides a way to check assertions within the method block and this is how we intend to check constraints involving local variables. Local variables are out of the scope of all other tags, though method parameters, method return value (post-condition only), and attributes can of course be accessed.

```

/**   - Javadoc comment -
 * invariant defined just before the class definition
 * @invariant _cents >= 0
 */
public class Money {
    private long _cents;

    /**   - Javadoc comment -
     * pre- and postconditions defined just before the method definition
     * @pre dollars >= 0
     * @post _cents == 100 * (new Integer (dollars)).floatValue()
     */
    public Money(int dollars) {
        _cents = 100L * dollars;
    }
    ...
}

```

Figure 2 Instrumentation of an invariant, pre- and postcondition with Jcontract

Using Jcontract, contracts are evaluated during the execution of a method as follows:

⁶ But it is not nearly as complete and is closer to implementation details as it is used to instrument the code.

- Evaluation of the pre-condition (defined by @pre) and the invariant (defined by @invariant) at method entry, just before the first statement of the method;
- Evaluation of the postcondition (defined by @post) and the invariant (defined by @invariant) at method exit, just after the last statement of the method;
- Evaluation of the assertions (defined by @assert) at their location in the body of the method.

4.4.2 Instrumentation guidelines

4.4.2.1 Side effects in postcondition assertions

A first practical issue when instrumenting contracts is the use of method calls in contract assertions (see examples 3 and 4 in Table 3). The execution of such calls may have side effects⁷ in terms of the modification of the system's state and this is something which is not acceptable as contracts are only supposed to verify the state of the system at some point during execution. We can identify three distinct situations:

1. The method called has no side effect. Then the method can be used in contracts without problem. For example, in the postcondition of method `ATM::checkIfCashAvailable(amount:Money)` (example 4 in Table 3), the two methods `currentCash()` and `less()` have no side effect.

⁷ OCL expressions are side effects free since they can only use method with `isQuery` being true [OMG 2001] (the `isQuery` property specifies whether the execution of the method leaves the state of the system unchanged). However, it may not be practical to define an OCL contract that exclusively contains such methods.

2. The method called has no side effect but requires an input from a user or a device⁸. This is the case in the postcondition of method `Transaction::chooseTransaction()` (example 3 in Table 3) in which method `getMenuChoice()` requests a choice to be entered from the keyboard. If that method is executed again during the evaluation of the postcondition, the choice would need to be re-entered, thus creating the opportunity for inconsistencies. During testing, user or device inputs are typically obtained from a stub, simulating a user entering information on the keyboard.
3. The method called has side effects, e.g., it modifies objects' states or associations. We did not encounter such a situation in the ATM case study but this is probably the most serious situation to watch for as contract executions may end up having an effect on the system state.

In a well-defined pre-condition or invariant, a method call corresponding to situations 2 or 3 is not expected. In a postcondition those situations can be encountered since properties that must hold are verified when the method terminates and the checking of these properties may involve method calls corresponding to situations 2 or 3. We therefore need to find an instrumentation solution for postconditions containing those types of method calls.

The solution involves removing, in some way, the method call from the postcondition and substituting it with something preserving the semantics of the postcondition. A solution consists in storing the result of the method call during execution so that it can be reused in the postcondition. From a practical standpoint, this can be done in two ways:

⁸ Such a method has property `isQuery = true`.

1. We can add an attribute to the caller class (e.g., of type integer, for calling `Transaction::chooseTransaction()`) and use that attribute to record a relevant result after the execution of the callee (e.g., `getMenuChoice()` which returns an integer) in the body of the caller.
2. We can add a local variable to the caller method in order to record the callee's result and use an `@assert` tag to access that local variable and check the postcondition's clauses involving that variable.

The first solution is not recommended as it requires a modification to the system class diagram, thus leading to confusion and a probable cluttering of the diagram. The second solution only requires the introduction of a local variable (see **Error! Reference source not found.** for the implementation of the postcondition of example 2 in Table 3). Such a variable may already exist and coding standards might also be used to ensure this is the case. However, not all tools supporting contracts enable the check of local variables. This capability was one of the major reasons for us to select Jcontract as it became clear, early in our study, that this was unavoidable if the postconditions were to be reasonably complete and effective.

When a local variable must be checked, a good practice is to put the `@assert` tag as close as possible to a return statement (usually just before it), or at the end of the method. This ensures that the value of the variable checked in the assertion is exactly the one that was intended to be checked in the postcondition (but could not because of a method call corresponding to situations 2 or 3).

```

public static Transaction chooseTransaction(Session session, ATM atm,
Bank bank){
    ...
    int option = atm.getMenuChoice("Please choose a transaction type:",
    4, transactionMenu);
    Transaction result; //added variable for the implementation of the
    //contract
    switch (option) {
        case 1: result = new WithdrawlTransaction(session, atm, bank);
            // the original statement was 'return new
            // WithdrawlTransaction(...);'
            break;
        case 2: result = new DepositTransaction(session, atm, bank);
            break;
        case 3: ...
        case 4: ...
        default: result = null; }
    /** - Javadoc comment -
    Assertion defined before (as close as possible to) the return
    statement
    @assert (option == 1)&&(result instanceof WithdrawlTransaction)
    || (option == 2)&&( result instanceof DepositTransaction )
    ...
    **/
    return result;}

```

Figure 3 Instrumentation of a postcondition using the @assert tag with Jcontract

The modification required in the @post tag and the addition of an @assert tag must be performed carefully. Consider for instance the situation in which the postcondition of a method is A or B, where A and B are clauses possibly involving method calls. It implies that, if at method exit the Boolean expression A or B is false (i.e., both A and B are false), an exception is raised. Let us consider the case where A is the only clause that contains a method call corresponding to situations 2 or 3 as defined above. Then an @assert tag verifying A is added just before the end of the method, and clause A is removed from the postcondition (the @post tag). But we also have to remove clause B from the postcondition (the @post tag) and add it to the @assert tag since, otherwise, the Boolean expression evaluated is implicitly A and B rather than A or B:

if B is evaluated alone in tag @post, an exception is raised when A is false (in the @assert), but not when both A and B are false.

An important limitation is that the use of an @assert tag, even placed just before method exit, might not be able to detect a failure due to the execution of a fault in the return statement. This is the case if (part of) the contract involving the return expression is verified before the return statement. In Figure 4, an example shows a case where the postcondition is expressed in terms of result, the returned variable, and instrumented as an @assert tag. Let us assume this method returns a probability between 0 and 1 and that a fault results into returning result+1 instead of result. Such a fault in the return statement cannot be detected by the postcondition instrumentation as:

- The Boolean expression `result >= 0` and `result <= 1` cannot be checked in the postcondition that does not have access to local variables or method return values;
- The assert tag is located before the return statement and hence cannot detect failures due to the fault in the return statement itself.

4.4.2.2 *Access to attribute values*

Other instrumentation issues concern the necessary built-in test support to enable sufficient observability of classes's concrete state and the translation process from OCL to the language used by the instrumentation tool (Jcontract). Regarding the former issue, `get()` methods for non-public attributes as well as methods verifying the state invariant can be necessary and can greatly simplify the instrumentation of contracts. This is discussed and referred to as *built-in test support* in [Binder 1999]. For example, in the

ATM system we often use method `Money::getValue()` for verifying amounts (private attribute in `Money`) in contracts that are not in class `Money`. But this requires that some standard is followed when defining class interfaces such that `get()` and state invariant check methods are available.

```

/**    - Javadoc comment -
 * @pre ...
 * @post ...
 */
public Method() {
    ...
    /** @assert result >= 0 and result <= 1
    */
    return result+1; // fault: should be 'return result'
}

```

Figure 4 Example where the postcondition cannot be fully checked

4.4.2.3 Collections

The translation from an OCL expression of the contract to its implementation using a specific instrumentation tool language (in Jcontract: DbC, a language close to OCL) also requires some attention. Collections are used in OCL to link an object of a class to a set of objects of another class when the multiplicity of the association between these two classes is greater than 1. Then contracts can, for instance, involve assessing the size of a collection or selecting a particular object in a collection. Such collections are eventually implemented with data structures in the source code, thus requiring the translation of the OCL contract into an expression that handles these data structures. This is easily done in our study since contracts in DbC have access to attributes, method parameters, and local variables, and any Java code (e.g., a `for` statement browsing a data

structure) can be used in a DbC contract. In general, this issue should be carefully considered when selecting a tool to support contract instrumentation.

Chapter 5 The Analysis of Contracts as a Tool for Testability

We describe in turn how we plan to estimate observability and diagnosability in our case study.

5.1 Contracts as a Substitute to Test Oracles

One of the most expensive activities during testing is the writing of test oracles in the test drivers. This is also a key activity as we want to ensure we detect all failures and do not waste time on false-positives, that is test cases indicated as failed by the test driver that turn out to pass after further investigation. As discussed in previous work [Binder 1999], contracts could be instrumented and used to serve as test oracles as they define conditions under which the system's state can be considered corrupt. The advantage would be the use of instrumented contracts as test oracles, thus eliminating the need to develop test oracles in each test driver, for each test case. For this to be possible, we need effective instrumentation tools to generate, in the code, executable assertions based on contracts. Furthermore, the contracts need to be reused from Analysis or design, depending on the development methodology in place. They should not be subsequently refined or modified for the sake of improving the test oracles as the economic argument of leveraging them to improve testing would be less compelling.

The procedure we use to investigate whether contracts are effective test oracles is as follows:

- We define, on an example system, contracts in OCL. Those contracts are intentionally defined to be at the level of what could be expected during Analysis (e.g., in a Fusion Analysis). In other words they are based on application domain knowledge, not implementation knowledge. We also

derive simplified versions of the contracts at their intermediate and low levels of precision.

- We implement the contracts using the language provided by a contract instrumentation tool. In our case study, we selected Jcontract from Parasoft [JContract], which provides a language (DbC) close to OCL⁹. Contracts are defined through Javadoc comments, in order to ensure they do not affect the execution of the delivered system.
- We randomly seed faults in the system, using a set of predefined mutation operators for Java [Kim 1999, Kim 2000a]. When seeding faults, the probability of selecting a mutation operator over another depends on the characteristics of the code, e.g., extent of inheritance usage, that determine what the opportunities are to consider a specific operator. For each seeded fault, we generate four mutant programs (i.e., faulty program versions): with (at the three levels of precision) and without contract assertions (i.e., using test oracles in test drivers). Test oracles here are implemented by comparing the mutant program outputs (i.e., an ordered set of strings) with the output of the correct program.
- We develop test cases (Appendix E) based on a black-box technique, i.e., category-partition [Ostrand 1988]. Totally we defined 51 test cases according to the ATM system specification (see section 6.3). Though our goal here is not to achieve ‘complete’ testing, we need to make sure the functionalities of the

⁹ Ideally, we would like such a language to be identical to OCL, to ease the transition from the analysis document to the code. But current tools, though improving, have not reached that stage yet.

system are reasonably exercised to provide contracts with the opportunity to execute seeded faults and detect failures.

- We run the test cases with the four mutant programs. We measure effectiveness of failure detection in each case and, for the instrumented system, check how many failures are caught by contract assertion violations (contract broken exception). This is used to assess the suitability of contract assertions as a substitute to test oracles and potential improvements in observability due to contract instrumentation.
- When failures are not detected by contracts, we investigate the root cause and try to generalize from all instances.

5.2 Contracts as a Tool to Improve Diagnosability

Isolating faults in an OO system is time-consuming as functionality is typically distributed across many objects. In that context, contract assertions can be helpful in order to reduce the number of methods and source code statements one has to look at to locate faults when failures are detected. When contracts are instrumented, the detection point — the contract that raises an exception — is expected to be significantly closer to the faulty statement(s) than an output statement making the failure visible to the user when contracts are not used. Moreover, since most methods in OO systems usually consist of a few lines of code, it is expected that contracts should be violated not long after faulty statements are executed. This should help failure diagnosis and fault isolation.

In order to assess how effective contracts are to improve testability, we need to define a diagnosability measure that enables objective, quantitative comparisons. Such a measure must be defined based on the *Diagnosis Flow* that determines the sequence of

methods one has to investigate from the detection of a failure to the location of the faulty statement. We define the diagnosis flow as a model of what would be common practice when testers detect a failure and search for faulty statements: They know the location of the failure (i.e., the violated contract or the output statement revealing the failure), and then they recursively browse the flow of methods executed from the method that contains the violated contract/output statement, say method A, back to the callers of A. Admittedly, like any model, this is a simplification of reality where a developer may have so much expertise that she uses shortcuts to identify faulty statement(s). But such simplifications are necessary for performing a systematic, objective study of diagnosability.

We use the following rules to determine the methods that appear in the diagnosis flow:

Starting method in the diagnosis flow:

The starting method depends on the location of the raised exception. As illustrated in Figure 5 (Figure 5.a), an exception can be raised in a method (method *e*) at one of the following locations: (1) At method's entry, i.e., the precondition or the invariant as evaluated at method's entry is violated; (2) During the execution of the method¹⁰ (an assertion is violated); (3) At method's exit (the postcondition or the invariant as evaluated at method's exit is violated). In case (1), the starting method in the diagnosis flow is the method that called method *e*, because the exception is raised before the execution of any statement of *e*. On the contrary, for cases (2) and (3), the faulty statement can be in method *e* and the starting method in the diagnosis flow is therefore method *e* itself. If we use

oracles instead of contracts, the detection of the failure is an output statement. The starting method in the diagnosis flow is then the method that contains this output statement.

Sequence of methods in the diagnosis flow:

When a method is investigated, one starts the investigation at the beginning of the method and then looks at the different flows of control in the method. During that process, each time a method call is encountered, the called method is (recursively) investigated. When the end of the method is reached, the investigation continues with the caller method. The diagnosis flow stops when a faulty statement is discovered in a method¹¹. Note that, if a method is involved several times in the diagnosis flow, the method is not investigated more than once unless different flows of control are executed each time.

The diagnosability measure we associate with the diagnosis flow determines the distance, in terms of number of methods investigated (that appear in the diagnosis flow), between the detection point (the violated contract or the output statement) and the faulty statement (where the mutation operator was applied). The measure can be further refined according to the complexity of the methods encountered, to give more weight to the more complex methods (e.g., with many different flows of control). This is an open issue that will be investigated in future work. However, though an approximation, we think the number of methods is a reasonable indicator since methods usually consist of a small number of lines of code.

¹⁰ Jcontract, unlike other tools supporting contracts, enables the check of assertions in the body of methods.

¹¹ This is a simplification justified in the context of our study since we insert faults using mutation operators and each failure can be traced back to a unique fault.

The use of a sequence diagram defined during Analysis or Design (or its construction if it does not exist) can help the determination of the diagnosis flow, as well as its visualization. We use this representation in this thesis, both for examples (as in Figure 5.b) and for the case study.

Figure 5 Diagnosability measure: example

According to this procedure, the diagnosis flows and the corresponding diagnosability measures for the example in Figure 5.b are determined as follows:

- If the exception is raised at the entry of method *e*, the diagnosis flow begins with method *a*. Then methods *b* and *c* are investigated in sequence. Method *d* is not investigated because the faulty statement is discovered before the call. The diagnosis flow is sequence [*a*, *b*, *c*] and the diagnosability measure equals to 3.
- If the exception is raised during the execution of *e*, but before the call to *f*, the diagnosis flow begins with method *e*. Method *f* is not investigated because the fault is revealed before its call in method *e*. The diagnosis flow is sequence [*e*, *a*, *b*, *c*] and the diagnosability measure equals to 4.

- If the exception is raised in e after the call to f , then e starts the diagnosis flow, and f is investigated. The diagnosis flow is sequence $[e, f, a, b, c]$ and the diagnosability measure equals to 5.

Given our measure for diagnosability, the procedure we use to determine how effective contract assertions are to improve the diagnosis of failures is as follows:

- We use an example system, its associated contracts and test cases, as discussed in Section 5.1. The test case executions give (i) the mutant programs killed when contracts are used (at the three levels of precision), and (ii) the mutant programs killed when test oracles are used. Based on test executions, for all the mutant programs that are killed by both contracts and oracles, we determine the diagnosis flows and compute the diagnosability measures for both mutant versions with contract assertions and with test oracles only, respectively. Since several test cases may kill the same mutant, and since they can involve different outputs (when the oracles are used alone) or failed contracts (when contracts are used alone), we may obtain different diagnosability values for one mutant program. Therefore, we define the diagnosability associated with a mutant as the average diagnosability of all the test cases that kill it.
- We compare the distributions of diagnosability values for both mutant versions in order to determine whether contract assertions make a practically significant difference.

Chapter 6 Case Study Description

We first present the system we used as a case study and justify its selection. Then, we present our strategy for seeding faults and the experimental procedure we followed to investigate the benefits of contracts in terms of failure detection and diagnosability.

6.1 The ATM System

An Automated Teller Machine (ATM) system was selected as an example since it was deemed of adequate complexity and the application domain is intuitive, thus facilitating the definition of contracts. Furthermore, the system was not designed or coded by the authors, represents a complete set of functionalities, and has a realistic design that is far to be perfect in terms of object definitions and responsibility assignment. Furthermore, the implementation language is Java, for which a number of contract instrumentation tools exist. The ATM is composed of 21 classes (see class diagram in Figure 6), a variety of associations (including aggregation, inheritance, use dependencies), and the Java source code contains 2200 LOCs (without comments).

However, this system did not come with a well-defined analysis model. So before we started defining the contract an Analysis class diagram was re-engineered from the existed source code.

Figure 6 Class diagram (simplified) for the ATM system

6.2 Seeding of Faults

Faults were seeded according to a set of mutation operators proposed both for procedural [King 1991] and object-oriented programs [Kim 1999, Kim 2000a] (previously used in an experiment [Kim 2000b]). Thus, some of these mutation operators are general and can apply to all imperative programming languages (e.g., the substitution of two operators) whereas others are specific to object-oriented languages (e.g., the removal of an overriding declaration). While seeding our program with faults, we tried to cover all mutation operators in a balanced way given the constraints of the code. As a result we seeded 112 faults (see Appendix F), covering 17 different mutation operators (see Figure 7). The criterion, which we used to evaluate whether the quantity of the

mutant is enough, is the ratio of lines of source code per mutant. In our experiment we seeded one mutant every eight lines. That number was deemed sufficient considering the size of the classes under test. It is also important to note that, to avoid any positive bias in the results, faults were seeded randomly *before* the test cases were designed. The rationale is that when designing test cases it is very difficult to recall where faults were seeded whereas it is easier to “guess” whether a fault is likely to be found by a test strategy that was previously applied to a system and implemented in a test driver. Ideally, these two activities (seeding of faults and test case generation) should be performed by different persons, or faults should be automatically seeded according to the characteristics of the source code and a set of mutation operators [Chevalley 2001].

Moreover, in order to obtain unbiased results, only faults that cannot be detected by the compiler and that can possibly be found by running test cases were considered. For example, if we take the example of the Access Modifier Changes (AMC) mutation operator [Kim 2000a], where an access mode is changed with another one (e.g., from private to public), it is difficult to imagine how a test case can uncover such a fault. We therefore did not use this particular mutation operator. Such faults should be addressed by systematic design and code reviews.

Figure 7 provides the distribution of seeded faults per mutation operator. The mutation operators are denoted by acronyms and have self-explanatory names (some examples are provided below): Argument Number Decrease (AND), Argument Order Change (AOC), Arithmetic Operator Replacement (AOR), Control Flow Disruption (CFD), Comparable Array Name Replacement (CNR), Constant Replacement (CRP), Do Statement End Replacement (DER), Hiding Field Variable Addition (HFA), Class

Instance Creation Expression changes (ICE), Logical Connector Replacement (LCR), Method name Replacement Operator (MRO), method Parameter Order Change (POC), Relational Operator Replacement (ROR), Return Statement Replacement (RSR), Scalar Variable Replacement (SCR), Statement Deletion (SDL), Statement Swap Operator (SSO).

Operator	Correct Statement	Mutant Statement
SSO	return Status.SUCCESS ; else return Status.TOO_LITTLE_CASH ;	Return Status.TOO_LITTLE_CASH ; else return Status.SUCCESS ;
ICE	case 4: return new InquiryTransaction (session, atm, bank);	case 4: return new DepositTransaction (session, atm, bank);
POC	static public Money add(Money first, Money second)	static public Money add(Money second, Money first)
AOC	number = bank.balance(amount , newBalance)	number = bank.balance(newBalance , amount)

Table 5 Mutant operator examples

In Table 5 we give some examples about the mutant operators seeded into the ATM system.

- SSO: the statements embedded in the IF-THEN-ELSE control flow were swapped.
- ICE: a new instance of Transaction Class is created.
- POC: the input parameters, first and second, were swapped in the definition of the method add ()
- AOC: the arguments of method balance() were swapped.

Another example, which is a little bit more complex, is HFA. In the child class WithdrawlTransaction class, we redefine the attribute, Money _newBalance, which is already defined in the parent class, Transaction class.

```

class WithdrawlTransaction extends Transaction {
..
private int    _fromAccount;
private Money  _amount;
private Money  _newBalance;//HFA mutant asserted
}

```

Most of these operators are not specific to the object-oriented paradigm (e.g., DER). The differences regarding the proportion of faults seeded per category is due to the characteristics of the system code under study. For instance, the larger the number of overriding methods, the larger the number of potential AND faults. We only have two AND faults in our case study since the ATM system includes only two instances of method overriding in class Money.

Figure 7 Number of faults seeded per mutation operator

Figure 8 provides the distribution of seeded faults per class. Due to the characteristics of the classes under study, only 11 classes out of 21 were seeded with faults. The class diagram shown in Figure 6 comes from Analysis (where we defined our contracts), and some of the classes were ultimately not implemented as Java classes in the source code, i.e., classes ATMMachineProfile, Card, and Account are implemented with arrays. Classes that represent the parts of an ATM machine (e.g., Keyboard) only

represent the interface with hardware devices, and thus only contain simple methods as the device driver code is missing here. But classes that participate in the core logic of the system (e.g., ATM, Bank, ...) show larger numbers of seeded faults.

Figure 8 Number of faults seeded per class

6.3 Test Cases Design

Totally there are 51 test cases designed for the ATM system according to Category-Partition strategy, which is a famous black-box testing technique. We apply this technique at the system level, using the eight Use Cases of ATM System. Because we apply the testing strategy at the system testing level, the oracles only use the inputs and the outputs of the system, and don't have any other observability into the system.

Here we use Withdrawal Transaction Use Case as an example to explain how we adopt this testing strategy. First of all we define the parameters of this Use Case and list the categories related to each parameter, as in Table 6. For each category the possible choices are identified, then the test cases are defined based on all the possible combinations among those categories. For the detailed test cases information please refers to Appendix E.

Parameter	Category	Choice
Withdraw Amount	Amount more than available cash amount in ATM machine	300
	Amount more than account available amount	200,
	Amount within all the limitation	20, 100
	Accumulated Withdrawal Amount more than Daily Limitation	400
Account	Account available	Checking, Money Market
	Account not available	Saving

Table 6 Category Designed for ATM System Withdrawal Transaction

6.4 Experimental Procedure

The procedure we used in our case study can be described as follows:

- We devise contracts in OCL (at three levels of precision), following clearly defined rules (see Chapter 4). These contracts are then implemented using Jcontract (see Section 4.4), so that three instrumented versions of the system can be generated.
- We seed faults according to clearly defined mutation operators in both instrumented and non-instrumented programs (see Section 6.2). We thus generate mutant programs for four program versions.
- We run the non-instrumented versions of mutant programs and compute the fault detection effectiveness and diagnosability. These program versions only use test oracles (that check program outputs) to detect faults.
- We run the instrumented mutant programs (for all three levels of contract precision) and compute again the fault detection effectiveness and

diagnosability, based on both instrumented contracts and oracles (the same drivers are used for instrumented and non-instrumented programs).

- We compare the fault detection effectiveness and diagnosability of instrumented and non-instrumented mutant program versions, for all three levels of contract precision.

Chapter 7 Analysis Results

In this section we provide a complete reporting of results. We first focus on the detection effectiveness of contracts with respect to faulty system states and then investigate how helpful they can be to support the diagnosis of failures.

7.1 The Detection of Failures

Recall that we are comparing the number of mutants killed among three versions of the ATM system including contract assertions and another version having only regular, hard-coded test oracles checking program outputs. We are looking at both the percentage of non-equivalent mutants killed and the subset of mutants killed by the contracts as opposed to the test oracles.

Contracts	Equivalent Mutants	Live Mutants	Mutants Killed by Oracles	Mutants Killed by Contracts
Without	7	6	99	-
Highest Precision	7	6	20	79
Intermediate	7	6	21	78
Lowest	7	6	24	75

Table 7 Mutant Detection Results for the ATM system

We see from Table 7 that the same number of mutants that are also the same ones remains alive with or without the use of contracts, and regardless of the level of precision of the contracts. It is therefore clear that contracts do not help improve failure detection effectiveness over test oracles. The 6 live mutants represent less than 10% of non-equivalent mutants and are due to imperfect black-box testing. But do contracts make failure detection more cost-effective? A first answer to that question is that, at the highest

level of precision, 79 mutants are killed by contracts, a number that represents 80% of all detected mutants. Using contracts at lower levels of decision results in some mutants not being detected but these differences are not practically significant (in the worst case, 4 additional mutants remain undetected). Note that this result was obtained from contracts defined during the analysis and following precise guidelines (Section 4.1). It is interesting to note that it is similar to the results reported in [Baudry 2001] (see the end of Section 3.2) where a mutant detection effectiveness of 87.5%. The specifics of the contracts with which this result was obtained are not known but we know their contract assertions were based in part on implementation information and this may explain the slightly higher percentage they obtained.

We also investigated the reason why some mutants could not be detected by contracts but were detected by test oracles. We identified five distinct mechanisms (see the Appendix A for complete examples and the Appendix D for other mutants) for which we report the number of instances we encountered:

- The order of parameter changed but the precondition on those parameters is the same and no computation is performed with them (1 instance).
- The contract was not precise enough to catch the failure (10 instances). For instance, postconditions often verify whether values are within ranges. If the effect of the mutant still results in values that are within-range, then the postcondition check does not report a problem.
- The postcondition is checked once the execution of the method is completed. However, if the method itself raises an exception as the result of executing faults, and there is no catch block for that exception in the method, then the

execution does not get the opportunity to check the postcondition (5 instances). A question is whether these cases should be included in our failure detection effectiveness percentage for instrumented mutant programs as assertions may not get the opportunity to catch the failures due to certain faults.

- Some mutants are related to constants being assigned incorrect values. However, when these constants get involved in the definition of contracts, the contract implementations are also faulty and cannot be violated in the mutant programs (3 instances).
- The mutant changes a message sent to one of the actors (i.e., the GUI if we refer to a human actor). We consider it more practical that messages sent to actors be the responsibility of an oracle that, for example, check output log files. Therefore, when output messages to actors were specified in the postconditions, we chose not to instrument this aspect of the contract. This category should not be considered when looking at the effectiveness of contracts to kill mutants (1 instance).

7.2 The Diagnosability of Faults

We now investigate whether using contracts can help reduce debugging effort by facilitating the isolation of faults in object-oriented systems. We only consider here the faults that were detected by contracts as for the others it is obvious that no diagnosability improvement can be measured. We thus consider 79/78/75 mutants out of the 99 mutants killed by oracles, for the highest, intermediate, and lowest contract precision levels, respectively.

Average diagnosability results are presented in Figure 9 and the diagnosability value for each mutant can be found in Appendix G. Without resorting to statistical inference testing, it is clear that the difference in diagnosability between the ATM system with contracts (at any level of precision) and the one without contracts is very significant as it approaches one order of magnitude. This should translate, in practice, into significant effort savings during debugging. However, in order to assess such savings precisely, an experiment would need to be run in a realistic context with actual software developers. We can also see that simple contracts, at the lower level of precision, are sufficient to improve the average diagnosability, though on a slightly lower number of mutants.

Assertions	Number of Mutants	Average Diagnosability
With Oracie only	79 mutants	13.30
Highest Precision	79 mutants	1.35
Intermediate	78 mutants	1.36
Lowest	75 mutants	1.37

Figure 9 Average Diagnosability

But averages do not tell the whole story. Five cases of poor diagnosability (see Appendix G) can be observed when using contracts whereas a number of minimum diagnosability values (1) can be observed without contracts. Figure 10 shows diagnosability values (for both the contract and oracle cases) on the Y-axis and mutants are ordered (X-axis) according to their diagnosability when using hard-coded oracles. This allows us to compare results graphically and identify cases where diagnosability did not improve as much as expected. The poor diagnosability cases when using contracts

correspond to five different mutation operators (POC, CRP, HFA, CFD, SCR). Note that two of them (POC and CRP) correspond to the two cases discussed above (mutants not killed by contracts) where mutants involve a compatible parameter order change and a constant value change. In these five cases, as a side effect and probably by chance, contracts get broken in other methods that are far away, in terms of diagnosis flow, from the methods where faults were seeded.

Figure 10 Diagnosability Distribution

In our experiment, faults are mostly detected in the classes where they lie or are not detected at all (in their clients), except in five instances. For the highest precision contracts, the diagnosability is on average 1.35, is equal to 1 for 74 mutants (the fault is detected in the class that contains it) and lies between 2 and 10 for five mutants. Results are nearly identical for contracts of lowest and intermediate precision. This contradicts, to

some extent, the results by [Baudry 2001] who estimated coefficient c to average at 0.8¹² (Section 3.1). Our results suggest that, in our system, c is small as failures are detected by client classes' contracts only in a couple of cases. This issue clearly needs to be further investigated.

¹² The procedure followed to obtain that estimation is not provided in [Baudry 2001].

Chapter 8 Validity

Based on our sensitivity analysis, we have shown that no theoretical model can, with sufficient precision, tell us whether instrumenting analysis contracts is likely to be a good solution for automating test oracles. We therefore resorted to carefully designing and performing a case study.

Any case study, despite the substantial effort it involves, only represents one data point. Such a problem is not specific to software engineering case studies but is encountered in many other fields such as social sciences. So the usual question is: to which extent are the results obtained externally valid, that is generalizable? Though this is inherently a difficult question, one way to alleviate the problem is to look beyond the numbers to understand the mechanisms by which we obtain the quantitative results we report. In our study, we investigated every mutant that remained undetected by contracts or oracles. We also looked at every mutant that showed a poor diagnosability. We identified, across mutants, the common mechanisms which would explain poor observability (Section 7.1) or diagnosability (Section 7.2), and assessed whether such mechanisms should be commonly encountered in other programs as well. This gave us a good insight into whether our results were likely to be encountered in many other situations, e.g., different software systems.

Following traditional testing research, we identified a general fault taxonomy (mutation operators) for Java programs and made sure we randomly seeded a wide variety of faults (Section 6.2) in the program to be tested. We used a well-known traditional black-box software technique (Section 6.3) to generate test cases. We also made sure the program selected (Section 6.1) for experimentation had no specific

peculiarity that could hinder us from getting meaningful results. We also ensured that contracts were defined using well-defined rules, at a realistic level of detail, so that the results could be interpreted in a well-defined context.

Chapter 9 Automation Tool

A tool named Mutant Insertion Automation and Testing tool (MIAT) was developed to automate the mutant insertion procedure and the execution of test cases on these mutants. This tool was also designed so that it can be easily used during future experiments.

9.1 Requirements

We could classify the system requirements into two major functions, the mutant insertion and the testing on the mutant programs.

According to the mutant information provided by the user, MIAT should be capable to create the mutant programs into a user-defined directory, compile each of them and acquire the result. Any failure during the process should be prompted to the user via either a Graphic User Interface (GUI) or a log file. The user can stop the process at any time if desired, and MIAT should not be fallen into a deadlock when the termination command is executed.

When all the mutant programs are created, MIAT is able to test them in a sequence either with the oracles or with the contracts. Before the execution of the test cases MIAT needs to compile the program either with the normal Java compilation command (testing with the oracle) or the JContract (section 4.4.1) compilation command (testing with the contracts) in accordance with the user's option. The testing could start only with a successful compilation, and all the testing results should be reported to the user via a GUI and written into a test log file for further investigation. The user should also be able to terminate the testing process at any time successfully.

The last requirement, which is also deemed very important, is that a detailed user manual should be available within the tool in order to give other users a guide.

9.2 Design

From the system requirements MIAT is composed of seven classes (Figure 11) and the responsibilities of each class are defined as follows.

Figure 11 Class Diagram of MIAT¹³

- **MIAT:** This class contains the main class and is responsible for initiating the GUI of MIAT.
- **MIATGUI:** This class generates the GUI of MIAT.

¹³ In this class diagram the attributes are hidden to highlight the relationship of the classes.

- **MTEvent:** This class implements the `ActionListener` Interface and offers two operations, `chooseFile()` and `reset()`, which are used to select the file and reset the parameters' fields on the GUI.
- **InsertionThread:** This thread is responsible for finishing the mutant insertion task and consists of three steps: making a copy of the correct program into the defined directory in which all the mutant programs are stored (Figure 12), seeding the mutant according to the mutant file and compiling the mutant program.
- **TestSession:** This class accomplishes the testing task, which involves testing all the mutant programs with the test driver, evaluating whether the mutant is killed or not, and recording the test result.
- **TestDriver:** In the `TestDriver` class the mutant program is compiled and all the test cases are executed one by one.
- **UserGuide:** This class supplies the user manual in an independent frame.

Figure 12 Directory Structure of Mutant programs

9.3 Implementation

Java, a platform-independent language, is chosen to implement MIAT, which comprises 1,200 LOCs (without comments) and seven classes. The GUI of MIAT is created with the Swing components, which are part of the Java™ Foundation Classes (JFC) and can be used with either JDK™ 1.1 or the Java™ 2 platform.

9.4 Usage

The user can browse the MIAT User Guide from the **Help** menu to get detailed information about how to use this tool. The following is a brief introduction.

Mutant Insertion Automation

By clicking on the **Mutation Insertion** tab (Figure 13), the mutant insertion automation module is displayed.

Figure 13 GUI of Mutant Insertion

The user should input three parameters before starting the mutant insertion process. The first one, **Project Path**, which is the directory containing the correct version of the tested program, can be inputted either from the keyboard manually or by clicking on the **OpenPP** button to browse the hard disk directory. Then the user needs to identify the path of the target directory (**Target Path**), in which the mutant programs will be created, by

clicking on the **OpenTP** button. The **Mutant File**, which contains the detailed information of each mutant designed by the user, could be selected from the **OpenMF** button (for the format and an example, see Appendix F). If the user wants to change the parameters, he/she can use the **Reset** button to clear the contents of all three fields. If not the process could be initiated by clicking on the **Start** button. During the course of the mutant insertion, the user can terminate the process at any time by using the **Stop** button, and the process will be stopped when the current mutant program's creation is finished.

While the mutant programs are being created, the user can observe the mutant information through the **Mutant considered** text field and the process results from the **Message Box** text area.

Test Driver

After clicking on the **Testing** tab (Figure 14), the user is ready to test all the mutant programs with or without the contracts via the test driver.

Figure 14 GUI of Testing Driver

Following the same GUI style in the mutant insertion process, the user is also required to feed MIAT with three parameters before starting the test. First of all the user can click on the **OpenTP** button and find the path of the directory including all the mutant

programs (**Tested Project**). Then the path of the test case (**Test Case Path**) should be inputted by clicking on the **OpenTC** button. In this case study we test the ATM system at the system level; therefore, the test cases are a set of the input text files, which simulate the input from the user. **Main Class** will be used by MIAT to initiate the tested program, and if the user selects the **With Contract** box, MIAT will test the mutant programs with the contracts. Otherwise, MIAT will test the mutant programs with the oracle. Now the user can start the testing process with the **Start** button, and terminate the process with the **Stop** button. When the testing is finished, the user can read the test report with the **Report** button.

In the process of the testing the user can find out the status information from other text fields. The **Test Case Executing** field tells the user which test case is executed at that time. The **Testing Message Box** displays the execution result of each test case on the mutant program that is failed by the contract, failed by the oracle or passed. The other three text fields, **Successful**, **Failed by the Contracts** and **Failed by the Oracle**, note the number of the mutant programs being alive, killed by the contracts, and killed by the oracles, respectively.

9.5 Summary

With the aid of MIAT we can conduct the ATM system testing not only correctly but also efficiently. The mutant insertion part of MIAT could also be reused in other research areas that involve mutation testing. However, this tool could still be improved in some places. For instance, the interface between the testing session and the test driver could be defined more clearly, so that the test driver could be changed to execute the test cases with a different testing strategy. Another improvement is considering the

performance of the mutant insertion. Currently MIAT needs to generate the mutant program by copying all the source code even if only one statement is changed in one source code file, which affects the generation speed significantly when there are thousands of lines of codes. The structure of the mutant programs (Figure 12) could be modified to address this problem in the future.

Chapter 10 Conclusions

From the results of our case study, it seems clear that contracts detect a large percentage of the failures, ranging from 76% to 80%, depending on the level of precision of the contract definitions. In the more precise case, if we include the 5 instances where run-time exceptions are raised in the program¹⁴, the percentage of failures detected rises to 85%. In other words, in roughly 80 percent of the cases, contracts are good enough substitutes to hard-coded oracles in test drivers. Though defining and instrumenting very precise contracts helps with the detection of failures, the difference does not seem practically significant and probably does not justify the cost entailed with precise contracts.

One issue is that to find 20% of the faults we still require test oracles to detect the corresponding failures and, more importantly, we would need to know beforehand which test cases require hard-coded test oracles. If this were the case, we could save a large percentage of the effort of defining and coding them. Future research has to provide a way to identify, during the coding of the test driver, the test cases that will need the hard-coded test oracles. One avenue to be explored is to identify weak contracts through fault seeding (mutation) and require oracles for those test cases that exercise weak contracts. Another strategy worth investigating is to compensate for the small percentage of failures that remain undetected when not using hard-coded oracles with a larger number of test cases. This may still be economically viable as the corresponding test drivers are less expensive to develop when they do not contain oracles.

¹⁴ They cannot be detected by instrumented contracts, but provide information regarding where the exception was raised

In terms of instrumentation, our experience is that with a tool like Jcontract, precise implementation rules can be devised to facilitate the transition from Analysis contracts (in OCL) to Javadoc contracts (in DbC). The complexity of the instrumentation depends, of course, on the complexity of the contracts, e.g., whether they manipulate sets and therefore require the traversing of complex data structures. As a consequence, an important future research direction is to investigate the impact of using approximate, simpler contracts instead of the precise contracts instrumented here. This is especially relevant in light of the sensitivity analysis results, suggesting that contracts need not be perfect to be effective.

Another important result that shows to be very strong and of high practical significance is that diagnosability, as we measure it, improves nearly an order of magnitude between mutant programs without contracts and the ones with contracts, regardless of the level of precision. This suggests significant savings during debugging, as faults will be much easier to locate if contracts are being instrumented and checked during execution.

In conclusion, our overall results suggest that instrumented contracts have a strong potential in terms of decreasing the cost of testing. Furthermore, they do not require to be defined and instrumented at a high level of precision to be useful. Those results are the outcome of a carefully planned and designed study and can be replicated on other system examples, using well-defined procedures and a commercial tool. Replication is essential in order to confirm the results we obtained here. During these future experiments, the tool developed (MIAT) will be of great help.

References

- [Baudry 2000]
B. Baudry, V. Le Hanh, J. M. Jezequel and Y. Le Traon, "Building Trust into OO Components using a Genetic Analogy," *Proc. International Symposium on Software Reliability Engineering (ISSRE)*, San Jose, California, pp. 4-14, October 8-11, 2000.
- [Baudry 2001]
B. Baudry, Y. Le Traon and J. M. Jezequel, "Robustness and Diagnosability of OO Systems Designed by Contracts," *Proc. 7th International Software Metrics Symposium*, London, England, pp. 272-283, 4-6 April, 2001.
- [Beizer 1990]
B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, 2nd Ed., 1990.
- [Binder 1999]
R. V. Binder, *Testing Object-Oriented Systems - Models, Patterns, and Tools*, Addison-Wesley, 1999.
- [Chevalley 2001]
P. Chevalley and P. Thévenod-Fosse, "An Empirical Evaluation of Statistical Testing Designed from UML State Diagrams: the Flight Guidance System Case Study," *Proc. 12th IEEE International Symposium on Software Reliability Engineering*, Hong-Kong (China), pp. 254-263, 27-30 November, 2001.
- [Coleman 1994]
D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. GilChrist, F. Hayes and P. Jeremaes, *Object-Oriented Development - The Fusion Method*, Prentice Hall Ed., 1994.
- [Cousot 2000]
P. Cousot and R. Cousot, "Abstract Interpretation Based Program Testing," *Proc. SSGRR 2000 Computer & eBusiness International Conference*, Rome (Italy), July, 2000.
- [iContract]
Reto Kramer, *iContract - The Java Design by Contract Tool*, Cambridge Technology Partners
<http://www.reliable-systems.com/tools/iContract/documentation/icontract-tools98usa.pdf>
- [Jass]
"The Jass Page", University of Oldenburg, Germany, <http://semantik.informatik.uni-oldenburg.de/~jass/>
- [JContract]
JContract, "Using Design by Contract to Automate Java Software and Component Testing," ParaSoft Corporation,
http://www.parasoft.com/products/jtest/papers/tech_dbc.htm.
- [Jezequel 2001]
J. M. Jezequel, D. Deveaux and Y. Le Traon, "Reliable Object: Lightweight Testing for OO Languages," *IEEE Software*, vol. 18 (4), pp. 76-83, 2001.

- [JMSAssert]
 "Design by Contract for Java Using JMSAssert", Man Machine Systems
<http://www.mmsindia.com/DBCForJava.html>
- [Kim 1999]
 S. Kim, J. A. Clark and J. A. McDermid, "The Rigorous Generation of Java Mutation Using HAZOP," *Proc. 12th International Conference on Software & Systems Engineering and their Applications (ICSSEA'99)*, CNAM - Paris, France, pp. 9-10(11), December 8-10, 1999.
- [Kim 2000a]
 S. Kim, J. A. Clark and J. A. McDermid, "Class Mutation: Mutation Testing for Object-Oriented Programs," *Proc. Net.ObjectDays*, Erfurt, Germany, October 9-12, 2000a.
- [Kim 2000b]
 S. Kim, J. A. Clark and J. A. McDermid, "Investigating the effectiveness of Object-Oriented Testing Strategies with the Mutation Method," *Proc. Mutation 2000*, San Jose, October 6-7, 2000b.
- [King 1991]
 K. N. King and A. J. Offutt, "A Fortran Language System for Mutation-Based Software Testing," *Software - Practice and Experience*, vol. 21 (7), pp. 686-718, 1991.
- [Meyer 1992]
 B. Meyer, "Design by Contracts," *IEEE Computer*, vol. 25 (10), pp. 40-52, 1992.
- [OMG 2001]
 OMG, "Object Constraint Language - OCL," Object Management Group, V1.4, 2001.
- [Ostrand 1988]
 T. J. Ostrand and M. J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Test," *Communications of the ACM*, vol. 31 (6), pp. 676-686, 1988.
- [Rosenblum 1995]
 D. Rosenblum, "A Practical Approach to Programming with Assertions," *IEEE Trans. on Software Engineering*, vol. 21 (11), pp. 19-31, 1995.
- [Voas 1995a]
 J. M. Voas, "Software Testability Measurement for Assertion Placement and Fault Localization," *Proc. 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, St Malo (France), May, 1995a.
- [Voas 1995b]
 J. M. Voas, "Testability of Object-Oriented Systems," NIST, Technical Report GCR-95-675, 1995b.
- [Voas 1991]
 J. M. Voas, L. Morell and K. Miller, "Predicting Where Faults Can Hide from Testing," *IEEE Software*, vol. 8 (2), pp. 41-48, 1991.
- [Wang 1997]
 Y. Wang, G. King, I. Court, M. Ross and G. Staples, "On Testable Object-Oriented Programming," *ACM Software Engineering Notes*, vol. 22 (4), pp. 84-90, 1997.
- [Warner 1999]
 J. Warner and A. Kleppe, *The Object Constraint Language*, Addison-Wesley, 1999.

Appendix A: Examples of mutants not killed by contracts

In this Appendix, we illustrate the five distinct mechanisms that can explain why mutant programs were not killed in our ATM case study (see Section 7.1). We provide a detailed example for each of the five cases.

Situation 1: The order of parameter changed but the precondition on those parameters is the same and no computation is performed with them.

Mutant 1 (mutation operator AOC) consists in swapping two parameters (`cardNumber` and `serialNumber`) in a call to method `printReceipt()` (class `ReceiptPrinter`). Method `printReceipt()` only verifies the range of values of these two parameters and issues a receipt of the transaction (no computation). Since the two parameters have the same range, the mutant program is not killed.

```
// In class ATM
/** @pre serialNumber>0
 * @pre balance.getValue() >= availableBalance.getValue()
 */
public void issueReceipt(int cardNumber,int serialNumber,String
description,Money amount,Money balance,Money availableBalance) {

    String [] receipt = new String[7];
    _receiptPrinter.printReceipt(_number, _location, serialNumber,
cardNumber, description, amount, balance, availableBalance,receipt );
    // this is the mutant: the two parameters are swapped
    for (int i = 0; i < 7; i ++){
        _display.receiptPrint(receipt[i]);
    }
}

// In class receiptPrinter
/** @pre cardNumber >=0
 * @pre serialNumber >0
 * --
 */
public void printReceipt(...) {
    // only outputs from the parameters
}
```

Figure 15 Mutant program not killed: example of Situation 1

Situation 2: The contract was not precise enough to catch the failure.

Mutant 57 (mutation operator MRO) consists in replacing the call to `getAmountEntry()` (method in class `ATM`) with a call to method `startupOperation()` (method in class `ATM`). Statement `_amount = _atm.getAmountEntry()` is changed into `_amount = _atm.startupOperation()`. This change is done in method `getTransactionSpecificFromCustomer()` in class `TransferTransaction`, which asks the customer to choose two accounts (for the transfer) and an amount. The postcondition does not (and cannot) involve the amount since it is entered by the user who can provide any number: whether the amount is available is checked elsewhere, and whether the amount is within acceptable range (i.e., the amount is a positive integer) is the responsibility of class `Money`'s invariant. Since `_atm.getAmountEntry()` and `_atm.startupOperation()` return an amount which can be within expected range (positive integer), the mutant is not killed.

```

/** @post $result == Status.SUCCESS
 */
public int getTransactionSpecificsFromCustomer() {
    _fromAccount = _bank.chooseAccountType("transfer from", _atm);
    _toAccount = _bank.chooseAccountType("transfer to", _atm);
    _amount = _atm.startupOperation(); // mutant
    return Status.SUCCESS;
}

```

Figure 16 Mutant program not killed: example of Situation 2

Situation 3: An exception is raised before the evaluation of the contract assertion.

Mutant 59 (mutation operator CRP) changes a case label in class `WithdrawalTransaction` in method `getTransactionSpecificFromCustomer()`. The case label "case 5: ..." is changed to "case 8: ..." (the statements executed are not changed). The corresponding

switch creates an instance of class Money according to the amount entered by the customer. Then, the amount is checked against the available amount in the selected account. In the mutant, the Money instance corresponding to amount 5 is not created, and since there is not default label in the switch, the reference used in the check is null, thus raising an exception. The postcondition does not have the opportunity to execute.

```

/** @post $result == Status.TOO_LITTLE_CASH || $result ==
 *                                     Status.SUCCESS
 */
public int getTransactionSpecificsFromCustomer() {
    _fromAccount = _bank.chooseAccountType("withdraw from", _atm);
    String menu[] = { "$ 20", "$ 40", "$ 60", "$ 80", "$ 100", "$ 200", "$
    300" };
    int option = (_atm.getMenuChoice("Please choose an amount:", 7, menu));
    switch (option) {
        case 1: _amount = new Money(20); break;
        case 2: _amount = new Money(40); break;
        case 3: _amount = new Money(60); break;
        case 4: _amount = new Money(80); break;
        case 8: _amount = new Money(100); break; // mutant
        case 6: _amount = new Money(200); break;
        case 7: _amount = new Money(300); break;
    }
    /** @assert (option ==1)&&(_amount.getValue()==2000.00 )||
     *          (option == 2)&&( _amount.getValue()==4000.00 )||
     *          (option == 3)&&(_amount.getValue() == 6000.00 )||
     *          (option == 4)&&(_amount.getValue() == 8000.00)||
     *          (option == 5)&&(_amount.getValue() == 10000.00)||
     *          (option == 6)&&(_amount.getValue() == 20000.00)||
     *          (option == 7)&&(_amount.getValue() == 30000.00) */
    if (_atm.checkIfCashAvailable(_amount))// an exception is raised here
        return Status.SUCCESS;
    else
        return Status.TOO_LITTLE_CASH;
}

```

Figure 17 Mutant program not killed: example of Situation 3

Situation 4: Mutants are related to constants being assigned incorrect values.

Mutant 45 (mutation operator CRP) changes the value of constant variable RUNNING in class Session: The original value is 0 and the new value is 1. However, Session's invariant, the pre- and postconditions of Session's methods, as well as the source code,

use the name of the constant (RUNNING) instead of the value. Then, the mutant is not killed.

```

/** @post _state==RUNNING
 * @post _PIN == 0
 * @post _atm == atm
 * @post _bank == bank
 * @post _cardNumber == cardNumber
 * @post _currentTransaction == null
 */
public Session(int cardNumber, ATM atm, Bank bank) {
    _cardNumber = cardNumber;
    _atm = atm;
    _bank = bank;
    _state = RUNNING;
    _PIN = 0;
    _currentTransaction = null;
}

```

Figure 18 Mutant program not killed: example of Situation 4

Situation 5: The mutant is related to messages sent to actors

Mutant 49 (mutation operator RSR), in method `Session::doFailedTransactionExtension()`, changes a string. This change eventually has an impact on what is displayed on the screen. As explained above, we considered that verifying messages sent to actors can be better performed with an oracle (that check output log files) than with instrumented contracts.

```

context: Session::doFailedTransactionExtension(reason:int):Boolean
pre:   reason = Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED
        or
        ..
        or
        reason = Status.TOO_LITTLE_CASH
post: ATM.explanation._messages -> includes(result)

```

Figure 19 Contracts for method `Session::doFailedTransactionExtension()`

Mutant 49 changes string "Envelope not deposited - transaction cancelled" with string "Transaction OK".

```

/**
 * @pre   reason == Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED || ... ||
         reason == Status.TOO_LITTLE_CASH
 */
public boolean doFailedTransactionExtension(int reason) {
    switch(reason) {
        case Status.TOO_LITTLE_CASH:
            return _atm.reportTransactionFailure("Sorry, there is not
                enough cash available to satisfy your request");

        case Status.ENVELOPE_DEPOSIT_TIMED_OUT:
            return _atm.reportTransactionFailure("Envelope not deposited -
                transaction cancelled");

        default:
            return
                _atm.reportTransactionFailure(_bank.rejectionExplanation(
                    reason));
    }
}

```

Figure 20 Mutant program not killed: example of Situation 5

Appendix B: contracts at different levels of precision

We provide below three post-conditions for method `Bank: initiateWithdrawl()`, one at each level of precision, in decreasing order of precision and according to the rules provided in Section 4.2.1.

```
post: if not self.card->exists(_cardNumber = cardNumber) then
    result = Status.UNKNOWN_CARD
else
    if PIN <> self._currentTransactionCard._PIN then
        result = Status.INVALID_PIN
    else
        if not self._currentTransactionAccount._available then
            result = Status.NO_SUCH_ACCOUNT
        else
            if from = SAVINGS then
                result = Status.CANT_WITHDRAW_FROM_ACCOUNT
            else
                if self._currentTransactionAccount._availableBalance.
                    less(amount) then
                    result = Status.INSUFFICIENT_AVAILABLE_BALANCE
                else
                    if (MAXIMUM_WITHDRAWAL_AMOUNT_PER_DAY.less(Money.add(
                        self._currentTransactionCard.withdrawalsToday, amount))
                        then
                        result = Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED
                    else
                        result = Status.SUCCESS
                        and
                        _currentTransactionAmount.equals(amount)
                        and
                        newBalance.getValue() =
                            self._currentTransactionAccount._currentBalance.
                                getValue() - amount.getValue()
                        and
                        availableBalance.getValue() =
                            self._currentTransactionAccount._availableBalance
                                .getValue() - amount.getValue()
                    endif
                endif
            endif
        endif
    endif
endif
```

Figure 21 Postcondition for `Bank: initiateWithdrawl()` at the highest precision level

```

post: if ( (self.card->exists(_cardNumber = cardNumber))
           and
           (PIN == self._currentTransactionCard._PIN)
           and
           (self._currentTransactionAccount._available)
           and
           (from <> SAVINGS)
           and
           (not
            self._currentTransactionAccount._availableBalance.less(amount))
           and
           (not MAXIMUM_WITHDRAWAL_AMOUNT_PER_DAY.less( Money.add(
            self._currentTransactionCard_withdrawalsToday, amount)))
        ) then
    result = Status.SUCCESS
    and
    _currentTransactionAmount.equals(amount)
    and
    newBalance.getValue() =
        self._currentTransactionAccount._currentBalance.getValue() -
        amount.getValue()
    and
    availableBalance.getValue() =
        self._currentTransactionAccount._availableBalance
        .getValue() - amount.getValue()
else
    result = Status.UNKNOWN_CARD
    or
    result = Status.INVALID_PIN
    or
    result = Status.NO_SUCH_ACCOUNT
    or
    result = Status.CANT_WITHDRAW_FROM_ACCOUNT
    or
    result = Status.INSUFFICIENT_AVAILABLE_BALANCE
    or
    result = Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED
endif

```

Figure 22 Postcondition for **Bank: initiateWithdrawal ()** at the intermediate precision level


```

post: result = Status.UNKNOWN_CARD
      or
      result = Status.INVALID_PIN
      or
      result = Status.NO_SUCH_ACCOUNT
      or
      result = Status.CANT_WITHDRAW_FROM_ACCOUNT
      or
      result = Status.INSUFFICIENT_AVAILABLE_BALANCE
      or
      result = Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED
      or
      ( result = Status.SUCCESS
        and
        _currentTransactionAmount.equals(amount)
        and
        newBalance.getValue() =
          self._currentTransactionAccount._currentBalance.getValue()
          - amount.getValue()
        and
        availableBalance.getValue() =
          self._currentTransactionAccount._availableBalance.
          getValue() - amount.getValue()
      )

```

Figure 23 Postcondition for **Bank: initiateWithdrawal ()** at the lowest precision level

Appendix C: Contract for ATM System

In this appendix we list the contracts designed in analysis phase (with OCL) and implementation phase (with JContract syntax).

Class ATM

Class Attributes:

```
RUNNING: static final int // the value is set to 0
STOPPED: static final int // the value is set to 1
```

Instance Attributes:

```
_number: int // this ATM Machine number
_location: String // this ATM Machine location
_bank: Bank
_state: int // whether the ATM machine is running or not
_cardReader: CardReader
_display: Display
_keyboard: Keyboard
_cashDispenser: CashDispenser
_envelopeAcceptor: EnvelopAcceptor
_receiptPrinter: ReceiptPrinter
_operatorPanel: OperatorPanel
_cardNumber: int
```

Invariant

Contract in Analysis:

```
context ATM inv:
    self.atmMachineProfile._ATMnumber-> includes (self._ATMnumber)
    and
    self.atmMachineProfile._ATMLocation -> includes (self._location)
    and
    (self._state = RUNNING or self._state = STOPPED)
```

Contract in Implementation

```
/**
 * invariant for ATM class
 * @invariant _number == 42
 * @invariant _location.equals("GORDON COLLEGE")
 * @invariant _state == RUNNING || _state == STOPPED
 */
```

ATM::ATM(number:int, location:String, bank:Bank, container:Container):void

This method is the constructor of ATM class.

Contract in Analysis:

```
context ATM::ATM()
    post: self._bank = bank and
          self._CardReader ->notEmpty() and
          self._keyboard -> notEmpty() and
          self._cashDispenser -> notEmpty() and
          self._envelopeAcceptor -> notEmpty() and
          self._receiptPrinter -> notEmpty() and
          self._operatorPanel -> notEmpty() and
```

```

self._number = number and
self._location = location

```

All these objects are created in the creator method.

Contract in Implementation

```

/**@post _number == number
 * @post _location.equals(location)
 * @post _bank == bank
 * @post _cardReader!=null
 * @post _keyboard != null
 * @post _cashDispenser !=null
 * @post _envelopeAcceptor !=null
 * @post _receiptPrinter !=null
 * @post _operatorPanel != null
 */

```

ATM::startupOperation():Money

This method shows the message that tells the user what can be done, and waits for the ATM machine to be turned on.

Contract in Analysis:

```

context ATM:: startupOperation():Money
  post: self._state = RUNNING and
        result.getValue() =
        _operatorPanel.getInitialCash(answer).getValue()

```

Notes: This postcondition is not implemented in the source code because `_operatorPanel.getInitialCash(answer)` is a method needing the input from device and can't be implemented. Therefore the walk around is adopted for this postcondition as:

```

post: self._state = RUNNING and
      result.getValue() >= 2000.00

```

Contract in Implementation

```

/**
 * @post _state == RUNNING
 * @post $result.getValue() >= 2000.00
 */

```

ATM::serviceCustomers(initialCash:Money):void

This method checks the customer card status. If the card is readable a new session will be started, otherwise the "unreadable card" message will be displayed to the customer and the card will be ejected.

Contract in Analysis:

```

context ATM:: serviceCustomers(initialCash:Money):void
    post: self._state = STOPPED

```

Contract in Implementation:

```

/**
 * @post _state == STOPPED
 */

```

ATM::getPIN():int

This method tells the customer to input his/her PIN and gets the PIN number from the customer input.

Contract in Analysis:

```

context ATM:: getPIN():int
    post: result = _keyboard.readNumber()

```

Notes: this post condition is not implemented into the source code because the `_keyboard.readNumber()` need customer to input some number from the keyboard.

ATM::getMenuChoice(whatToChoose: String, numItems: int, items[]: String) : int

This method shows a menu to the customer, and gets the customer's choice.

Contract in Analysis:

```

context ATM:: getMenuChoice( whatToChoose:String, numItems:int,
                             items[]:String):int
    pre: self.display->exists(_MenuItem = numItems)
    post: result >0 and result <= numItems

```

Contract in Implementation:

```

/**
 * @pre numItems < 8 && numItems >0
 * @post $result >0
 * @post $result <= numItems
 */

```

ATM::getAmountEntry():Money

This method gets the amount from the customer.

Contract in Analysis:

```

context ATM:: getAmountEntry():Money
    post: result.getValue() = _keyboard.readAmountEntry().getValue()

```

Notes: this post condition is not implemented into the source code because the `_keyboard.readAmountEntry()` need customer to input some number from the keyboard.

ATM::checkIfCashAvailable(amount:Money):Boolean

This method checks whether the ATM machine has enough cash

Contract in Analysis:

```

context ATM:: checkIfCashAvailable(amount:Money):Boolean
    post: result = not (self.cashDispenser.currentCash().less(amount))

```

Contract in Implementation:

```

/**
 * @post $result == *!_cashDispenser.currentCash().less(amount)
 */

```

ATM::dispenseCash(amount:Money):void

This method is a delegation method and no postcondition designed for it.

ATM::acceptEnvelop():Boolean

This method is a delegation method and no postcondition designed for it.

**ATM::issueReceipt(cardNumber:int, serialNumber:int
description:String, amount:Money, balance:Money,
availableBalance:Money):void**

This method is a delegation method and no postcondition designed for it.

Contract in Analysis:

```

context: ATM:: issueReceipt(cardNumber:int, serialNumber:int
    description:String, amount:Money, balance:Money,
    availableBalance:Money):void
    pre:    serialNumber > 0 and
    balance.getValue() >= availableBalance.getValue()

```

Contract in Implementation:

```

/**
 * @pre serialNumber>0
 * @pre balance.getValue()>=availableBalance.getValue()
 */

```

ATM::reEnterPIN():int

This method tells the customer he/she needs to reenter the PIN number.

Contract in Analysis:

```

context: ATM:: reEnterPIN():int
    post:    result = _keyboard.readNumber()

```

Notes: this post condition is not implemented into the source code because the
_keyboard.readNumber() need customer to input some number from the keyboard.

ATM::reportTransactionFailure(explanation:String):Boolean

This method displays the transaction failure message to customer.

Contract in Analysis:

```

context: ATM:: reportTransactionFailure(explanation:String):Boolean
    pre: self.explanation._message -> includes (explanation)
    post: result = (_keyboard.readMenuChoice(2) =1)

```

Notes: The precondition of this method is not implemented because we can't find the suitable data structure to implement it.

ATM::ejectCard():void

This method ejects the customer's card when the operation is finished or because of other abnormal reasons. This method simply delegates its processing therefore there is no contract designed for it.

ATM::retainCard():void

This method retains the customer's card. It simply delegates its processing therefore there is no contract designed for it.

ATM::number():int

This method returns the number of this ATM machine.

Contract in Analysis:

```
context: ATM::number():int
post: result = self._number
```

Contract in Implementation:

```
/**
 * @post $result == _number
 */
```

Class Bank

Method `verifyAccount(int account)` is added to support the contract checking.

Class Attributes

```
CHECKING : static final int // the value is set to 0
SAVINGS : static final int // the value is set to 1
MONEY_MARKET : static final int // the value is set to 2
NUMBER_CARDS: static final int // the value is set to 2
NUMBER_ACCOUNTS: static final int // the value is set to 3
MAXIMUM_WITHDRAWL_AMOUNT_PER_DAY: static final Money
// the value is set as new
// Money(300);
```

Instance Attributes

```
_currentTransactionCard : Card
_currentTransactionAccount : Account
_currentTransactionAmount : Money
```

Invariant

Contract in Analysis:

```
context: Bank
  inv : self._currentTransactionCard -> notEmpty() and
        self._currentTransactionAccount -> notEmpty() and
        self._currentTransactionAccount._availableBalance.
            getValue() <= self._currentTransactionAccount.
                _currentBalance.getValue()
```

Contract in Implementation:

```
/**
 * Invariant for Bank class
 * @invariant {
 *     for (int i = 0; i < _currentBalance.length; i++)
 *     $assert (_currentBalance[i].getValue() >=
 *         _availableBalance[i].getValue());
 * }
 * @invariant _currentBalance.length == _availableBalance.length
 * @invariant _currentTransactionCard >= 0
 * @invariant _currentTransactionAccount >= 0
 */
```

Bank::bank()

This method is the constructor of Bank class. It does nothing therefore no contract designed.

Bank::initiateWithdrawl(cardNumber:int, PIN:int, serialNumber:int, from:int, amount:Money, newBalance:Money, availableBalance:Money):int

This method processes the information which is transferred from customer withdraw operation. We design the highest precise, lowest precise and intermediate precise contracts for this method.

Highest Precise Contract in Analysis:

```
context: Bank::initiateWithdrawl(cardNumber:int, PIN:int, serialNumber:int,
                                from:int, amount:Money, newBalance:Money,
                                availableBalance:Money):int
  pre:  serialNumber > 0 and
        from = CHECKING or from = SAVINGS or from = MONEY_MARKET

  --the post-condition below almost rewrites the code
  post: if not self.card->exists(_cardNumber = cardNumber) then result =
        Status.UNKNOWN_CARD
        else
        if PIN <> self._currentTransactionCard._PIN then
            result = Status.INVALID_PIN
        else
            if not self._currentTransactionAccount._available then
                result = Status.NO_SUCH_ACCOUNT
```

```

else
  if from = SAVINGS then
    result = Status.CANT_WITHDRAW_FROM_ACCOUNT
  else
    if self._currentTransactionAccount.
      _availableBalance. less(amount) then
      result = Status.INSUFFICIENT_AVAILABLE_BALANCE
    else
      if (MAXIMUM_WITHDRAWAL_AMOUNT_PER_DAY.less(
        Money.add(self._currentTransactionCard_withdrawalsToday,
          amount))) then
        result = Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED
      else
        result = Status.SUCCESS and
        _currentTransactionAmount.equals(amount) and
        newBalance.getValue() =
          self._currentTransactionAccount._currentBalance.ge
            tValue() - amount.getValue() and
        availableBalance.getValue()=
          self._currentTransactionAccount._availableBalance
            .getValue() - amount.getValue()
      endif
    endif
  endif
endif
endif
endif
endif
endif

```

Highest Precise Contract in Implementation:

```

**
* @pre serialNumber > 0
* @pre from == CHECKING || from == SAVINGS || from == MONEY_MARKET
* @post {if(cardNumber<1 ||cardNumber> NUMBER_CARDS)
*       $assert ($result == Status.UNKNOWN_CARD);
*       else
*         { if(PIN != _PIN[cardNumber] ) $assert ($result ==
*           Status.INVALID_PIN);
*         else
*           {if(_accountNumber[cardNumber][from] == 0) $assert (
*             $result == Status.NO_SUCH_ACCOUNT);
*           else
*             {if(from == SAVINGS) $assert ($result ==
*               Status.CANT_WITHDRAW_FROM_ACCOUNT);
*             else
*               if(_availableBalance
*                 [_currentTransactionAccount].less(
*                   amount)) $assert ( $result ==
*                     status.INSUFFICIENT_AVAILABLE_BALANCE);
*               else
*                 {if(MAXIMUM_WITHDRAWAL_AMOUNT_PER_DAY.getValue()
*                   < (_withdrawalsToday [ cardNumber].
*                     getValue()+ amount.getValue())) $assert(
*                       $result ==
*                         Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED);
*                 else
*                   $assert(_currentTransactionAmount.
*                     equals(amount)
*                     && newBalance.getValue() == _currentBalance
*                       [_currentTransactionAccount].getValue()
*                         - amount.getValue()
*                     && availableBalance.getValue() ==
*                       _availableBalance
*                         [_currentTransactionAccount].getValue()

```



```

--the post-condition below is the Intermediate Precise
postcondition
post: if ((self.card->exists(_cardNumber = cardNumber))
    and
    (PIN == self._currentTransactionCard._PIN)
    and
    (self._currentTransactionAccount._available)
    and
    (from <> SAVINGS)
    and
    (not self._currentTransactionAccount.
        _availableBalance.
        less(amount))
    and
    (not MAXIMUM_WITHDRAWAL_AMOUNT_PER_DAY.
        less( Money.add(
        self._currentTransactionCard_withdrawlsToday,
        amount))))
    ) then
    result = Status.SUCCESS
    and
    _currentTransactionAmount.equals(amount)
    and
    newBalance.getValue() =
        self._currentTransactionAccount._currentBalance.
        getValue() - amount.getValue()
    and
    availableBalance.getValue() =
        self._currentTransactionAccount._availableBalance
        .getValue() - amount.getValue()
else
    result = Status.UNKNOWN_CARD
    or
    result = Status.INVALID_PIN
    or
    result = Status.NO_SUCH_ACCOUNT
    or
    result = Status.CANT_WITHDRAW_FROM_ACCOUNT
    or
    result = Status.INSUFFICIENT_AVAILABLE_BALANCE
    or
    result = Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED
endif

```

The Intermediate Contract in Implementation:

```

/**
 * @pre serialNumber > 0
 * @pre from == CHECKING || from == SAVINGS || from == MONEY_MARKET
 *
 * @post (if (cardNumber>=1 && cardNumber<= NUMBER_CARDS) &&
 *         (PIN == _PIN[cardNumber]) &&
 *         (_accountNumber[cardNumber][from] == 0) &&
 *         (from <> SAVINGS) &&
 *         (!_availableBalance[_currentTransactionAccount].
 *          less(amount)) &&
 *         (MAXIMUM_WITHDRAWAL_AMOUNT_PER_DAY.getValue() >=
 *          (_withdrawlsToday [ cardNumber].getValue()+
 *          amount.getValue()))
 *         $assert ( (_currentTransactionAmount.equals(amount)
 *                    && newBalance.getValue() == _currentBalance
 *                     [_currentTransactionAccount].getValue() -
 *                     amount.getValue()
 *                    && availableBalance.getValue() ==

```

```

*
*         _availableBalance[_currentTransactionAccount].
*             getValue() - amount.getValue()
*         && $result == Status.SUCCESS))
*
*     else
*         $assert ( $result == Status.UNKNOWN_CARD ||
*             $result = Status.INVALID_PIN ||
*             $result = Status.NO_SUCH_ACCOUNT ||
*             $result = Status.CANT_WITHDRAW_FROM_ACCOUNT ||
*             $result = Status.INSUFFICIENT_AVAILABLE_BALANCE ||
*             $result = Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED )
*
*     :
*
*/

```

Bank::finishWithdraw(ATMnumber:int, serialNumber:int, succeeded:Boolean):void

This method finishes the withdraw operation when the withdraw operation is approved.

Contract in Analysis:

```

context: Bank::finishWithdraw(ATMnumber:int, serialNumber:int,
                                succeeded:Boolean):void
pre:   self.atmMachineProfile->exists(_ATMNumber = ATMnumber)
        serialNumber > 0

post:  succeeded implies
        --today's withdraws amount calculation, should be added the --
        withdraw amount

        self._currentTransactionCard._withdrawlsToday .getValue() =
            self._currentTransactionCard._withdrawlsToday .getValue()@pre
            + self._currentTransactionAmount._getValue()
        and
        --current balance calculation, should remove the withdraw ---
        amount
        self._currentTransactionAccount._currentBalance.getValue() =
            self._currentTransactionAccount._currentBalance.
            getValue()@pre - self._currentTransactionAmount.getValue()
        and
        --available balance also should remove the withdraw amount
        self._currentTransactionAccount._availableBalance.
            getValue() = self._currentTransactionAccount.
            _availableBalance.getValue()@pre -
            self._currentTransactionAmount.getValue()

```

Contract in Implementation:

```

/**
* @pre ATMnumber == 42
* @pre serialNumber > 0
*
* @post { if (succeeded) {
*     $assert(_withdrawlsToday[_currentTransactionCard].getValue() ==
*         $pre(long, _withdrawlsToday[_currentTransactionCard].
*             getValue()) + _currentTransactionAmount.getValue());
*
*     $assert( _currentBalance [_currentTransactionAccount ].getValue()
*         == $pre(long, _currentBalance[_currentTransactionAccount].
*             getValue()) - _currentTransactionAmount.getValue());
*
*     $assert( _availableBalance[_currentTransactionAccount ].getValue()

```

```

*           == $pre(long, _availableBalance[_currentTransactionAccount]
*               .getValue()) - currentTransactionAmount.getValue());
*           }
*       }
*/

```

Bank::initiateDeposit(cardNumber:int, PIN:int, ATMnumber:int, serialNumber:int, to:int, amount:Money, newBalance:Money, availableBalance:Money):int

This method processes the customer information that is transferred from customer deposit operation. We design the highest precise, lowest precise and intermediate precise contracts for this method.

Highest Precise Contract in Analysis:

```

context: Bank::initiateDeposit(cardNumber:int, PIN:int, ATMnumber:int,
                               serialNumber:int, to:int, amount:Money,
                               newBalance:Money, availableBalance:Money):int
pre:   self.aTMachineProfile->exists(_ATMNumber = ATMnumber)and
       serialNumber > 0 and
       to = CHECKING or to = SAVINGS or to = MONEY_MARKET

-- the Highest Precise contracts are designed as the
-- following
post:  if not self.card->exists(_cardNumber = cardNumber) then
       result = Status.UNKNOWN_CARD
     else
       if PIN <> self._currentTransactionCard._PIN then
         result = Status.INVALID_PIN
       else
         if not self._currentTransactionAccount._available then
           result = Status.NO_SUCH_ACCOUNT
         else
           _currentTransactionAmount = amount and

           -- new current balance in to account will be added
           -- the amount
           newBalance.getValue() =
             self._currentTransactionAccount._currentBalance.
               getValue()+ amount.getValue() and

           -- the available balance in to account will remain
           -- the amount
           availableBalance.getValue() =
             self._currentTransactionAccount._currentBalance.getVal
               ue() and

           result = Status.SUCCESS
         endif
       endif
     endif
endif

```

Highest Precise Contract in Implementation:

```

/**
* @pre serialNumber >0
* @pre ATMnumber == 42

```

```

* @pre to == CHECKING || to == SAVINGS || to == MONEY_MARKET
*
* @post
* {
*   if(cardNumber < 1 || cardNumber > NUMBER_CARDS)
*     $assert( $result == Status.UNKNOWN_CARD);
*   else
*     (if(PIN != _PIN[cardNumber])
*       $assert( $result == Status.INVALID_PIN);
*     else
*       (if(_currentTransactionAccount == 0)
*         $assert( $result == Status.NO_SUCH_ACCOUNT);
*       else
*         $assert( newBalance.getValue() ==
*                   _currentBalance[_accountNumber
*                                     [cardNumber][to]].getValue()+
*                   amount.getValue() &&
*
*                   availableBalance.getValue() ==
*                   _availableBalance[_accountNumber
*                                       [cardNumber][to]].getValue()&&
*
*                   $result == Status.SUCCESS);
*       )
*     )
*   }
* }
*/

```

Lowest Precise Contract in Analysis:

```

context: Bank::initiateDeposit(cardNumber:int, PIN:int, ATMnumber:int,
                                serialNumber:int, to:int, amount:Money,
                                newBalance:Money, availableBalance:Money):int
pre: self.atmMachineProfile->exists(_ATMNumber = ATMnumber) and
      serialNumber > 0 and
      to = CHECKING or to = SAVINGS or to = MONEY_MARKET

      -- the Lowest Precise contracts are designed as the
      -- following
post: result = Status.UNKNOWN_CARD or
      result = Status.INVALID_PIN or
      result = Status.NO_SUCH_ACCOUNT or
      (_currentTransactionAmount = amount and

      -- new current balance in to account will be added the
      -- amount
      newBalance.getValue() =
        self._currentTransactionAccount._currentBalance.
        getValue()+amount.getValue() and

      -- the available balance in to account will remain the
      -- amount
      availableBalance.getValue() =
        self._currentTransactionAccount._currentBalance.
        getValue() and
      result = Status.SUCCESS )

```

Lowest Precise Contract in Implementation:

```

/**
* @pre serialNumber >0
* @pre ATMnumber == 42
* @pre to == CHECKING || to == SAVINGS || to == MONEY_MARKET

```

```

*
* @post $result == Status.UNKNOWN_CARD ||
*       $result == Status.INVALID_PIN ||
*       $result == Status.NO_SUCH_ACCOUNT ||
*       ( newBalance.getValue() ==
*         _currentBalance[_accountNumber[cardNumber][to]].getValue()+
*         amount.getValue()
*         && availableBalance.getValue() ==
*         _availableBalance[_accountNumber[cardNumber][to]].
*           getValue()
*         && $result == Status.SUCCESS)
*/

```

Intermediate Precise Contract in Analysis:

```

context: Bank::initiateDeposit(cardNumber:int, PIN:int, ATMnumber:int,
                               serialNumber:int, to:int, amount:Money,
                               newBalance:Money, availableBalance:Money):int
pre:   self.ATMMachineProfile->exists(_ATMNumber = ATMnumber) and
       serialNumber > 0 and
       to = CHECKING or to = SAVINGS or to = MONEY_MARKET

-- the Intermediate Precise contracts are designed as the
-- following
post: if self.card->exists(_cardNumber= cardNumber) and
      PIN == self._currentTransactionCard._PIN and
      self._currentTransactionAccount._available then

    _currentTransactionAmount = amount and

    -- new current balance in to account will be added the
    -- amount
    newBalance.getValue() =
      self._currentTransactionAccount._currentBalance.
        getValue()+ amount.getValue() and

    -- the available balance in to account will remain the
    -- amount
    availableBalance.getValue() =
      self._currentTransactionAccount._currentBalance.
        getValue() and

    result = Status.SUCCESS
  else
    result = Status.UNKNOWN_CARD or
    result = Status.INVALID_PIN or
    result = Status.NO_SUCH_ACCOUNT
  endif

```

Intermediate Precise Contract in Implementation:

```

/**
* @pre serialNumber >0
* @pre ATMnumber == 42
* @pre to == CHECKING || to == SAVINGS || to == MONEY_MARKET
*
* @post
* {
*   if ((cardNumber>=1 && cardNumber <= NUMBER_CARDS) &&
*       (PIN == _PIN[cardNumber]) &&
*       (_currentTransactionAccount <> 0 ))
*     Sassert (newBalance.getValue() ==
*             _currentBalance[_accountNumber[cardNumber][to]].
*               getValue()+ amount.getValue() &&

```

```

*           availableBalance.getValue() ==
*             _availableBalance[_accountNumber][cardNumber][to]].
*             getValue() &&
*             $result == Status.SUCCESS)
*     else
*       $assert ( $result == Status.UNKNOWN_CARD ||
*                 $result == Status.INVALID_PIN ||
*                 $result == Status.NO_SUCH_ACCOUNT)
*   }
*
*/

```

Bank::finisheDeposit(ATMnumber:int,serialNumber:int,succeeded:Boolean):void

This method finishes the deposit operation when the bank approves the operation.

Contract in Analysis:

context:

Bank::finisheDeposit(ATMnumber:int,serialNumber:int,successful:Boolean):void

pre: self.aTMMachineProfile->exists(_ATMNumber = ATMnumber) and
serialNumber > 0

--when the PIN and account check is OK, the current balance will
--be added by the deposit amount

post: succeeded **implies**
self._currentTransactionAccount._currentBalance.
getValue() = self._currentTransactionAccount.
_currentBalance.getValue@pre +
currentTransactionAmount.getValue()

Contract in Implementation:

```

/**
* @pre      ATMnumber == 42
* @pre      serialNumber > 0
*
* @post
* {
*   if (succeeded)
*     $assert(_currentBalance[_currentTransactionAccount].getValue()
*             == $pre(long,_currentBalance[_currentTransactionAccount].
*                 getValue()) + _currentTransactionAmount.getValue());
* }
*/

```

Bank::doTransfer(cardNumber:int, PIN:int, ATMnumber:int, serialNumber:int, from:int, to:int, amount:Money, newBalance:Money, availableBalance:Money):int

This method processes the transfer operation.

Highest Precise Contract in Analysis:

context: Bank::doTransfer(cardNumber:int, PIN:int, ATMnumber:int, serialNumber:int, from:int, to:int, amount:Money, newBalance:Money, availableBalance:Money):int

pre: self.aTMMachineProfile->exists(_ATMNumber = ATMnumber) and
serialNumber > 0 and
from = CHECKING or from = SAVINGS or from = MONEY_MARKET and
to = CHECKING or to = SAVINGS or to = MONEY_MARKET

```

-- the highest precise contracts are designed as the
-- following
post: if not self.card->exists(_cardNumber = cardNumber) then
    result = Status.UNKNOWN_CARD
else
    if PIN <> self._currentTransactionCard._PIN then
        result = Status.INVALID_PIN
    else
        if not self.account->select(_accountType = from) or
            not self.account->select(_accountType = to ) then result =
            Status.NO_SUCH_ACCOUNT
        else
            if from = SAVINGS then
                result = Status.CANT_WITHDRAW_FROM_ACCOUNT
            else
                if self.account->select(_accountType = from)
                    _availableBalance.getValue()@pre < amount.getValue()
                then result = Status.INSUFFICIENT_AVAILABLE_BALANCE
                else
                    --update from account current balance
                    self.account->select(_accountType =
                    from)._currentBalance.getValue() =
                    self.account->select(_accountType =
                    from)._currentBalance.getValue()@pre
                    - amount.getValue() and

                    --update to account current balance
                    self.account->select(_accountType =
                    to)._currentBalance.getValue() =
                    self.account->select(_accountType =
                    to)._currentBalance.getValue()@pre +
                    amount.getValue() and

                    --update the from account available balance
                    self.account->select(_accountType=from).
                    _availableBalance.getValue() =
                    self.account->select(_accountType=from).
                    _availableBalance.getValue()@pre - amount.getValue()
                    and

                    --update the to account available balance
                    self.account->select(_accountType =
                    to)._availableBalance.getValue() = self.account->
                    >select(_accountType =
                    to)._availableBalance.getValue()@pre +
                    amount.getValue() and

                    --current balance for to account
                    newBalance.getValue() =
                    self.account->select(_accountType =
                    to)._currentBalance.getValue() and

                    --available balance for to account
                    availableBalance.getValue() =
                    self.account->select(_accountType=from).
                    _availableBalance.getValue()
                    and

                    result = Status.SUCCESS
                endif
            endif
        endif
    endif
endif

```


endif

Highest Precise Contract in Implementation:

```
/**
 * @pre ATMNumber == 42
 * @pre serialNumber >0
 * @pre from == CHECKING || from == SAVINGS || from == MONEY_MARKET
 * @pre to == CHECKING || to == SAVINGS || to == MONEY_MARKET
 *
 *
 * @post{
 *   if(cardNumber < 1 || cardNumber > NUMBER_CARDS) $assert($result
 *     == Status.UNKNOWN_CARD);
 *   else
 *     (if (PIN != _PIN[cardNumber]) $assert( $result ==
 *       Status.INVALID_PIN);
 *     else
 *       (if(_accountNumber[cardNumber][from] == 0 ||
 *         _accountNumber[cardNumber][to]== 0)
 *         $assert( $result == Status.NO_SUCH_ACCOUNT);
 *       else
 *         (if (from == SAVINGS) $assert( $result ==
 *           Status.CANT_WITHDRAW_FROM_ACCOUNT);
 *         else
 *           (if ($pre(long,_availableBalance
 *             [_accountNumber [ cardNumber ] [ from ]].
 *             getValue()) < amount.getValue()) $assert($result ==
 *             Status.INSUFFICIENT_AVAILABLE_BALANCE);
 *           else $assert( _currentBalance
 *             [ _accountNumber [ cardNumber ] [ from ] ].getValue()
 *             == $pre(long,_currentBalance[_accountNumber
 *             [cardNumber][ from ]].getValue()) - amount.getValue()
 *
 *             && _currentBalance[_accountNumber[cardNumber][ to ] ]
 *             .getValue() == $pre(long,_currentBalance
 *             [_accountNumber [ cardNumber ] [ to ] ]
 *             .getValue()+amount.getValue()
 *
 *             && _availableBalance[_accountNumber[cardNumber]
 *             [from]].getValue() ==
 *             $pre(long,_availableBalance[_accountNumber
 *             [cardNumber][from]].getValue()) - amount.getValue()
 *
 *             && _availableBalance[_accountNumber[cardNumber][to]]
 *             .getValue() == $pre(long,
 *             _availableBalance[_accountNumber[cardNumber][to]]
 *             .getValue() + amount.getValue()
 *
 *             && newBalance.getValue() ==
 *             _currentBalance[_accountNumber[cardNumber][to]]
 *             .getValue()
 *
 *             && availableBalance.getValue() ==
 *             _availableBalance[_accountNumber[cardNumber][to]]
 *             .getValue()
 *
 *             && $result == Status.SUCCESS);
 *           }
 *         }
 *       }
 *     }
 *   }
 * }
```

*/

Lowest Precise Contract in Analysis:

```
context: Bank:: doTransfer(cardNumber:int, PIN:int, ATMnumber:int,
                           serialNumber:int, from:int, to:int, amount:Money,
                           newBalance:Money, availableBalance:Money):int
pre:  self.ATMMachineProfile->exists(_ATMNumber = ATMnumber) and
        serialNumber > 0 and
        from = CHECKING or from = SAVINGS or from = MONEY_MARKET and
        to = CHECKING or to = SAVINGS or to = MONEY_MARKET

        -- the lowest precise contracts are designed as the
        -- following
post: result = Status.UNKNOWN_CARD or
        result = Status.INVALID_PIN or
        result = Status.NO_SUCH_ACCOUNT or
        result = Status.CANT_WITHDRAW_FROM_ACCOUNT or
        result = Status.INSUFFICIENT_AVAILABLE_BALANCE or
        (--update from account current balance
        self.account->select(_accountType =
        from)._currentBalance.getValue() =
        self.account->select(_accountType =
        from)._currentBalance.getValue()@pre - amount.getValue() and

        --update to account current balance
        self.account->select(_accountType = to)._currentBalance.getValue()
        =
        self.account->select(_accountType =
        to)._currentBalance.getValue()@pre + amount.getValue() and

        --update the from account available balance
        self.account->select(_accountType=from)._availableBalance.
        getValue() = self.account->select(_accountType=from).
        _availableBalance.getValue()@pre - amount.getValue() and

        --update the to account available balance
        self.account->select(_accountType = to)._availableBalance.
        getValue() = self.account->select(_accountType = to).
        _availableBalance.getValue()@pre + amount.getValue() and

        --current balance for to account
        newBalance.getValue() = self.account->select(_accountType =
        to)._currentBalance.getValue() and

        --available balance for to account
        availableBalance.getValue() = self.account->
        >select(_accountType=from)._availableBalance.getValue() and

        result = Status.SUCCESS )
```

Lowest Precise Contract in Implementation:

```
/**
 * @pre ATMnumber == 42
 * @pre serialNumber >0
 * @pre from == CHECKING || from == SAVINGS || from == MONEY_MARKET
 * @pre to == CHECKING || to == SAVINGS || to == MONEY_MARKET
 *
 *
 * @post $result == Status.UNKNOWN_CARD ||
 *       $result == Status.INVALID_PIN ||
 *       $result == Status.NO_SUCH_ACCOUNT ||
 *       $result == Status.CANT_WITHDRAW_FROM_ACCOUNT ||
```

```

*      $result == Status.INSUFFICIENT_AVAILABLE_BALANCE||
*      (_currentBalance [_accountNumber [cardNumber][from] ].
*      getValue()
*      == $pre(long,_currentBalance[_accountNumber[cardNumber ]
*      [ from ]] .getValue()) - amount.getValue()
*      && _currentBalance [_accountNumber [ cardNumber ]
*      [ to ] ].getValue()
*      == $pre(long,_currentBalance[_accountNumber[ cardNumber ]
*      [ to ]].getValue()+amount.getValue()
*      && _availableBalance[_accountNumber[cardNumber][from]].
*      getValue() == $pre(long,_availableBalance[_accountNumber
*      [cardNumber][from] ].getValue()) - amount.getValue()
*      && _availableBalance[_accountNumber[cardNumber][to]].
*      getValue() == $pre(long, _availableBalance[_accountNumber
*      [cardNumber][to]].getValue() + amount.getValue()
*      && newBalance.getValue() ==
*      _currentBalance[_accountNumber[cardNumber][to]].getValue()
*      && availableBalance.getValue() ==
*      _availableBalance[_accountNumber
*      [cardNumber][to]].getValue()
*      && $result == Status.SUCCESS)
*/

```

Intermediate Precise Contract in Analysis:

```

context: Bank:: doTransfer(cardNumber:int, PIN:int, ATMnumber:int,
                           serialNumber:int, from:int, to:int, amount:Money,
                           newBalance:Money, availableBalance:Money):int
pre:   self.ATMMachineProfile->exists(_ATMNumber = ATMnumber) and
       serialNumber > 0 and
       from = CHECKING or from = SAVINGS or from = MONEY_MARKET and to =
       CHECKING or to = SAVINGS or to = MONEY_MARKET

-- the intermediate precise contracts are designed as the ---
following
post:  if   self.card->exists(_cardNumber = cardNumber) and
         PIN == self._currentTransactionCard._PIN and
         self.account->select(_accountType = from) and
         self.account->select(_accountType = to ) and
         from <> SAVINGS and
         self.account->select(_accountType = from)
         _availableBalance.getValue()@pre >= amount.getValue()
then
    --update from account current balance
    self.account->select(_accountType =
    from)._currentBalance.getValue()
    = self.account->select(_accountType
    =from)._currentBalance.getValue()@pre
    - amount.getValue() and

    --update to account current balance
    self.account->select(_accountType =
    to)._currentBalance.getValue() =
    self.account->select(_accountType =
    to)._currentBalance.getValue()@pre + amount.getValue() and

    --update the from account available balance
    self.account>select(_accountType=from).
    _availableBalance.getValue() = self.account -
    select(_accountType=from)._available.Balance.
    getValue()@pre - amount.getValue() and

```

```

--update the to account available balance
self.account->select(_accountType =
to)._availableBalance.getValue()
= self.account->select(_accountType =
to)._availableBalance.getValue()@pre + amount.getValue()
and

--current balance for to account
newBalance.getValue() = self.account->select(_accountType =
to)._currentBalance.getValue() and

--available balance for to account
availableBalance.getValue() = self.account-
>select(_accountType=from)._availableBalance.
getValue()
and

result = Status.SUCCESS )
else
result = Status.UNKNOWN_CARD or
result = Status.INVALID_PIN or
result = Status.NO_SUCH_ACCOUNT or
result = Status.CANT_WITHDRAW_FROM_ACCOUNT or
result = Status.INSUFFICIENT_AVAILABLE_BALANCE
end if

```

Intermediate Precise Contract in Implementation:

```

/**
 * @pre ATMnumber == 42
 * @pre serialNumber >0
 * @pre from == CHECKING || from == SAVINGS || from == MONEY_MARKET
 * @pre to == CHECKING || to == SAVINGS || to == MONEY_MARKET
 *
 * @post(
 *   if ((cardNumber >= 1 && cardNumber <= NUMBER_CARDS) &&
 *      (PIN == _PIN[cardNumber]) &&
 *      (_accountNumber[cardNumber][from] <> 0) &&
 *      (_accountNumber[cardNumber][to] <> 0) &&
 *      (from <> SAVINGS) &&
 *      ($pre(long,_availableBalance[_accountNumber[cardNumber]
 *      [from]].getValue()) >= amount.getValue()))
 *   Sassert(_currentBalance[_accountNumber[cardNumber][from]]
 *     .getValue() ==
 *     $pre(long,_currentBalance[_accountNumber
 *     [cardNumber][from]].getValue() -
 *     amount.getValue()
 *     && _currentBalance[_accountNumber[cardNumber][to]]
 *     .getValue() == $pre(long,_currentBalance[_accountNumber
 *     [cardNumber][to]].getValue()+amount.getValue()
 *     && _availableBalance[_accountNumber[cardNumber][from]]
 *     .getValue() == $pre(long,_availableBalance
 *     [_accountNumber[cardNumber][from]].getValue()
 *     - amount.getValue()
 *     && _availableBalance[_accountNumber[cardNumber][to]]
 *     .getValue() == $pre(long,_availableBalance
 *     [_accountNumber[cardNumber][to]].getValue() +
 *     amount.getValue()
 *     && newBalance.getValue() == _currentBalance[_accountNumber
 *     [cardNumber][to]].getValue()
 *     && availableBalance.getValue() ==
 *     _availableBalance[_accountNumber[cardNumber][to]]

```

```

*         .getValue()
*         && $result == Status.SUCCESS))
*     else
*         Sassert ($result == Status.UNKNOWN_CARD ||
*                 $result == Status.INVALID_PIN ||
*                 $result == Status.NO_SUCH_ACCOUNT ||
*                 $result == Status.CANT_WITHDRAW_FROM_ACCOUNT ||
*                 $result == Status.INSUFFICIENT_AVAILABLE_BALANCE||)
*
*     }
*
*/

```

Bank::doInquiry(cardNumber:int, PIN:int, ATMnumber:int, serialNumber:int, from:int, amount:Money, newBalance:Money, availableBalance:Money):int

This method handles the balance inquiry request.

Highest Precise Contract in Analysis:

```

context: Bank::doInquiry(cardNumber:int, PIN:int, ATMnumber:int,
                           serialNumber:int, from:int, amount:Money,
                           newBalance:Money, availableBalance:Money):int
pre:  self.atmMachineProfile->exists(_ATMNumber = ATMnumber) and
        serialNumber > 0 and
        from = CHECKING or from = SAVINGS or from = MONEY_MARKET

        -- the highest precise contracts are designed as the
        -- following
post: if not self.card->exists(_cardNumber = cardNumber) then
        result = Status.UNKNOWN_CARD
        else if PIN <> self._currentTransactionCard._PIN then
        result = Status.INVALID_PIN
        else if not self._currentTransactionAccount._available then
        result = Status.NO_SUCH_ACCOUNT
        else
            --get the new balance for inquiry account
            newBalance.getValue() = self.account
            ->select(_accountType =
            from)._currentBalance.getValue()
            and

            --get the available balance for inquiry
            --account
            availableBalance.getValue() =
            self.account->select(_accountType =
            from)._availableBalance.getValue() and

            result = Status.SUCCESS
        endif
    endif
endif

```

Highest Precise Contract in Implementation:

```

/**
* @pre ATMnumber == 42
* @pre serialNumber > 0
* @pre from == CHECKING || from == SAVINGS || from == MONEY_MARKET
*
* @post{
*     if(cardNumber < 1 || cardNumber > NUMBER_CARDS) Sassert($result ==

```

```

*           Status.UNKNOWN_CARD);
*
*   else
*       {if(PIN != _PIN[cardNumber]) $assert ($result ==
*           Status.INVALID_PIN);
*
*       else
*           {if( _accountNumber[cardNumber][from] == 0)
*               $assert( $result == Status .NO_SUCH_ACCOUNT);
*           else $assert (newBalance.getValue() ==
*               _currentBalance[_accountNumber[cardNumber]
*                   [from]].getValue()
*               && availableBalance.getValue() ==
*                   _availableBalance[_accountNumber
*                       [cardNumber][from]].getValue()
*               && $result == Status.SUCCESS
*               && newBalance.getValue() >=
*                   availableBalance.getValue());
*           }
*       }
*   }
*
*/

```

Lowest Precise Contract in Analysis:

```

context: Bank::doInquiry(cardNumber:int, PIN:int, ATMnumber:int,
    serialNumber:int, from:int, amount:Money,
    newBalance:Money, availableBalance:Money):int
pre: self.ATMMachineProfile->exists(_ATMNumber = ATMnumber) and
    serialNumber > 0 and
    from = CHECKING or from = SAVINGS or from = MONEY_MARKET

    -- the lowest precise contracts are designed as the
    -- following
post: result = Status.UNKNOWN_CARD or
result = Status.INVALID_PIN or
result = Status .NO_SUCH_ACCOUNT or
    (--get the new balance for inquiry account
    newBalance.getValue() = self.account->select(_accountType =
    from)._currentBalance.getValue() and

    --get the available balance for inquiry account
    availableBalance.getValue() = self.account-
    >select(_accountType = from)._availableBalance.getValue() and

    result = Status.SUCCESS )

```

Lowest Precise Contract in Implementation:

```

/**
* @pre ATMnumber == 42
* @pre serialNumber > 0
* @pre from == CHECKING || from == SAVINGS || from == MONEY_MARKET
*
* @post $result == Status.UNKNOWN_CARD||
*       $result == Status.INVALID_PIN||
*       $result == Status .NO_SUCH_ACCOUNT||
*       (newBalance.getValue() ==
*         _currentBalance[_accountNumber[cardNumber]
*             [from]].getValue() &&
*         availableBalance.getValue() ==
*         _availableBalance[_accountNumber[cardNumber][from]].getValue()
*         &&
*         $result == Status.SUCCESS &&
*         newBalance.getValue() >= availableBalance.getValue())

```

*/

Intermediate Precise Contract in Analysis:

```
context: Bank::doInquiry(cardNumber:int, PIN:int, ATMnumber:int,
                          serialNumber:int, from:int, amount:Money,
                          newBalance:Money, availableBalance:Money):int
pre:   self.ATMMachineProfile->exists(_ATMNumber = ATMnumber) and
       serialNumber > 0 and
       from = CHECKING or from = SAVINGS or from = MONEY_MARKET

-- the intermediate precise contracts are designed as the
-- following
post:  if   self.card->exists(_cardNumber = cardNumber) and
          PIN == self._currentTransactionCard._PIN and
          self._currentTransactionAccount._available
       then

           (--get the new balance for inquiry account
           newBalance.getValue() =
           self.account->select(_accountType =
           from)._currentBalance.getValue() and

           --get the available balance for inquiry account
           availableBalance.getValue() =
           self.account->select(_accountType =
           from)._availableBalance.getValue() and

           result = Status.SUCCESS )
       else
           result = Status.UNKNOWN_CARD or
           result = Status.INVALID_PIN or
           result = Status.NO_SUCH_ACCOUNT
       endif
```

Intermediate Precise Contract in Implementation:

```
/**
 * @pre ATMnumber == 42
 * @pre serialNumber > 0
 * @pre from == CHECKING || from == SAVINGS || from == MONEY_MARKET
 *
 * @post{
 *   if (cardNumber >= 1 || cardNumber <= NUMBER_CARDS) &&
 *     (PIN != _PIN[cardNumber]) &&
 *     (_accountNumber[cardNumber][from] <> 0 )
 *   $assert(newBalance.getValue() ==
 *     _currentBalance[_accountNumber
 *     [cardNumber][from]].getValue()
 *     && availableBalance.getValue() ==
 *     _availableBalance[_accountNumber
 *     [cardNumber][from]].getValue()
 *     && $result == Status.SUCCESS
 *     && newBalance.getValue() >= availableBalance.getValue())
 *   else
 *     $assert ($result == Status.UNKNOWN_CARD||
 *       $result == Status.INVALID_PIN||
 *       $result == Status.NO_SUCH_ACCOUNT|| )
 * }
 */
```

Bank::chooseAccountType(purpose:String, atm:ATM):int

This method is used for user to choose an account.

Contract in Analysis:

```
context: Bank::chooseAccountType(purpose:String, atm:ATM):int
  pre: atm -> notEmpty()
  post: self.account->exists(_accountType=result)
  post: result = atm.getMenuChoice("Please choose an account to" +
    purpose, 3, {"Checking", "Saving", "Money Market"}) - 1
```

Contract in Implementation:

The last postcondition is implemented with `@assert` in the method body.

```
/**
 * @pre atm != null
 * @post $result <3 && $result >= 0
 */

public int chooseAccountType(String purpose, ATM atm)
{ String menu[] = { "Checking", "Savings", "Money Market" };
  String prologue = "Please choose an account to ";
  int choice = atm.getMenuChoice(prologue + purpose, 3, menu);
  int result = choice - 1; // result is local vairiable to support the contract
  /* @assert result == choice - 1 */
  return result;
}
```

Bank::accountName(type:int):String

This method returns the customer account information.

Contract in Analysis:

```
context: Bank:: accountName(type:int):String
  pre: type >= 0 and type <= AccountType::MONEYMARKET
  post: result = self.account._accountName->at(type)
```

Contract in Implementation:

```
/**
 * @pre type >= 0 && type < accountNames.length
 * @post $result == accountNames[type]
 */
```

Bank::rejectionExplanation(code:int):String

This method returns the transaction rejection message.

Contract in Analysis:

```
context: Bank:: rejectionExplanation(code:int):String
  pre: code < =self.explanation._message->size() and
    code >= Status.UNKNOWN_CARD
  post: result = self.explanation._message->at(code -
    Status.UNKNOWN_CARD)
```

Contract in Implementation:

```
/**
 * @pre code < explanation.length+3 &&
 *   code >=Status.UNKNOWN_CARD
 * @post $result == explanation[code - Status.UNKNOWN_CARD]
 */
```


Bank::verifyAccount(account:int):Boolean

This method is added to support the contracts because the constants, CHECKING, SAVINGS and MONEY_MARKET, are private attributes of Bank Class and can't be accessed by other class method. The purpose of this method is checking whether the account number is valid, i.e. which is CHECKING, SAVINGS or MONEY MARKET. So we don't need to design the contracts for this method.

Class CardReader

Class Attributes:

```
NO_CARD : static final int           // the value is set as 0
UNREADABLE_CARD : static final int   // the value is set as 1
CARD_HAS_BEEN_READ: static final int // the value is set as 2
```

Instance Attributes:

```
_status: int           // card read status
_cardNumberRead : int   // the card number
_originalBounds : Rectangle
```

Invariant

Contract in Analysis:

```
context: CardReader
    inv:  self._status == CARD_HAS_BEEN_READ or
          self._status == UNREADABLE_CARD or
          self._status == NO_CARD
```

Contract in Implementation:

```
/**
 * @invariant _status == CARD_HAS_BEEN_READ ||
 *             _status == UNREADABLE_CARD ||
 *             _status == NO_CARD
 */
```

CardReader::CardReader():void

This method is the constructor of CardReader class.

Contract in Analysis:

```
context: CardReader::CardReader():void
    post: self._status = NO_CARD
```

Contract in Implementation:

```
/**
 * @post _status == NO_CARD
 */
```

CardReader::EjectCard():void

This method ejects the customer's card from ATM machine.

Contract in Analysis:

```
context CardReader:: EjectCard():void
    post: self._status = NO_CARD
```

Contract in Implementation:

```
/**
 * @post _status == NO_CARD
 */
```

CardReader::retainCard():void

This method retains the customer card.

Contract in Analysis:

```
context CardReader::retainCard():void
    post self._status = NO_CARD
```

Contract in Implementation:

```
/**
 * @post _status == NO_CARD
 */
```

CardReader::checkForCardInserted(int number):int

This method checks whether the card is inserted into the machine or, operator turn off the machine.

Contract in Analysis:

```
context CardReader::checkForCardInserted():int
    post:result = UNREADABLE_CARD or
        (result = CARD_HAS_BEEN_READ and
         self._cardNumberRead = number)
```

Contract in Implementation:

```
/**
 * @post ($result == UNREADABLE_CARD) || ($result==CARD_HAS_BEEN_READ &&
 *     _cardNumberRead == number)
 */
```

CardReader::cardNumber():int

This method returns the card number.

Contract in Analysis:

```
context CardReader::cardNumber():int
    post result = self._cardNumberRead
```

Contract in Implementation:

```
/**
 * @post $result==_cardNumberRead
 */
```

Class CashDispenser**Instance Attributes:**

```
_currentCash: Money
_label: Label
```

Invariant

None

CashDispenser::CashDispenser():void

This method is the constructor of Cash Dispense class.

Contract in Analysis:

```
context CashDispenser::CashDispenser():void
    post self._currentCash.equals(new Money (0))
```

Contract in Implementation:

```
/**
 * @post _currentCash.equals(new Money(0))
 */
```

CashDispenser::setCash(initialCash:Money):void

This method sets the current cash amount.

Contract in Analysis:

```
context CashDispenser::setCash(initialCash:Money):void
    post: self._currentCash.equals(initialCash)
```

Contract in Implementation:

```
/**
 * @post _currentCash.equals(initialCash)
 */
```

CashDispenser::dispenseCash(amount:Money):void

This method dispenses current cash amount.

Contract in Analysis:

-- Current cash amount should be more than or equal the amount which will be dispensed.

```
context CashDispenser::dispenseCash(amount:Money):void
    pre: amount.getValue() <= self._currentCash.getValue()
    post: self._currentCash.getValue() = self._currentCash.getValue()@pre-
        amount.getValue()
```

Contract in Implementation:

```
/**
 * @pre amount.getValue() <= _currentCash.getValue()
 * @post _currentCash.getValue() == $pre(long,_currentCash.getValue()) -
 *                                     amount.getValue()
 */
```

CashDispenser::currentCash():Money

This method returns the amount of current cash.

Contract in Analysis:

```
Context: CashDispenser::currentCash():Money
    post: result.equals(self._currentCash)
```

Contract in Implementation:

```
/**
 * @post $result.equals(_currentCash)
 */
```

Class EnvelopeAcceptor

Instance Attributes:

`_originalBounds : Rectangle`
`_inserted : Boolean`

Invariant

There is no invariant designed for EnvelopeAcceptor class.

EnvelopeAcceptor::EnvelopeAcceptor()

This method is the constructor of EnvelopeAcceptor class.

Contract in Analysis:

context: EnvelopeAcceptor::EnvelopeAcceptor()
post: self._inserted = false

Contract in Implementation:

```
/**  
 * @post _inserted == false  
 **/
```

EnvelopeAcceptor::acceptEnvelope(String insert):Boolean

This method identifies whether the customer inserts the deposit envelope or not.

Contract in Analysis:

context: EnvelopeAcceptor::acceptEnvelope(String insert):Boolean
post: result = insert.equals("insert")

Contract in Implementation:

```
/**  
 * @post $result == insert.equals("insert")  
 **/
```

Class Money

In this class there is one method defined for the purpose of contract, method `getValue()`.

This method returns the value of attribute, `_cents`, because `_cents` is a private member variable and could not be accessed out of Money class.

Instance Attributes:

`_cents : long`

Invariant

Contract in Analysis:

context: Money
inv: self._cents >= 0

Contract in Implementation:

```
/**
```

```
* @invariant _cents >= 0
**/
```

Money::Money()

This method is the constructor of Class Money, which is used to initiate a Money object whose value is 0.

Contract in Analysis:

```
context: Money::Money()
post: self._cents = 0
```

Contract in Implementation:

```
/**
* @post _cents==0
**/
```

Money::Money(dollars:int)

This method is the constructor of Money class, which is used to initiate the Money object with dollar.

Contract in Analysis:

```
context: Money::Money(dollars:int)
pre: dollars >= 0
post: self._cents = 100 * dollars
```

Contract in Implementation:

```
/**
* @pre dollars >= 0
* @post _cents == 100 * (new Integer (dollars)).floatValue()
**/
```

Money::Money(dollars:int, cents:long)

This method is the constructor of Money class, which is used to initiate the Money object with dollar and cent.

Contract in Analysis:

```
context: Money::Money(dollars:int, cents:long)
pre: dollars >= 0
pre: cents >= 0
post: self._cents = 100*dollars + cents
```

Contract in Implementation:

```
/**
* @pre dollars >= 0 && cents >= 0
* @post _cents == 100* (new Integer (dollars)).floatValue() + cents
**/
```

Money::set(value:Money):void

This method sets the amount value.

Contract in Analysis:

```
context: Money::set(value:Money):void
      post: self._cents = value.getValue()
```

Contract in Implementation:

```
/**
 * @post _cents == value.getValue()
 */
```

Money::add(first:Money, second:Money):Money

This method calculates the total amount of the two times money added.

Contract in Analysis:

```
context: Money::add(first:Money, second:Money):Money
      post: result=first.getValue() + second.getValue()
```

Contract in Implementation:

```
/**
 * @post $result.getValue() == first.getValue()+second.getValue()
 */
```

Money::subtract(minuend:Money,subtrahend:Money):Money

This method calculates the amount that the value of minuend minus the value of subtrahend.

Contract in Analysis:

```
context: Money::subtract(minuend:Money, subtrahend:Money):Money
      pre: not minuend.less(subtrahend)
      post: result.getValue()=minuend.getValue()-subtrahend.getValue()
```

Contract in Implementation:

```
/**
 * @pre minuend.getValue() >= subtrahend.getValue()
 * @post $result.getValue() == minuend.getValue() - subtrahend.getValue()
 */
```

Money::add(other:Money):Money

This method calculates the result when the amount increases.

Contract in Analysis:

```
context: Money::add(other:Money):Money
      post: result.getValue() = self.getValue()@pre + other.getValue()
```

Contract in Implementation:

```
/**
 * @post $result.getValue() == $pre(long,_cents) + other.getValue()
 */
```

Money::subtract(other:Money):Money

This method calculates the result when the amount decreases.

Contract in Analysis:

```

context: Money::subtract(other:Money):Money
    pre: other.getValue() <=self.getValue()
    post: result.getValue() = self.getValue()@pre - other.getValue()

```

Contract in Implementation:

```

/**
 * @pre other.getValue() <= _cents
 * @post $result.getValue() == $pre(long,_cents) - other.getValue()
 */

```

Money::dollars():int

This method returns the dollar value of the money.

Contract in Analysis:

```

context: Money::dollars():int
    post: result.getValue() = self.getValue().div(100)

```

Contract in Implementation:

```

/**
 * @post $result == (new Float(_cents / 100)).intValue()
 */

```

Money::cents():int

This method returns the cents value of the money.

Contract in Analysis:

```

context: Money::cents():int
    post: result.getValue() = self.getValue().mod(100)

```

Contract in Implementation:

```

/**
 * @post $result == (new Float(_cents % 100)).intValue()
 */

```

Money::equals(other:Money):boolean

This method compares whether the two amounts are equal.

Contract in Analysis:

```

context: Money::equals(other:Money):boolean
    post: result = (other.getValue() = self.getValue())

```

Contract in Implementation:

```

/**
 * @post $result == (other.getValue()== _cents)
 */

```

Money::less(other:Money):Boolean

This method compares whether the value of other object is less than the value of this money object.

Contract in Analysis:

```

context: Money::less(other:Money):Boolean
    post: result = (other.getValue() >self.getValue())

```

Contract in Implementation:

```

/**

```

```
* @post $result == (other.getValue() > _cents)
**/
```

Money::getValue():long

This method returns the private attribute of Class Money, `_cents`. It is added to support the contracts, therefore there is no contract defined for this method.

Class OperatorPanel

Instance Attributes:

```
_message : Label
_offButton : Checkbox
_onButton : Checkbox
password : String
```

Invariant

None

OperatorPanel::OperatorPanel()

This method is the constructor of OperatorPanel class.

Contract in Analysis:

```
context: OperatorPanel::OperatorPanel()
post: self.password -> notEmpty()
```

Contract in Implementation:

```
/**
 * @post password !=null
 **/
```

OperatorPanel::switchOn():Boolean

This method reports whether the ATM system is switched on or not. This is a delegation method and no postcondition designed.

OperatorPanel::getInitialCash(String answer):Money

This method transfers the number of \$20 bills to the amount of initial cash.

Contract in Analysis:

```
context: OperatorPanel::getInitialCash(String answer):Money
pre: Integer.parseInt(answer)>0
post: result.equals(new Money(20*Integer.parseInt(answer)))
```

Notes: We have to use non-OCL operation, `Integer.parseInt (String)`, because there is no equivalent OCL operation.

Contract in Implementation:

```
/**
 * @pre Integer.parseInt(answer)>0
```



```
* @post $result.equals(new Money(20*Integer.parseInt(answer)))
**/
```

OperatorPanel::switchApprove(String password):Boolean

This method decides whether the switch can be approved or not.

Contract in Analysis:

**CONTEXT: OPERATORPANEL::SWITCHAPPROVE(STRING
PASSWORD):BOOLEAN**

```
post: result = password.equals("123456")
```

Contract in Implementation:

```
/**
 * @post $result == password.equals("123456")
 **/
```

Class ReceiptPrinter

Attributes

None.

Invariant

None

ReceiptPrinter::ReceiptPrinter()

This is the constructor of ReceiptPrinter class and no contract designed.

ReceiptPrinter::printReceipt(int theATMnumber, String theATMlocation, int cardNumber, int serialNumber, String description, Money amount, Money balance, Money availableBalance, String [] receipt):void

This method is responsible for printing the receipt of each transaction with the same format.

Contract in Analysis:

```
context: ReceiptPrinter::printReceipt(int theATMnumber, String theATMlocation,
int cardNumber, int serialNumber, String
description, Money amount, Money balance,
Money availableBalance, String []
receipt):void
pre: self.atmMachineProfile->exists(_ATMNumber = ATMnumber)
pre: self.atmMachineProfile->exists(_ATMLocation = theATMlocation)
pre: serialNumber > 0
pre: balance.getValue() >= availableBalance.getValue()

post: receipt[0] = "RECEIPT"
post: receipt[1] = theATMnumber + " " + theATMlocation + "\n"
post: receipt[2] = "CARD " + cardNumber + " TRANS " + serialNumber
+ "\n"
post: receipt[3] = description + "\n"
post: if (amount.equals(new Money(0))) then receipt[4] = "\n"
else
if amount.cents() >= 10 then
```

```

        receipt[4] = "AMOUNT:      " + amount.dollars() + "." + "" +
        amount.cents()
    else receipt[4] = "AMOUNT:      " + amount.dollars + "." + "0" +
        amount.cents() + "\n"
    endif
post: if balance.cents() >= 0 then
    receipt[5] = "CURR BAL:  " +
        balance.dollars() + "." + "" + balance.cents() : "0" +
        balance.cents() + "\n"
    else receipt[5] = "CURR BAL:  " + balance.dollars + "." + "0" +
        balance.cents() + "\n"
    endif
post: if availableBalance.cents() >= 10 then
    receipt[6] = "AVAILABLE: " + availableBalance.dollars() + "." +
        "" + availableBalance.cents() + "\n";
    else receipt[6] = "AVAILABLE: " + availableBalance.dollars()
        + "." + "0" + availableBalance.cents() + "\n"

```

Contract in Implementation:

```

/**
 * @pre theATMnumber == 42
 * @pre theATMlocation.equals("GORDON COLLEGE")
 * @pre cardNumber >= 0
 * @pre serialNumber > 0
 * @pre !balance.less(availableBalance)
 *
 * @post receipt[0] == "RECEIPT\n"
 * @post receipt[1].equals( theATMnumber + " " + theATMlocation + "\n")
 * @post receipt[2].equals("CARD " + cardNumber + " TRANS " + serialNumber +
 * "\n")
 * @post receipt[3].equals(description + "\n")
 * @post {
 *     if (amount.equals(new Money(0)))
 *         $assert (receipt[4].equals( "\n"));
 *     else
 *         $assert (receipt[4].equals( "AMOUNT:      " +
 *             amount.dollars() + "." + ((amount.cents() >= 10) ?
 *             "" + amount.cents(): "0" + amount.cents()) +
 *             "\n"));
 *     }
 *
 * @post receipt[5].equals( "CURR BAL:  " + balance.dollars() + "." +
 *     ((balance.cents() >= 10) ? "" + balance.cents()
 *     : "0" + balance.cents()) + "\n")
 * @post receipt[6].equals( "AVAILABLE: " + availableBalance.dollars() +
 *     "." + ((availableBalance.cents() >= 10) ? "" +
 *     availableBalance.cents(): "0" + availableBalance.cents()) +
 *     "\n")
 */

```

Class Session

Class Attributes:

```

RUNNING : static final int          // value is set as 0
FINISHED : static final int         // value is set as 1
ABORTED : static final int          // value is set as 2

```

Instance Attributes:

```

_cardNumber : int
_atm : ATM

```

```

_bank : Bank
_state : int
_PIN : int
_currentTransaction : Transaction

```

Invariant

Contract in Analysis:

```

context: Session
    inv: self._state = RUNNING or self._state = FINISHED or self._state =
        ABORTED

```

Contract in Implementation:

```

/**
 * @invariant _state == RUNNING || _state == FINISHED || _state == ABORTED
 */

```

Session::Session(cardNumber:int, atm:ATM, bank:Bank):void

This method is the constructor of Session class.

Contract in Analysis:

```

context: Session::Session(cardNumber:int, atm:ATM, bank:Bank):void
    post: self._state = RUNNING
    post: self._PIN = 0
    post: self._atm = atm
    post: self._bank = bank
    post: self._cardNumber = cardNumber
    post: self._currentTransaction->isEmpty()

```

Contract in Implementation:

```

/**
 * @post _state==RUNNING
 * @post _PIN == 0
 * @post _atm == atm
 * @post _bank == bank
 * @post _cardNumber == cardNumber
 * @post _currentTransaction == null
 */

```

Session::doSessionUseCase():void

This method handles the session to be continual or aborted according to the information from transaction class.

Contract in Analysis:

```

context: Session::doSessionUseCase():void
    post: self._state = FINISHED or self._state = ABORTED

```

Contract in Implementation:

```

/**
 * @post _state == FINISHED || _state == ABORTED
 */

```

Session::doInvalidPINExtension():int

This method handles the invalid PIN number.

Contract in Analysis:

```
context: Session::doInvalidPINExtension():int
    post: (Status->includes(result) and result <> Status.INVALID_PIN
        and _state <>ABORTED)
    or (self._state = ABORTED and result = Status.INVALID_PIN )
```

Contract in Implementation:

```
/**
 * @post (Status.contains($result) && $result !=Status.INVALID_PIN && _state !=
 * ABORTED) || (_state == ABORTED && $result == Status.INVALID_PIN)
 */
```

Session::doFailedTransactionExtension(reason:int):Boolean

This method handles the other wrong operations such as the balance of the account is not enough for transfer or withdraw, or the deposit envelope is not inserted etc.

Contract in Analysis:

```
context: Session::doFailedTransactionExtension(reason:int):Boolean
    pre: reason = Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED or
        reason = Status.ENVELOPE_DEPOSIT_TIMED_OUT or
        reason = Status.UNKNOWN_CARD or
        reason = Status.INVALID_PIN or
        reason = Status.NO_SUCH_ACCOUNT or
        reason = Status.CANT_WITHDRAW_FROM_ACCOUNT or
        reason = Status.INSUFFICIENT_AVAILABLE_BALANCE or
        reason = Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED or
        reason = Status. TOO_LITTLE_CASH
```

Contract in Implementation:

```
/**
 * @pre reason == Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED ||
 * reason == Status.ENVELOPE_DEPOSIT_TIMED_OUT ||
 * reason == Status.UNKNOWN_CARD ||
 * reason == Status.INVALID_PIN ||
 * reason == Status.NO_SUCH_ACCOUNT ||
 * reason == Status.CANT_WITHDRAW_FROM_ACCOUNT ||
 * reason == Status.INSUFFICIENT_AVAILABLE_BALANCE ||
 * reason == Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED ||
 * reason == Status. TOO_LITTLE_CASH
 */
```

Session::cardNumber():int

This method returns the card number.

Contract in Analysis:

```
context: Session::cardNumber():int
    post: result = _cardNumber
```

Contract in Implementation:

```
/**
 * @post $result == _cardNumber
 */
```

Session::PIN():int

This method returns the PIN number.

Contract in Analysis:

```
context: Session::PIN():int
post: result = _PIN
```

Contract in Implementation:

```
/**
 * @post $result == _PIN
 */
```

Class Status

This class defines the values of the different states in ATM transaction, such as successful, invalid PIN and so on.

Class Attributes:

```
SUCCESS : static final int // value is set as 0
TOO_LITTLE_CASH : static final int // value is set as 1
ENVELOPE_DEPOSIT_TIMED_OUT : static final int // value is set as 2
UNKNOWN_CARD : static final int // value is set as 3
INVALID_PIN : static final int // value is set as 4
NO_SUCH_ACCOUNT : static final int // value is set as 5
CANT_WITHDRAW_FROM_ACCOUNT : static final int // value is set as 6
INSUFFICIENT_AVAILABLE_BALANCE : static final int // value is set as 7
DAILY_WITHDRAWAL_LIMIT_EXCEEDED : static final int // value is set as 8
```

Invariant

There is no any contract can be written for this class, because this class is stateless and methodless class.

Class Transaction

This class lets the customer to choose a transaction he/she wants and starts the specific transaction procedure. There are four classes, WithdrawlTransaction, DepositTransaction, TransaferTransaction and InquiryTransaction, inherited from this class.

Instance Attributes:

```
_session : Session
_atm : ATM
_bank : Bank
_serialNumber : int
_newBalance : Money
_availableBalance : Money
_lastSerialNumberAssigned : int // value is initiated as 0
```

Invariant

Contract in Analysis:

```

context: Transaction
    inv: self._serialNumber > 0 and
        not self._newBalance.less(self._availableBalance)

```

Contract in Implementation:

```

/**
 * @invariant _serialNumber > 0
 * @invariant !_newBalance.less(_availableBalance)
 */

```

Transaction::Transaction(session:Session, atm:ATM, bank:Bank)
 This method is the constructor of Transaction class.

Contract in Analysis:

```

context: Transaction::Transaction(session:Session, atm:ATM, bank:Bank)
    post: self._serialNumber = _lastSerialNumberAssigned@pre + 1 and
        self._lastSerialNumberAssigned =
            self._lastSerialNumberAssigned@pre + 1 and
            self._session = session and
            self._atm = atm and
            self._bank = bank and
            self._newBalance = new Money() and
            self._availableBalance = new Money()

```

Contract in Implementation:

```

/**
 * @post _serialNumber == $pre(int,_lastSerialNumberAssigned) + 1
 * && _lastSerialNumberAssigned == $pre(int,_lastSerialNumberAssigned)
 *    +1
 * && _session == session
 * && _atm == atm
 * && _bank == bank
 * && _newBalance.equals(new Money())
 * && _availableBalance.equals(new Money())
 */

```

Transaction::chooseTransaction(session:Session, atm:ATM, bank:Bank):Transaction

This method passes the customer transaction's request to the related transaction.

Contract in Analysis:

```

context: Transaction::chooseTransaction(session:Session, atm:ATM, bank:Bank):Transaction
    post: let option : integer = atm.getMenuChoice ("Please choose a transaction type:", 4, TransactionMenu)
        (option = 1 and result oclIsTypeOf WithdrawalTransaction)
        or(option = 2 and result oclIsTypeOf DepositTransaction)
        or(option = 3 and result oclIsTypeOf TransferTransaction)
        or(option = 4 and result oclIsTypeOf InquiryTransaction)

```

Contract in Implementation:

```

public static Transaction chooseTransaction(Session session, ATM atm, Bank bank)
{

```

```

String transactionMenu[] =
    { "Cash Withdrawl",
      "Deposit",
      "Transfer Funds Between Accounts",
      "Balance Inquiry"
    };

// local variable option is used to support contract
// implementation.
int option = atm.getMenuChoice("Please choose a transaction
                                type:", 4, transactionMenu);
Transaction result; // local variable result is sused to support
                    // contract implementation.

switch (option)
{
    case 1:
        result = new WithdrawlTransaction(session, atm, bank);
        break;
    case 2:
        result = new DepositTransaction(session, atm, bank);
        break;
    case 3:
        result = new TransferTransaction(session, atm, bank);
        break;
    case 4:
        result = new InquiryTransaction(session, atm, bank);
        break;
    default:
        result = null; // To keep the compiler happy
}

/** assertion is used to implement the postcondition
 *
 * @assert (option == 1)&&( result instanceof
 *         WithdrawlTransaction )
 * ||(option == 2)&&( result instanceof depositTransaction )
 * ||(option == 3)&&( result instanceof TransferTransaction )
 * ||(option == 4)&&( result instanceof InquiryTransaction )
 * || result == null
 */
return result;
}

```

Transaction::doTransactionUseCase():int

This method gets the transaction's finished result.

Contract in Analysis:

```

context Transaction:: doTransactionUseCase():int
post:  if self.getTransactionSpecificsFromCustomer() <> Status.Success
      then result = getTransactionSpecificsFromCustomer()
      else
        if self.sendToBank() = Status.INVALID_PIN and
          _session.doInvalidPINExtension() = Status.INVALID_PIN then
          result = _session.doInvalidPINExtension()
        else if self.sendToBank() = Status.INVALID_PIN and
          _session.doInvalidPINExtension() = Status.SUCCESS then
          result = finishApprovedTransaction()
        endif
      endif
endif

```

endif

Contract in Implementation:

```
/**
 * Part of the postcondition in OCL is implemented in the body of the
 * method with assert.
 *
 * @post Status.contains($result)
 */
public int doTransactionUseCase()
{
    int code;
    int result; // local variable codeCheck is used to support Contract.
    code = getTransactionSpecificsFromCustomer(); result = code;
    if (code != Status.SUCCESS)
    {
        result = code;
        /** @assert code != Status.SUCCESS && result == code */
        return result;
    }

    code = sendToBank(); result = code;
    if (code == Status.INVALID_PIN)
    {
        code = _session.doInvalidPINExtension(); result = code;
        if (code == Status.INVALID_PIN)
        {
            result = code;
            /**@assert result == code && code == Status.INVALID_PIN */
            return result;
        }
    }
    if (code == Status.SUCCESS)
    {
        code = finishApprovedTransaction(); result = code;
    }
    return result;
}
```

Transaction::getTransactionSpecificsFromCustomer():int

Abstract method, which will be implemented in the child class

Transaction::sendToBank():int

Abstract method, which will be implemented in the child class

Transaction::finishApproveTransaction():int

Abstract method, which will be implemented in the child class

Class WithdrawlTransaction

Instance Attributes:

_fromAccount : int
_amount : Money

Invariant

Contract in Analysis:

context: WithdrawlTransaction


```
inv: self._bank.verifyAccount(self._fromAccount)
```

Contract in Implementation:

```
/** @invariant _bank.verifyAccount(_fromAccount)
**/
```

WithdrawlTransaction::WithdrawlTransaction(session:Session, atm:ATM, bank:Bank)

This method is the constructor of WithdrawlTransaction class.

WithdrawlTransaction::getTransactionSpecificsFromCustomer():int

This method gets the customer withdrawal requirements like account type, amount etc.

Contract in Analysis:

```
context: WithdrawlTransaction::getTransactionSpecificsFromCustomer():int
post: result = Status.SUCCESS or result = Status.TOO_LITTLE_CASH
post: : let option : integer = atm.getMenuChoice ("Please choose
an amount:", 7)
(option = 1 and result.getValue() == 2000) or
(option = 2 and result.getValue() == 4000) or
(option = 3 and result.getValue() == 6000) or
(option = 4 and result.getValue() == 8000) or
(option = 5 and result.getValue() == 10000) or
(option = 6 and result.getValue() == 20000) or
(option = 7 and result.getValue() == 30000)
```

Contract in Implementation:

```
/** part of the postcondition is implemented in the body of the method
* with assert
*
* @post $result == Status.TOO_LITTLE_CASH || $result == Status.SUCCESS
*
**/
public int getTransactionSpecificsFromCustomer()
{
    _fromAccount = _bank.chooseAccountType("withdraw from", _atm);
    String menu[] =
        { "$ 20", "$ 40", "$ 60", "$ 80", "$ 100", "$ 200", "$ 300" };
    int option = _atm.getMenuChoice("Please choose an amount:", 7,
        menu); //local variable is defined to support the contract.
    switch (option)
    {
        case 1: _amount = new Money(20); break;
        case 2: _amount = new Money(40); break;
        case 3: _amount = new Money(60); break;
        case 4: _amount = new Money(80); break;
        case 5: _amount = new Money(100); break;
        case 6: _amount = new Money(200); break;
        case 7: _amount = new Money(300); break;
    }

    /** @assert (option == 1)&&( _amount.getValue() == 2000.00 ) ||
    *           (option == 2)&&( _amount.getValue() == 4000.00 ) ||
    *           (option == 3)&&( _amount.getValue() == 6000.00 ) ||
```

```

*          (option == 4)&&(_amount.getValue() == 8000.00)||
*          (option == 5)&&(_amount.getValue() == 10000.00)||
*          (option == 6)&&(_amount.getValue() == 20000.00)||
*          (option == 7)&&(_amount.getValue() == 30000.00) */
    if (_atm.checkIfCashAvailable(_amount))
        return Status.SUCCESS; else return Status.TOO_LITTLE_CASH;
}

```

WithdrawlTransaction::sendToBank():int

This method gets the transaction approval from the bank. There is no contract written for this method because this is a delegation method.

WithdrawlTransactin::finishApprovedTransaction():int

This method finishes the withdraw transaction and prints the receipt to customer.

Contract in Analysis:

```

context WithdrawlTransaction::finishApprovedTransaction():int
post: result = Status.SUCCESS

```

Contract in Implementation:

```

/**
 * @post $result == Status.SUCCESS
 *
 */

```

Class DepositTransaction

Instance Attributes:

```

_toAccount : int
_amount : Money

```

Invariant

Contract in Analysis:

```

context: DepositTransaction
inv: self._bank.verifyAccount(_toAccount)

```

Contract in Implementation:

```

/**@invariant _bank.verifyAccount(_toAccount)
 */

```

DepositTransaction::DepositTransaction(session:Session, atm:ATM, bank:Bank)

This method is the constructor of DepositTransaction class.

DepositTransaction::getTransactionSpecificsFromCustomer():int

This method gets the customer deposit information such as the account type and the deposit amount.

Contract in Analysis:

```
context DepositTransaction::getTransactionSpecificsFromCustomer():int
    post: result = Status.SUCCESS
```

Contract in Implementation:

```
/**
 * @post $result == Status.SUCCESS
 *
 */
```

DepositTransaction::sendToBank():int

This method gets the approval from bank for the deposit request. There is no contract

designed for this method because this is a delegation method.

DepositTransaction::finishApprovedTransaction():int

This method finishes the approved deposit transaction.

Contract in Analysis:

```
context DepositTransaction::finishApprovedTransaction():int
    post result = Status.SUCCESS or result =
        Status.ENVELOPE_DEPOSIT_TIMED_OUT
```

Contract in Implementation:

```
/**
 * @pre _newBalance.getValue() >= _amount.getValue()
 * @post $result == Status.SUCCESS || $result ==
 *         Status.ENVELOPE_DEPOSIT_TIMED_OUT
 *
 */
```

Class TransferTransaction

Instance Attributes:

```
_fromAccount : int
_toAccount : int
_amount : Money
```

Invariant

Contract in Analysis:

```

context: TransferTransaction
  inv: self._bank.verifyAccount(self._fromAccount)
  inv: self._bank.verifyAccount(self._toAccount)

```

Contract in Implementation:

```

/**
 * @invariant _bank.verifyAccount(_fromAccount)
 * @invariant _bank.verifyAccount(_toAccount)
 */

```

TransferTransaction::TransferTransaction(session:Session, atm:ATM, bank:Bank)

This method is the constructor of TransferTransaction class.

TransferTransaction::getTransactionSpecificsFromCustomer():

int

This method gets the requirement information from customer like the amount type from and to, transfer amount, etc.

Contract in Analysis:

```

context: TransferTransaction::getTransactionSpecificsFromCustomer():int
  post: result == Status.SUCCESS

```

Contract in Implementation:

```

/**
 *
 * @post $result == Status.SUCCESS
 */

```

TransferTransaction::sendToBank():int

There is no contract designed for this method because it is a delegation method.

TransferTransaction::finishApprovedTransaction():int

This method prints the receipt for customer.

Contract in Analysis:

```

context: TransferTransaction:: finishApprovedTransaction():int
  pre: self._newBalance.getValue() >= _amount.getValue() and
       self._availableBalance.getValue() >= _amount.getValue()

  post: result ==Status.SUCCESS

```

Contract in Implementation:

```

/**
 * @pre _newBalance.getValue() >= _amount.getValue()
 * @pre _availableBalance.getValue() >= _amount.getValue()
 *
 * @post $result == Status.SUCCESS

```

*
* */

Class InquiryTransaction

Instance Attributes:

`_fromAccount : int`

Invariant

Contract in Analysis:

context: InquiryTransaction
inv `self._bank.verifyAccount(self._fromAccount)`

Contract in Implementation:

/** @invariant `_bank.verifyAccount(_fromAccount)`

****/**

InquiryTransaction::TransferTransaction(session:Session, atm:ATM, bank:Bank)

This method is the constructor of InquiryTransaction class.

InquiryTransaction::getTransactionSpecificsFromCustomer():int

This method gets the requirement information from customer like the amount type.

Contract in Analysis:

context: InquiryTransaction::getTransactionSpecificsFromCustomer():int
post: `result = Status.SUCCESS`

Contract in Implementation:

/** @post `$result == Status.SUCCESS`
****/**

InquiryTransaction::sendToBank():int

This method gets the approval from bank and finishes the balance check.

InquiryTransaction::finishApprovedTransaction():int

This method prints the receipt for customer

Contract in Analysis:

context: InquiryTransaction::finishApprovedTransaction():int
post: `result = Status.SUCCESS`

Contract in Implementation:

/** @post `$result == Status.SUCCESS`
****/**

Appendix D: Root Cause of Mutants not Killed by the Contracts

In this section we explain why those mutants are killed by the oracles, but not by the contracts.

Limitation Category: Contract not Precise Enough

Mutant 3 & 4

In mutant 3 and 4 we change the logic operator in the same statement of `ATM` class. This statement is the `WHILE` statement in the `DO-WHILE` loop, and the contract can't be precise enough to catch these two faults.

Mutant 8

In this mutant we change the return value from `PIN` to `_number`, and the postcondition for both of them are the same so that the contracts can't catch this fault.

Mutant 57

The detailed information please refers to Appendix A.

Mutnat 94

In this mutant, we delete one statement, `break`, in the `Session` class `doSessionUseCase ()` method. Then the return value is `FINISHED (1)`, not `ABORTED (0)`. However the contract defined for this method is: `result == FINISHED OR result == ABORTED`, therefore we can't catch this fault with the contracts.

Mutant 100

One statement, `_display.reportCardRetained()`, is deleted in `ATM` class `retainCard()` method. And the contract can't be defined precise enough to catch such kind of fault.

Mutant: 102

In this mutant we change the variable name from `_fromAccount` to `_toAccount`, therefore the `_fromAccount` is not initiated with a value, and only with the default value 0. However in the contract, the attribute `_fromAccount` will be checked in the invariant to see whether it is one of the possible values that are 0,1 or 2. So in this example the contract is also not precise enough to catch the fault.

Mutant: 104

One statement, `_display.cardEjected()`, is deleted in ATM class `ejectCard()` method. And the contract can't be defined precise enough to catch such kind of fault.

Mutant: 106

In this mutant we change the value of the variable `_status` from `CARD_HAS_BEEN_READ` to `UNREADABLE_CARD`, which is just fallen into the range of the post-condition. Therefore we can't catch the mutant.

Mutant 108

This mutant is similar with mutant 106, the return value is just fallen into the range of the postcondition.

Limitation Category: No Computation Involved

Mutant 1

The detailed information please refers to Appendix A.

Limitation Category: Involving Constant Change

Mutant 2

In this mutant we change the value of constant `STOPPED` from 1 to 0, which is equal to the constant `RUNNING`. This mutant is not killed because the constant is also used in the contract definition.

Mutant 45

The detailed information please refers to Appendix A.

Mutant 52

In this mutant we change the value of constant `INVALID_PIN` from 4 to 5 in `Status` class.

There is no contract broken because we also use `INVALID_PIN` in the contract definition.

Limitation Category: Exception Raised

Mutant 23 & 24

The same situation is happened in mutant 23 and 24. We change the logic operator from `||` to `&&` in the card number identification operation of method `bank.initiateWithdrawl(...)` (mutant 23) and `bank.doInquiry(...)` (mutant24). When we feed the system with the incorrect card number, exception is raised as the following,

Mutant 23

```
java.lang.ArrayIndexOutOfBoundsException
    at Bank.initiateWithdrawl(Bank.java)
    at WithdrawlTransaction.sendToBank(Transaction.java)
    at Transaction.doTransactionUseCase(Transaction.java)
    at Session.doSessionUseCase(Session.java)
    at ATM.serviceCustomers(ATM.java)
    at ATMMain.run(ATMMain.java)
    at java.lang.Thread.run(Unknown Source)
```

Mutant 24

```
java.lang.ArrayIndexOutOfBoundsException
    at Bank.doInquiry(Bank.java)
    at InquiryTransaction.sendToBank(Transaction.java)
    at Transaction.doTransactionUseCase(Transaction.java)
    at Session.doSessionUseCase(Session.java)
    at ATM.serviceCustomers(ATM.java)
    at ATMMain.run(ATMMain.java)
    at java.lang.Thread.run(Unknown Source)
```

Therefore the ATM system is stopped and the contract can't be executed further more either.

Mutant 58

In mutant 58 we change the CASE value in SWITCH-CASE control block, then the system will return a null object according to the specified test case. When this null object is used in the following execution the exception is reported and the system is terminated.

```
java.lang.NullPointerException
    at Session.doSessionUseCase(Session.java)
    at ATM.serviceCustomers(ATM.java)
    at ATMMain.run(ATMMain.java)
    at java.lang.Thread.run(Unknown Source)
```

Mutant 59

The detailed information please refers to Appendix A

Mutant 80

When we add a new Session object, which has defined in the Transaction class, in the TransferTransaction class inheritted from Transaction class, the contracts didn't detect this fault. The reason is the contracts are designed according to the existed attributes and methods defined in the specification. If the parents' attribute is defined again in the child class and there is no any invariant, precondition or postcondition designed to check it, such kind of fault will not be caught by the contracts. However the run time exception will be raised cause this object is not initiated.

The run time exception is:

```
java.lang.NullPointerException
    at TransferTransaction.sendToBank(Transaction.java)
    at Transaction.doTransactionUseCase(Transaction.java)
    at Session.doSessionUseCase(Session.java)
    at ATM.serviceCustomers(ATM.java)
    at ATMMain.run(ATMMain.java)
    at java.lang.Thread.run(Unknown Source)
```

Limitation Category: Message Displayed to the User

Mutant 49

The detailed information please refers to Appendix A.

Appendix E: Test Cases Designed for ATM System

In this section the test cases designed for ATM System according to Category-Partition testing strategy.

Use Case 001: System Startup

The system is started up with greeting

* Welcome to ATM 42 at GORDON COLLEGE *

* Please input Operator Password to turn me on *

When the operator inputs the correct password (123456) and turns the ATM machine on the message [1.1] is displayed

[1.1] How Many \$20 Bills in the Cash Dispenser?

When the password is incorrect, the message [1.2] is displayed.

[1.2] Reenter Operator Password.

Only when the operator input the correct password, then the message [1.1] is displayed.

When the password is correct, the operator will be asked to enter the amount of money currently in the cash dispenser. If operator didn't input any number or 0 the message [1.1] will be repeated, otherwise a new customer session can start with message [1.3].

[1.3] Please Insert Your Card to Begin, or input Operator Password to Turn Me off

When one session is finished the message will be displayed as message [1.3]

When the operator password is inputted the ATM machine is turned off, otherwise a new customer session starts.

Table 8 Parameter, Category and Choices for Use Case 001

Parameter	Category	Choice
Operator Password for Machine On	Correct	123456

Parameter	Category	Choice
Initial Cash Amount	Incorrect	222222
	Correct Amount	1, 20
	Incorrect Amount	0
Operator Password for Machine Off	Correct	123456
	Incorrect	333333 or return

Table 9 Test Cases for Test Suite 001

TS001-TC001		
Input	Process	Output
After the ATM machine is turned on, the operator inputs the correct operator password: 123456 Then input the correct initial cash amount: 1	ATM machine is turned on and sets up the initial amount	<i>How Many \$20 Bills in the Cash Dispenser?</i> <i>Please Insert Your Card to Begin, or Input Operator Password to turn me off</i>
TS001-TC002		
Input	Process	Output
After the ATM machine is turned on, the operator inputs the correct operator password: 123456 Then input the correct initial cash amount: 20	ATM machine is turned on and sets up the initial amount	<i>How Many \$20 Bills in the Cash Dispenser?</i> <i>Please Insert Your Card to Begin, or Input Operator Password to turn me off</i>
TS001-TC003		
Input	Process	Output
After the ATM machine is turned on, the operator inputs the incorrect operator password: 222222	ATM machine doesn't accept the incorrect number and ask the operator to input the number again.	<i>Reenter the Operator Password:</i>
TS001-TC004		
Input	Process	Output
After the ATM machine is turned on, input the correct operator password: 123456 Then input the incorrect initial cash amount: 0	ATM machine is turned on and needs the operator to set the initial cash again.	<i>How Many \$20 Bills in the Cash Dispenser?</i> 0 or return <i>How Many &20 Bills in the Cash Dispenser</i>
TS001-TC005		
Input	Process	Output
When one customer service session is finished, the operator inputs the correct password to turn off the machine: 123456	ATM receives the correct password and is turned off.	<i>Please Insert Your Card to Begin, or Input Operator Password to turn me off</i>
TS001-TC006		
Input	Process	Output
When one customer session is finished the operator input the incorrect password: 222222	ATM receives the incorrect password and starts the new customer session	Start the new customer session.

Use Case 002: Session

A session is started when a customer inserts an ATM card into the card reader slot of the machine, which reads the card as it is inserted (message [1.3]).

[1.3] Please Insert Your Card to Begin, or Input Operator Password to Turn Me Off

After customer inputs the card number he is asked to enter his/her PIN (message [2.1]), and is allowed to perform one or more transactions, choosing the transaction type each time from a menu of options.

[2.1] Please Input PIN Number

After customer input the PIN number the session will initiate the appropriate transaction use case (message [2.2]), and will furnish the customer's card number and PIN to the transaction.

[2.2] Please choose a transaction type:

- 1) Cash Withdraw*
- 2) Deposit*
- 3) Transfer Funds Between Accounts*
- 4) Balance Inquiry*

After each successful transaction, the customer is asked whether he/she would like to perform another (message [2.3]).

[2.3] Do you want to perform another transaction?

- 1) Yes*
- 2) No*

If yes the Transaction Menu (message [2.2]) will be displayed again, if no the ATM machine will display message [2.4] and message [1.3]:

[2.4] Your Card is ejected

[1.3] Please Insert Your Card to Begin, or Input Operator Password to Turn Me Off

Table 10 Parameter, Category and Choices for Use Case 002

Parameter	Category	Choice
Card number	Correct	2
	Incorrect	3
PIN number	Correct	1234
	Incorrect	2222
Transaction Chosen	Within Menu	1,2,3,4
	Out of Menu	5
Another Transaction Chosen	Within Menu	1,2
	Out of Menu	3

Note: The incorrect Card number and incorrect PIN number will not be tested in this test suite.

Table 11 Test Cases for Test Suite 002

TS002-TC001		
Input	Process	Output
The customer inputs the correct card number and correct PIN number. Card number = 2 PIN number = 1234	ATM machine saves the card number and PIN number, and displays the transaction menu.	Transaction Menu
TS002-TC002		
Input	Process	Output
When the transaction menu is displayed, customer chooses one option to proceed the transaction: Option = 1	ATM machine displays the related information	Display the account information
TS002-TC003		
Input	Process	Output
When the transaction menu is displayed, customer chooses one option to proceed the transaction: Option = 2	ATM machine displays the related information	Display the account information
TS002-TC004		
Input	Process	Output
When the transaction menu is displayed, customer chooses one option to proceed the transaction: Option = 3	ATM machine displays the related information	Display the account information
TS002-TC005		
Input	Process	Output
When the transaction menu is displayed, customer chooses one option to proceed the transaction: Option = 4	ATM machine displays the related information	Display the account information
TS002-TC006		

Input	Process	Output
When the transaction menu is displayed, customer chooses one option to proceed the transaction: Option = 5	ATM machine displays the related information	The customer needs to input the option number again.
TS002-TC007		
Input	Process	Output
When one transaction is finished, customer needs to choose whether he want to proceed another transaction: Option = 1	ATM will display the transaction menu again.	Transaction Menu
TS002-TC008		
Input	Process	Output
When one transaction is finished, customer needs to choose whether he wants to proceed another transaction: Option = 2	ATM will quit to read card status.	Display <i>Your Card is Ejected Please Insert Your Card to Begin or Input Operator Password to Turn Me Off</i>
TS002-TC009		
Input	Process	Output
When one transaction is finished, customer needs to choose whether he wants to proceed another transaction: Option = 3	ATM will not display any message.	The customer needs to input the option number again.

Use Case 003: Cash Withdraw Transaction Use Case

A cash withdraw transaction is started from within a session when the customer chooses cash withdraw from the menu of possible transaction types. The customer chooses a type of account to withdraw from (e.g. checking) from a menu of the possible accounts (message [3.1]), and then chooses a dollar amount from a menu of the possible amounts (message [3.2]).

[3.1] Please choose an account to withdraw from

- 1) *Checking*
- 2) *Savings*
- 3) *Money Market*

[3.2] Please choose an amount:

1) \$ 20

2) \$ 40

3) \$ 60

4) \$ 80

5) \$ 100

6) \$ 200

7) \$ 300

The system verifies that it has sufficient money on hand to satisfy the request. If not, it reports a failure to the session, which initiates the Failed Transaction Extension to report the problem with the message [3.3].

[3.3] Sorry, there is not enough cash available to satisfy your request

If there is sufficient cash, it sends the customer's card number, PIN, chosen account and amount to the bank, which either approves or disapproves the transaction. If the transaction is approved, the machine dispenses the correct amount of cash and issues a receipt. If the transaction is disapproved due to an incorrect PIN, the Incorrect PIN extension is executed.

If the account that the customer wants to operate doesn't exist, the no such kind of account error message [3.4] will be reported to the customer.

[3.4] No account of the type you requested is link to your card.

If the withdrawal amount is more than the available balance of the operated account, the not enough balance error message [3.5] will be reported to the customer.

[3.5] The available balance in your account is not sufficient

If the customer total withdrawal amount within one day is more than \$300, the Daily Withdrawal Limit Exceed error message [3.6] will be reported to the customer.

[3.6] You have exceeded the daily limit on cash withdrawal with your card

All other disapprovals are reported to the session, which initiates the Failed Transaction Extension. The bank is notified whether or not an approved transaction was completed in its entirety by the machine; if it is completed then the bank completes debiting the customer's account for the amount.

Table 12 Parameter, Category and Choices for Use Case 003

Parameter	Category	Choice
Withdraw Amount	Amount more than available cash amount in ATM machine	300
	Amount more than account available amount	200,
	Amount within all the limitation	20, 100
	Accumulated Withdrawal Amount more than Daily Limitation	400
Account	Account available	Checking, Money Market
	Account not available	Saving

Default parameters:

- ❖ Initial Cash Amount: 200
- ❖ Available amount in the account: 100 (checking), 5000(money market)

Table 13 Test Cases for Test Suite 003

TS003-TC001		
Input	Process	Output
After inputting the correct card number and PIN number, the customer chooses the withdraw transaction with the valid account the amount within the available balance. Card number = 2 PIN = 1234 Account: checking Amount: 20	ATM machine processes the correct transaction and issues a receipt	Receipt is printed with this format: RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 WITHDRAWL FROM CHKNG AMOUNT: 20.00 CURR BAL: 80.00 AVAILABLE: 80.00
TS003-TC002		
Input	Process	Output
After inputting the correct card number and PIN number, the	ATM machine processes the correct transaction and	Receipt is printed with this format:

customer chooses the withdraw transaction with the valid account the amount within the available balance. Card number = 2 PIN = 1234 Account: money market Amount: 20	issues a receipt	RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 WITHDRAWL FROM MNYSMKT AMOUNT: 20.00 CURR BAL: 4980.00 AVAILABLE: 4980.00
TS003-TC003		
Input	Process	Output
After inputting the correct card number and PIN number, the customer chooses the withdraw transaction with the invalid account the amount within the available balance. Card number = 2 PIN = 1234 Account: saving Amount: 20	ATM machine gives a error message and asks the customer whether he wants another transaction.	Error message: <i>No account of the type you requested is link to your card. Do you want to perform another transaction? (1=Yes, 2=No)</i>
TS003-TC004		
Input	Process	Output
After inputting the correct card number and PIN number, the customer chooses the withdraw transaction with the valid account the amount within the available balance, but out of the ATM machine cash limit amount: Initial = 40 \$ Card number = 2 PIN = 1234 Account: checking Amount: 60	ATM machine gives an error message and asks the customer whether he wants another transaction.	Error message will be displayed: <i>Sorry, there is not enough cash available to satisfy your request Do you want to perform another transaction? (1 = Yes, 2 = No)</i>
TS003-TC005		
Input	Process	Output
After inputting the correct card number and PIN number, the customer chooses the withdraw transaction with the valid account the amount that is more than the available balance. Card number = 2 PIN = 1234 Account: checking Amount: 200	ATM machine reports the error message and asks the customer whether he wants another transaction.	Error message: <i>The available balance in your account is not sufficient Do you want to perform another transaction? (1 = Yes, 2 = No)</i>
TS003-TC006		
Input	Process	Output
After inputting the correct card number and PIN number, the customer chooses the withdraw transaction with the valid account the amount within the available balance. Card number = 2	ATM machine processes the correct transaction and issues a receipt	Receipt is printed with this format: RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 WITHDRAWL FROM CHKNG AMOUNT: 100.00

PIN = 1234 Account: checking Amount: 100		CURR BAL: 0.00 AVAILABLE: 0.00
TS003-TC007		
Input	Process	Output
<p>After inputting the correct card number and PIN number, the customer chooses the withdraw transaction with the valid account the amount that is more than the daily withdrawl limited amount.</p> <p>Initial Cash = 600 Card number = 2 PIN = 1234 Account = checking Amount: 100</p> <p>Withdrawal again: Account = Money Market Amount: 300</p>	<p>ATM machine processes the two withdrawal operations and issues the receipt or displays the error message.</p>	<p>Receipt is printed with this format: RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 WITHDRAWL FROM CHKNG AMOUNT: 100.00 CURR BAL: 0.00 AVAILABLE: 0.00</p> <p>Error message is:</p> <p><i>You have exceeded the daily limit on cash withdrawal with your card</i></p> <p><i>Do you want to perform another transaction (1 = Yes, 2 = No)?</i></p>

Use Case 004: Deposit Transaction Use Case

A deposit transaction is started from within a session when the customer chooses deposit from the menu of possible transaction types. The customer chooses a type of account to deposit to (e.g. checking) from a menu of possible accounts (message [4.1]), and then input a dollar amount by typing it on the keyboard with the message [4.2]. The system sends the customer's card number, PIN, chosen account and amount to the bank, which either approves or disapproves the transaction.

[4.1] Please choose an account to deposit to

1) Checking

2) Savings

3) Money Market

[4.2] Please enter amount

Press ENTER when finished or CLEAR to start over

If the transaction is approved, the machine accepts an envelope from the customer containing cash and/or checks and then issues a receipt (message [4.3]).

[4.3] Please deposit an envelope in the slot by typing insert

If the customer input “ insert” the envelope is inserted, otherwise the envelope is not inserted and the system quit (message [4.4])

[4.4] Envelope not deposited - transaction cancelled

If the transaction is disapproved due to an incorrect PIN, the Incorrect PIN extension is executed.

All other disapprovals are reported to the session, which initiates the Failed Transaction Extension.

The bank is notified whether or not an approved transaction was completed in its entirety by the machine; if it is completed then the bank completes crediting the customer's account for the amount - contingent on manual verification of the deposit envelope contents by an operator later.

Table 14 Parameter, Category and Choices for Use Case 004

Parameter	Category	Choice
Deposit Amount	Any amount	20, 20.45
Account	Available	Checking, money market
	Not Available	Saving
Envelop Insert	Insert the envelope	Type “insert”
	Not insert the envelope	Type “ ”

Default Parameter:

- ❖ Initial cash set: 400

Table 15 Test Cases for Test Suite 004

TS004-TC001		
Input	Process	Output
After inputting the correct card number and PIN number, the customer chooses deposit transaction with the valid account	ATM machine processes this correct deposit transaction and issues a receipt.	Issue the receipt with this format and display the message that whether the customer want to do another transaction or not.

the deposit amount. Card Number = 2 PIN = 1234 Account = checking Amount = 20 And insert the envelop right now with typing "insert"		RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 DEPOSIT TO CHKNG AMOUNT: 20.00 CURR BAL: 120.00 AVAILABLE: 100.00
TS004-TC002		
Input	Process	Output
After inputing the correct card number and PIN number, the customer chooses deposit transaction with the valid account the deposit amount. Card Number = 2 PIN = 1234 Account = checking Amount = 20.45 And insert the envelope.	ATM machine processes this correct deposit transaction and issues a receipt.	Issue the receipt with this format and display the message whether the customer wants to do another transaction. RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 DEPOSIT TO MNYMKT AMOUNT: 20.45 CURR BAL: 120.45 AVAILABLE: 100.00
TS004-TC003		
Input	Process	Output
After inputing the correct card number and PIN number, the customer chooses deposit transaction with the valid account the deposit amount. Card Number = 2 PIN = 1234 Account = money market Amount = 20 And insert the envelop	ATM machine processes this correct deposit transaction and issues a receipt.	Issue the receipt with this format and display the message whether the customer wants to do another transaction. RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 DEPOSIT TO MNYMKT AMOUNT: 20.00 CURR BAL: 5020.00 AVAILABLE: 5000.00
TS004-TC004		
Input	Process	Output
After inputing the correct card number and PIN number, the customer chooses deposit transaction with the invalid account the deposit amount. Card Number = 2 PIN = 1234 Account = savings Amount = 20	ATM machine displays the error message and asks the customer weather he wants another transaction	Error Message: <i>No account of the type you requested is link to your card. Do you want to perform another transaction? (1=Yes, 2=No)</i>
TS004-TC005		
Input	Process	Output
After inputing the correct card number and PIN number, the customer chooses deposit transaction with the valid account	ATM machine aborts this deposit transaction and asks the customer whether he wants another transaction.	Message will be displayed: <i>Envelope not deposited - transaction cancelled.</i>

the deposit amount. But he forgot inserting the envelope. Card Number = 2 PIN = 1234 Account = checking Amount = 20 Not inserting the envelope		Do you want to perform another transaction? (1=Yes, 2=No)
--	--	---

Use Case 005: Transfer Transaction Use Case

A transfer transaction is started from within a session when the customer chooses transfer from the menu of possible transaction types.

The customer chooses a type of account to transfer from (e.g. checking) from a menu of possible accounts (message [5.1]), chooses a different account to transfer to (message [5.2]), and then chooses a dollar amount by typing it (message [5.3]). The system sends the customer's card number, PIN, chosen account and amounts to the bank, which either approves or disapproves the transaction.

[5.1] Please choose an account transfer from

1) Checking

2) Savings

3) Money Market

[5.2] Please choose an account transfer to

1) Checking

2) Savings

3) Money Market

[5.3] Please enter amount

Press ENTER when finished or CLEAR to start over

If the transaction is approved, the machine issues a receipt.

If the transaction is disapproved due to an incorrect PIN, the Incorrect PIN extension is executed.

All other disapprovals are reported to the session, which initiates the Failed Transaction Extension. The bank will consider the transaction complete once it has been approved.

Table 16 Parameter, Category and Choices for Use Case 005

Parameter	Category	Choice
From account	Available	Checking, Money Market
	Not Available	Savings
To account	Available	Checking, Money Market
	Not available	Savings
Transfer Amount	Within the available balance	100, 50
	Out of the available	101,5001

Default Parameter:

- ❖ Initial cash set: 400
- ❖ Available Balance: checking (100), Money Market (5000)

Table 17 Test Cases for Test Suite 005

TS005-TC001		
Input	Process	Output
<p>After inputting the correct card number and the correct PIN number, the customer chooses the transfer option with the valid from account, valid to account and the transfer amount within the available balance.</p> <p>Card Number = 2 PIN = 1234 From account = checking To account = money market Amount = 100</p>	<p>ATM machine processes this correct transfer transaction, issues a receipt and asks the customer whether he wants another transaction</p>	<p>Receipt is issued:</p> <p>RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 TRANSFER CHKNG TO MNYMKT AMOUNT: 100.00 CURR BAL: 5100.00 AVAILABLE: 5100.00</p> <p>Current balance and available balance will be deducted at the same time.</p>
TS005-TC002		
Input	Process	Output
<p>After inputting the correct card number and the correct PIN number, the customer chooses the transfer option with the valid from account, valid to account and the transfer amount within the available balance.</p> <p>Card Number = 2</p>	<p>ATM machine processes this correct transfer transaction, issues a receipt and asks the customer whether he wants another transaction</p>	<p>Receipt is issued:</p> <p>RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 10 TRANSFER CHKNG TO MNYMKT AMOUNT: 5050.00 CURR BAL: 5050.00</p>

PIN = 1234 From account = checking To account = money market Amount = 50		AVAILABLE: 50.00 Current balance and available balance will be deducted at the same time.
TS005-TC003		
Input	Process	Output
After inputing the correct card number and the correct PIN number, the customer chooses the transfer option with the valid from account, valid to account and the transfer amount that is more than the available balance. Card Number = 2 PIN = 1234 From account = checking To account = money market Amount = 101	ATM machine reports the error message and asks the customer whether he wants another transaction.	Error message is displayed as: <i>The available balance in your account is not sufficient</i> <i>Do you want to perform another transaction?</i> <i>(1=Yes, 2=No)</i>
TS005-TC004		
Input	Process	Output
After inputing the correct card number and the correct PIN number, the customer chooses the transfer option with the invalid from account, valid to account and the transfer amount within the available balance. Card Number = 2 PIN = 1234 From account = savings To account = money market Amount = 50	ATM machine reports the error message and asks the customer whether he wants another transaction.	Error message: <i>No account of the type you requested is link to your card.</i> <i>Do you want to perform another transaction?</i> <i>(1=Yes, 2=No)</i>
TS005-TC005		
Input	Process	Output
After inputing the correct card number and the correct PIN number, the customer chooses the transfer option with the valid from account, invalid to account and the transfer amount within the available balance. Card Number = 2 PIN = 1234 From account = checking To account = savings Amount = 50	ATM machine reports the error message and asks the customer whether he wants another transaction.	Error message: <i>No account of the type you requested is link to your card.</i> <i>Do you want to perform another transaction?</i> <i>(1=Yes, 2=No)</i>
TS005-TC006		
Input	Process	Output
After inputing the correct card number and the correct PIN number, the customer chooses the	ATM machine reports the error message and asks the customer whether he wants	Error message is displayed as: <i>The available balance in your</i>

transfer option with the valid from account, valid to account and the transfer amount that is more than the available balance. Card Number = 2 PIN = 1234 From account = money market To account = checking Amount = 5001	another transaction.	<i>account is not sufficient</i> <i>Do you want to perform another transaction?</i> <i>(1=Yes, 2=No)</i>
---	----------------------	---

Use Case 006: Balance Inquiry Transaction Use Case

A balance inquiry transaction is started from within a session when the customer chooses balance inquiry from the menu of possible transaction types.

The customer chooses a type of account to inquire about (Message [6.1]). The system sends the customer's card number, PIN and chosen account to the bank, which either approves or disapproves the transaction. If the transaction is approved the bank sends the balance to the machine with the approval, and the system prints it on a receipt.

[6.1] Please choose an account you want to inquiry

1) Checking

2) Savings

3) Money Market

If the transaction is disapproved due to an incorrect PIN, the Incorrect PIN extension is executed.

All other disapprovals are reported to the session, which initiates the Failed Transaction Extension.

Table 18 Parameter, Category and Choices for Use Case 006

Parameter	Category	Choice
Inquiry account	Available	Checking, money market
	Not Available	Saving

❖ Initial Cash = 400

Table 19 Test Cases for Test Suite 006

TS006-TC001		
Input	Process	Output
<p>After inputting the correct card number and the correct PIN number, the customer chooses the inquiry option from one account.</p> <p>Card Number = 2 PIN = 1234 Inquiry account = checking</p>	<p>ATM machine processes the correct transaction, issues a receipt and asks the customer whether he wants to do another transaction</p>	<p>Issue receipt as below:</p> <p>RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 3 INQUIRY FROM CHKNG</p> <p>CURR BAL: 100.00 AVAILABLE: 100.00</p> <p><i>Do you want to perform another transaction? (1=Yes, 2=No)</i></p>
TS006-TC002		
Input	Process	Output
<p>After inputting the correct card number and the correct PIN number, the customer chooses the inquiry option from one account.</p> <p>Card Number = 2 PIN = 1234 Inquiry account = money market</p>	<p>ATM machine processes the correct transaction, issues a receipt and asks the customer whether he wants to do another transaction.</p>	<p>Issue receipt as below:</p> <p>RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 INQUIRY FROM MNY MKT</p> <p>CURR BAL: 5000.00 AVAILABLE: 5000.00</p> <p><i>Do you want to perform another transaction? (1=Yes, 2=No)</i></p>
TS006-TC003		
Input	Process	Output
<p>After inputting the correct card number and the correct PIN number, the customer chooses the inquiry option from one account.</p> <p>Card Number = 2 PIN = 1234 Inquiry account = saving</p>	<p>ATM machine reports the error message and asks the customer whether he wants to do another transaction.</p>	<p>Error message:</p> <p><i>No account of the type you requested is link to your card.</i></p> <p><i>Do you want to perform another transaction? (1=Yes, 2=No)</i></p>

Use Case 007: Invalid PIN Extension

An invalid PIN extension is started from within a transaction when the bank reports that the customer's transaction is disapproved due to an invalid PIN (Message [7.1]).

[7.1] Your PIN was entered incorrectly

Please re-enter it

The customer is required to re-enter the PIN (message [7.1]) and the original request is sent to the bank again. If the bank now approves the transaction, or disapproves it for some other reasons, the original case is continued; otherwise the process of re-entering the PIN is repeated ([7.1]).

Once the PIN is successfully re-entered, it is used for both the current transaction and all subsequent transactions in the session.

If the customer fails three times to enter the correct PIN, the card is permanently retained, a screen is displayed informing the customer of this and suggesting he/she contact the bank (message [7.2]), and the entire customer session is aborted.

[7.2] Your PIN was entered incorrectly.

Your card has been retained - please contact the bank

Table 20 Parameter, Category and Choices for Use Case 007

Parameter	Category	Choice
PIN number	Correct	1234
	Incorrect	2222
PIN input times	Correct PIN within 3 times	Second time, third time
	Incorrect PIN more than 3 times	Forth time
Operation	Correct	Withdraw, deposit, transfer, inquiry

❖ Initial Cash = 400

Table 21 Test Cases for Test Suite 007

TS007-TC001		
Input	Process	Output
<p>The customer inputs the correct card number, the incorrect PIN number and one transaction.</p> <p>Card Number = 2 PIN = 2222 Transaction = withdraw Account = checking Amount = 10</p> <p>Then customer input the correct PIN at the first time PIN = 1234</p>	<p>ATM machine reports the error message and lets the customer input the PIN number again. When the customer inputs the correct PIN number, the transaction will be preceded.</p>	<p>Error message:</p> <p><i>Your PIN was entered incorrectly Please re-enter it.</i></p> <p>After correct PIN input, continue.</p>

TS007-TC002		
Input	Process	Output
<p>The customer inputs the correct card number, the incorrect PIN number and one transaction.</p> <p>Card Number = 2 PIN = 2222 Transaction = withdraw Account = checking Amount = 10</p> <p>Then customer inputs the correct PIN number at the second time</p> <p>PIN = 1234</p>	<p>ATM machine reports the error message and lets the customer to input the PIN number again. When the customer inputs the correct PIN number at the second time, the transaction will be proceeded.</p>	<p>Error message:</p> <p><i>Your PIN was entered incorrectly Please re-enter it.</i></p> <p>After correct PIN input, continue.</p>
TS007-TC003		
Input	Process	Output
<p>The customer inputs the correct card number, the incorrect PIN number and one transaction.</p> <p>Card Number = 2 PIN = 2222 Transaction = withdraw Account = checking Amount = 10</p> <p>And the customer inputs the incorrect PIN number at the forth time.</p>	<p>ATM machine reports the error message and notifies the customer that his/her card is retained.</p>	<p>Error message:</p> <p><i>Your PIN was entered incorrectly Your card has been retained – please contact the bank.</i></p> <p>This customer service session is finished.</p>
TS007-TC004		
Input	Process	Output
<p>The customer inputs the correct card number, the incorrect PIN number and one transaction.</p> <p>Card Number = 2 PIN = 2222 Transaction = deposit Account = checking Amount = 10</p> <p>Then customer input the correct PIN at the second time</p> <p>PIN = 1234</p>	<p>ATM machine reports the error message and lets the customer to input the PIN number again. When the customer inputs the correct PIN number, the transaction will be proceeded.</p>	<p>Error message:</p> <p><i>Your PIN was entered incorrectly Please re-enter it.</i></p> <p>After correct PIN input, continue.</p>
TS007-TC005		
Input	Process	Output
<p>The customer inputs the correct card number, the incorrect PIN number and one transaction.</p> <p>Card Number = 2 PIN = 2222</p>	<p>ATM machine reports the error message and lets the customer to input the PIN number again. When the customer inputs the correct PIN number, the transaction</p>	<p>Error message:</p> <p><i>Your PIN was entered incorrectly Please re-enter it.</i></p> <p>After correct PIN input, continue.</p>

Transaction = transfer From Account = checking To Account = money market Amount = 10 Then customer input the correct PIN at the second time PIN = 1234	will be proceeded.	
TS007-TC006		
Input	Process	Output
The customer inputs the correct card number, the incorrect PIN number and one transaction. Card Number = 2 PIN = 2222 Transaction = Inquiry From Account = checking Then customer inputs the correct PIN at the second time PIN = 1234	ATM machine reports the error message and lets the customer to input the PIN number again. When the customer inputs the correct PIN number, the transaction will be proceeded.	Error message: <i>Your PIN was entered incorrectly Please re-enter it.</i> After correct PIN input, continue.

The correct PIN number inputed at the first time is not listed in this test suite because it has been tested in test suite 003 – 006.

Use Case 008: Failed Transaction Extension

A failed transaction extension is started from a session when a transaction use case fails to complete successfully for some reason other than repeated entries of an invalid PIN.

A screen is displayed informing the customer of the problem, and the customer is asked whether he/she wants to do another transaction.

Table 22 Parameter, Category and Choices for Use Case 008

Parameter	Category	Choice
Failed Transaction Reason	Correct	1. Invalid Card 2. Envelop insert time out 3. No such account 4. Insufficient available balance 5. Daily withdraw limit exceed 6. Dispenser didn't have enough cash

There are six possible error messages could be reported to the customer, and we have already tested five of them in other test cases. Therefore in this test suite we only test the first error situation, Invalid Card.

Default parameters:

❖ Initial Cash = 400

Table 23 Test Cases for Test Suite 008

TS008-TC001		
Input	Process	Output
Customer inputs the invalid card number, the correct PIN, and one transaction. Card Number = 3 PIN = 1234 Transaction = withdraw Account = checking Amount = 10	ATM machine reports the error message and lets the customer to choose another transaction.	Error message: <i>Your card number was not recognized</i> <i>Do you want to perform another transaction?</i> <i>(1 = Yes, 2 = No)</i>
TS008-TC002		
Input	Process	Output
Customer inputs the invalid card number, the correct PIN, and one transaction. Card Number = 3 PIN = 1234 Transaction = deposit Account = checking Amount = 10	ATM machine reports the error message and lets the customer to choose another transaction.	Error message: <i>Your card number was not recognized</i> <i>Do you want to perform another transaction?</i> <i>(1 = Yes, 2 = No)</i>
TS008-TC003		
Input	Process	Output
Customer inputs the invalid card number, the correct PIN, and one transaction. Card Number = 3 PIN = 1234 Transaction = transfer From Account = checking To Account = money market Amount = 10	ATM machine reports the error message and lets the customer to choose another transaction.	Error message: <i>Your card number was not recognized</i> <i>Do you want to perform another transaction?</i> <i>(1 = Yes, 2 = No)</i>
TS008-TC004		
Input	Process	Output
Customer inputs the invalid card number, the correct PIN, and one transaction. Card Number = 3 PIN = 1234	ATM machine reports error message and lets the customer to choose another transaction	Error message: <i>Your card number was not recognized</i> <i>Do you want to perform another transaction?</i>

Transaction = inquiry Account = checking		(1 = Yes, 2 = No)
---	--	-------------------

Use Case 009: Combination of Transaction

In this section, different transactions are executed within one customer service session.

Table 24 Test Cases for Test Suite 009

TS009-TC001		
Input	Process	Output
<ol style="list-style-type: none"> 1. Set initial amount: 400 2. Card number = 2, PIN = 1234, 3. Choose deposit transaction, deposit 120 to checking account 4. Choose withdraw transaction, withdraw 200 5. Choose Inquiry transaction for checking account 	ATM machine process the transaction and issue receipt	<p>Receipt is printed with this format for deposit transaction:</p> <p>RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 DEPOSIT FOR CHKNG AMOUNT: 120.00 CURR BAL: 220.00 AVAILABLE: 100.00</p> <p>Then when the withdraw operation is chosen, the error message is display:</p> <p><i>The available balance in your account is not sufficient</i></p> <p><i>Do you want to perform another transaction?</i> (1 = Yes, 2 = No)</p> <p>Receipt for Inquiry: RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 2 INQURY FOR CHKNG CURR BAL: 220.00 AVAILABLE: 100.00</p>
TS009-TC002		
Input	Process	Output
<ol style="list-style-type: none"> 1. Set initial amount: 200 2. Card number = 2, PIN = 1234 3. Withdraw 20. 4. Withdraw 20 again. 5. Inquiry for checking account 	ATM machine processes the transaction and issues receipt	<p>The three receipts are:</p> <p>RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 WITHDRAW FROM CHKNG AMOUNT: 20.00 CURR BAL: 80.00 AVAILABLE: 80.00</p> <p>RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 2 WITHDRAW FROM CHKNG</p>

		<p>AMOUNT: 20.00 CURR BAL: 60.00 AVAILABLE: 60.00</p> <p>RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 3 INQURY FROM CHKNG</p> <p>CURR BAL: 60.00 AVAILABLE: 60.00</p>
TS009-TC003		
Input	Process	Output
<p>1. Deposit: Card Number = 2 PIN = 1234 To account = checking Amount = 20</p> <p>2. Transfer From account = checking To account = money market Amount = 80</p> <p>3. Inquiry for checking</p>	<p>ATM machine processes this correct transfer, issues a receipt and lets the customer choose whether he need the next transaction.</p>	<p>Receipt is issued:</p> <p>RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 DEPOSIT TO CHKNG AMOUNT: 100.00 CURR BAL: 120.00 AVAILABLE: 100.00</p> <p>RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 2 TRANSFER CHKNG TO MNYMKT AMOUNT: 80.00 CURR BAL: 5080.00 AVAILABLE: 5080.00</p> <p>RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 3 INQURY FROM MNYMKT</p> <p>CURR BAL: 40.00 AVAILABLE: 20.00</p>
TS009-TC004		
Input	Process	Output
<p>1. Deposit: Card Number = 2 PIN = 1234 To account = checking Amount = 100</p> <p>2. Transfer From account = checking To account = money market Amount = 120</p> <p>3. Inquiry for Money Market Account</p>	<p>ATM machine processes this correct transfer, issues a receipt and lets the customer to choose whether he need another transaction</p>	<p>Receipt is issued:</p> <p>RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 DEPOSIT TO CHKNG AMOUNT: 100.00 CURR BAL: 200.00 AVAILABLE: 100.00</p> <p><i>The available balance in your account is not sufficient</i></p> <p><i>Do you want to perform another transaction?</i></p>

		<i>(1 = Yes, 2 = No)</i> RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 2 INQURY FROM MNYMKT CURR BAL: 5100.00 AVAILABLE: 5100.00
TS009-TC005		
Input	Process	Output
1. Set initial amount: 40 2. Card number = 2, PIN = 1234, 3. Choose withdraw transaction, withdraw 20 from checking account 4. Choose withdraw transaction, withdraw 40 from checking account	Process the withdrawal transaction and report the related receipt and error message	RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 WITHDRAW FROM CHKNG AMOUNT: 20.00 CURR BAL: 80.00 AVAILABLE: 80.00 <i>Sorry, there is not enough cash available to satisfy your request</i> <i>Do you want to perform another transaction?</i> <i>(1 = Yes, 2 = No)</i>
TS009-TC006		
Input	Process	Output
1. Set initial amount: 40 2. Card number =2, PIN 1234 3. Choose withdraw transaction, withdraw 20 from checking account 4. Choose withdraw transaction, withdraw 20 from checking account 5. Quit from the ATM machine	Process the withdraw transaction and report the related receipt	RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 WITHDRAW FROM CHKNG AMOUNT: 20.00 CURR BAL: 80.00 AVAILABLE: 80.00 RECEIPT 42 GORDON COLLEGE CARD 2 TRANS 1 WITHDRAW FROM CHKNG AMOUNT: 20.00 CURR BAL: 60.00 AVAILABLE: 60.00 <i>Do you want to perform another transaction?</i> <i>(1 = Yes, 2 = No)</i>

Appendix F: Mutants Designed for ATM System Case Study

In the following table (Table 25) we provide the detailed information of the mutants that are designed for ATM System case study. Another format of this mutant list, which is used by MIAT to insert the mutant, is shown as the follows.

<No.>@<Class>@<Line><Original Statement>@<Faulty Statement>

For example the first mutant can be written as:

1@ATM.java@305@_receiptPrinter.printReceipt(_number, _location, cardNumber, serialNumber, @_receiptPrinter.printReceipt(_number, _location, serialNumber, cardNumber,

Table 25 Mutants Designed for ATM System

No.	Class	Line	Original Statement	Faulty Statement	Category
1	ATM.java	305	_receiptPrinter.printReceipt(_number, _location, cardNumber, serialNumber,	_receiptPrinter.printReceipt(_number, _location, serialNumber, cardNumber,	AOC
2	ATM.java	401	private static final int STOPPED = 1;	Private static final int STOPPED = 0;	CRP
3	ATM.java	173	while (_state == RUNNING && readerStatus == CardReader.NO_CARD);	while (_state == RUNNING && readerStatus != CardReader.NO_CARD);	DER
4	ATM.java	173	while (_state == RUNNING && readerStatus == CardReader.NO_CARD);	while (_state == RUNNING readerStatus == CardReader.NO_CARD);	DER
5	ATM.java	302	public void issueReceipt(int cardNumber,int serialNumber,String description,Money amount,Money balance,Money availableBalance)	public void issueReceipt(int cardNumber,int serialNumber,String description,Money balance,Money amount,Money availableBalance)	POC
6	ATM.java	384	return _number;	return _state;	RSR
7	ATM.java	345	boolean result = (response == 1);// local variable result is used to support the contract	boolean result = (response == 2);// local variable result is used to support the contract	CRP
8	ATM.java	212	return PIN;	return _number;	RSR
9	ATM.java	265	{ return ! _cashDispenser.currentCash().less(amount);	{ return _cashDispenser.currentCash().less(amount);	RSR
10	Bank.java	131	_availableBalance [_currentTransactionAccount], amount));	amount, _availableBalance [_currentTransactionAccount]));	AOC
11	Bank.java	129	_currentBalance [_currentTransactionAccount], amount));	amount, _currentBalance [_currentTransactionAccount));	AOC
12	Bank.java	368	int result = choice - 1; // result is local variable to support the	int result = choice + 1; // result is local variable to support the	AOR

No.	Class	Line	Original Statement	Faulty Statement	Category
			contract	contract	
13	Bank.java	157	_currentBalance [_currentTransactionAccount].subtract(	_currentBalance [_currentTransactionAccount].add(	MRO
14	Bank.java	233	_currentBalance [_currentTransactionAccount].add(	_currentBalance [_currentTransactionAccount].subtract(	MRO
15	Bank.java	298	_currentBalance [fromAccount].subtract(amount);	_currentBalance [fromAccount].add(amount);	MRO
16	Bank.java	299	_currentBalance [toAccount].add(amount);	_currentBalance [toAccount].subtract(amount);	MRO
17	Bank.java	300	_availableBalance [fromAccount].subtract(amount);	_availableBalance [fromAccount].add(amount);	MRO
18	Bank.java	301	_availableBalance [toAccount].add(amount);	_availableBalance [toAccount].subtract(amount);	MRO
19	Bank.java	129	_currentBalance [_currentTransactionAccount], amount));	_availableBalance [_currentTransactionAccount], amount));	CNR
20	Bank.java	209	availableBalance.set(_availableBalance[_currentTransactionAccount]);	availableBalance.set(_currentBalance[_currentTransactionAccount]);	CNR
21	Bank.java	298	_currentBalance [fromAccount].subtract(amount);	_availableBalance [fromAccount].subtract(amount);	CNR
22	Bank.java	304	availableBalance.set(_availableBalance [toAccount]);	availableBalance.set(_currentBalance [toAccount]);	CNR
23	Bank.java	339	if (cardNumber < 1 cardNumber > NUMBER_CARDS)	if (cardNumber < 1 && cardNumber > NUMBER_CARDS)	LCR
24	Bank.java	106	if (cardNumber < 1 cardNumber > NUMBER_CARDS)	if (cardNumber < 1 && cardNumber > NUMBER_CARDS)	LCR
25	Bank.java	152	public void finishWithdrawl(int ATMnumber,int serialNumber,boolean succeeded)	public void finishWithdrawl(int serialNumber,int ATMnumber,boolean succeeded)	POC
26	Bank.java	104	public int initiateWithdrawl(int cardNumber,int PIN,int ATMnumber,int serialNumber,int from,Money amount,Money newBalance /* return value */, Money availableBalance /* return value */)	public int initiateWithdrawl(int cardNumber,int PIN,int ATMnumber,int serialNumber,int from,Money amount,Money availableBalance /* return value */, Money newBalance /* return value */)	POC
27	Bank.java	278	Money amount,Money newBalance,Money availableBalance)	Money newBalance, Money availableBalance,Money amount)	POC
28	Bank.java	333	int PIN,int ATMnumber,	int ATMnumber,int PIN,	POC
29	Bank.java	197	return	return	RSR

No.	Class	Line	Original Statement	Faulty Statement	Category
			Status.UNKNOWN_CARD;	Status.TOO_LITTLE_CASH;	
30	Bank.java	205	return Status.NO_SUCH_ACCOUNT;	return Status.SUCCESS;	RSR
31	Bank.java	383	{ return accountNames[type];	{ return accountNames[0];	RSR
32	Bank.java	303	newBalance.set(_currentBalance [toAccount]);	//	SDL
33	CashDispenser.java	70	_currentCash.subtract(amount);	_currentCash.add(amount);	MRO
34	EnvelopeAcceptor.java	77	if (insert.compareTo("insert") == 0)	if (insert.compareTo("insert") != 0)	ROR
35	Money.java	76	{ return new Money(0, minuend._cents - subtrahend._cents);	{ return new Money(0);	AND
36	Money.java	69	{ return new Money(0, first._cents + second._cents);	{ return new Money(0);	AND
37	Money.java	55	{ _cents = 100L * dollars + cents;	{ _cents = 100L * dollars - cents;	AOR
38	Money.java	69	{ return new Money(0, first._cents + second._cents);	{ return new Money(0, first._cents - second._cents);	AOR
39	Money.java	76	{ return new Money(0, minuend._cents - subtrahend._cents);	{ return new Money(0, minuend._cents + subtrahend._cents);	AOR
40	Money.java	83	{ _cents += other._cents;	{ _cents -= other._cents;	AOR
41	Money.java	92	{ _cents -= other._cents;	{ _cents += other._cents;	AOR
42	Money.java	107	{ return (int) _cents % 100;	{ return (int) _cents / 100;	AOR
43	Money.java	68	static public Money add(Money first, Money second)	static public Money add(Money second, Money first)	POC
44	Money.java	121	{ return _cents < other._cents;	{ return _cents > other._cents;	ROR
45	Session.java	208	private static final int RUNNING = 0;	private static final int RUNNING = 1;	CRP
46	Session.java	113	while (_state == RUNNING);	while (_state > RUNNING);	DER
47	Session.java	133	if (code != Status.INVALID_PIN)	if (code >= Status.INVALID_PIN)	ROR
48	Session.java	138	return Status.INVALID_PIN;	return Status.TOO_LITTLE_CASH;	RSR
49	Session.java	167	return _atm.reportTransactionFailure ("Envelope not deposited - transaction cancelled");default: return _atm.reportTransactionFailure (_bank.rejectionExplanation(r eason));	return _atm.reportTransactionFailure ("Transaction OK");default: return _atm.reportTransactionFailure (_bank.rejectionExplanation(r eason));	RSR
50	Session.java	198	{ return _PIN;	{ return _cardNumber;	RSR

No.	Class	Line	Original Statement	Faulty Statement	Category
	va				
51	Status.java	25	public static final int SUCCESS = 0;	public static final int SUCCESS = 1;	CRP
52	Status.java	39	public static final int INVALID_PIN = 4;	public static final int INVALID_PIN = 5;	CRP
53	Transaction.java	364	return _bank.initiateDeposit(_session .cardNumber(),_session.PIN(), _atm.number(),_serialNumber, _toAccount,_amount,_newBalance, _availableBalance);	return _bank.initiateDeposit(_session .cardNumber(),_session.PIN(), _atm.number(),_serialNumber, _toAccount,_amount,_availableBalance, _newBalance);	AOC
54	Transaction.java	293	_bank.finishWithdrawl(_atm.number(), _serialNumber, true);	_bank.finishWithdrawl(_serialNumber, _atm.number() , true);	AOC
55	Transaction.java	297	_amount,_newBalance,	_newBalance,_amount,	AOC
56	Transaction.java	483	_newBalance,_availableBalance);	_availableBalance,_newBalance);	AOC
57	Transaction.java	443	_amount = _atm.getAmountEntry();	_amount = _atm.startupOperation();	MRO
58	Transaction.java	97	case 1:	case 0:	CRP
59	Transaction.java	259	case 5: _amount = new Money(100); break;	case 8: _amount = new Money(100); break;	CRP
60	Transaction.java	104	result = new TransferTransaction(session, atm, bank);	result = new WithdrawlTransaction(session , atm, bank);	ICE
61	Transaction.java	107	result = new InquiryTransaction(session, atm, bank);	result = new DepositTransaction(session, atm, bank);	ICE
62	Transaction.java	98	result = new WithdrawlTransaction(session , atm, bank);	result = new InquiryTransaction(session, atm, bank);	ICE
63	Transaction.java	101	result = new DepositTransaction(session, atm, bank);	result = new WithdrawlTransaction(session , atm, bank);	ICE
64	Transaction.java	457	_session.PIN(),_atm.number(),	_atm.number(),_session.PIN(),	AOC
65	Transaction.java	146	{ result = code;	{result = Status.SUCCESS;	RSR
66	Transaction.java	264	return Status.SUCCESS;else return Status.TOO_LITTLE_CASH;	Return Status.SUCCESS;else return 2;	RSR
67	ReceiptPrinter.java	104	receipt[i++] = "AMOUNT:" " + amount.dollars() + "." +	receipt[i++] = "AMOUNT:" " + amount.cents() + "." +	MRO
68	ReceiptPrinter.java	105	((amount.cents() >= 10) ? "" + amount.cents()	((amount.cents() >= 10) ? "" + amount.dollars()	CIM
69	ReceiptPrinter.java	112	: "0" + availableBalance.cents() + "\n";	: "0" + balance.cents() + "\n";	SCR
70	Bank.java	110	return Status.INVALID_PIN;	return Status.UNKNOWN_CARD;	CRP
71	Bank.java	284	return Status.INVALID_PIN;	return	CRP

No.	Class	Line	Original Statement	Faulty Statement	Category
				Status.NO_SUCH_ACCOUNT;	
72	Bank.java	290	return Status.NO_SUCH_ACCOUNT;	return Status.INVALID_PIN;	CRP
73	Session.java	137	_state = ABORTED;	_state = RUNNING;	CRP
74	Session.java	186	{ return _cardNumber;	{ return _PIN;	RSR
75	Transaction.java	311	private Money _amount;	private Money _amount; private Money _newBalance;	HFA
76	Transaction.java	311	private Money _amount;	private Money _amount; private Money _availableBalance;	HFA
77	Transaction.java	403	private Money _amount;	private Money _amount; private int _serialNumber;	HFA
78	Transaction.java	403	private Money _amount;	private Money _amount; private ATM _atm;	HFA
79	Transaction.java	498	private Money _amount;	private Money _amount; private Bank _bank;	HFA
80	Transaction.java	498	private Money _amount;	private Money _amount; private Session _session;	HFA
81	Transaction.java	576	private int _fromAccount;	private int _fromAccount; private Money _newBalance;	HFA
82	Session.java	70	_currentTransaction = null;	_currentTransaction = new WithdrawlTransaction(this,_atm,_bank);	ICE
83	Session.java	70	_currentTransaction = null;	_currentTransaction = new DepositTransaction(this,_atm,_bank);	ICE
84	ATM.java	302	public void issueReceipt(int cardNumber,int serialNumber,String description,Money amount,Money balance,Money availableBalance)	public void issueReceipt(int cardNumber,int serialNumber,String description,Money amount,Money availableBalance,Money balance)	POC
85	ATM.java	302	public void issueReceipt(int cardNumber,int serialNumber,String description,Money amount,Money balance,Money availableBalance)	public void issueReceipt(int cardNumber,int serialNumber,String description,Money availableBalance,Money balance,Money amount)	POC
86	Bank.java	104	public int initiateWithdrawl(int cardNumber,int PIN,int ATMnumber,int serialNumber,int from,Money amount,Money newBalance /* return value */, Money availableBalance /* return value */)	public int initiateWithdrawl(int from,int PIN,int ATMnumber,int serialNumber,int cardNumber,Money amount,Money newBalance /* return value */, Money availableBalance /* return value */)	POC

No.	Class	Line	Original Statement	Faulty Statement	Category
87	Bank.java	278	Money amount,Money newBalance,Money availableBalance)	Money newBalance,Money amount,Money availableBalance)	POC
88	Bank.java	293	return Status.CANT_WITHDRAW_FROM_ACCOUNT;	return Status.INSUFFICIENT_AVAILABLE_BALANCE;	CRP
89	Bank.java	352	return Status.SUCCESS;	return Status.NO_SUCH_ACCOUNT;	CRP
90	CashDispenser.java	46	_currentCash = new Money(0);	_currentCash = new Money(100);	CRP
91	CashDispenser.java	81	{ return _currentCash;	{ return new Money(100);	RSR
92	Transaction.java	255	case 1: _amount = new Money(20); break;case 2: _amount = new Money(40); break;	case 1: _amount = new Money(20);/* break;*/case 2: _amount = new Money(40); break;	CFD
93	Session.java	97	break;	//break;	CFD
94	Session.java	102	break;	//break;	CFD
95	Session.java	134	return code;	break;else return code;	CFD
96	Transaction.java	259	case 5: _amount = new Money(100); break;	case 5: _amount = new Money(100);	CFD
97	Transaction.java	264	return Status.SUCCESS;else return Status.TOO_LITTLE_CASH;	return Status.TOO_LITTLE_CASH;else return Status.SUCCESS;	SSO
98	Session.java	167	return _atm.reportTransactionFailure("Envelope not deposited - transaction cancelled");default:return _atm.reportTransactionFailure(_bank.rejectionExplanation(reason));	return _atm.reportTransactionFailure(_bank.rejectionExplanation(reason)); default: return _atm.reportTransactionFailure("Envelope not deposited - transaction cancelled");	SSO
99	Transaction.java	255	case 1: _amount = new Money(20); break;case 2: _amount = new Money(40); break;	case 1: _amount = new Money(40); break; case 2: _amount = new Money(20); break;	SSO
100	ATM.java	369	{ _display.reportCardRetained();	{ // _display.reportCardRetained();	SDL
101	Transaction.java	292	_atm.dispenseCash(_amount);	_atm.dispenseCash(_newBalance);	SCR
102	Transaction.java	441	_fromAccount = _bank.chooseAccountType("transfer from", _atm);	_toAccount = _bank.chooseAccountType("transfer from", _atm);	SCR
103	ATM.java	231	while (choice>numItems)// compare the choice and numItems to avoid	while (choice<numItems)// compare the choice and numItems to avoid	ROR
104	ATM.java	358	_display.cardEjected();	// _display.cardEjected();	SDL
105	ATM.java	265	{ return !	{ return !	CIM

No.	Class	Line	Original Statement	Faulty Statement	Category
			<code>_cashDispenser.currentCash().less(amount);</code>	<code>_cashDispenser.currentCash().equals(amount);</code>	
106	CardReader.java	171	<code>_status = CARD_HAS_BEEN_READ;</code>	<code>_status = UNREADABLE_CARD;</code>	CRP
107	CardReader.java	188	<code>{ return _cardNumberRead;</code>	<code>{ return _status;</code>	SCR
108	CardReader.java	166	<code>if (String.valueOf(number) == null)</code>	<code>if (String.valueOf(number) != null)</code>	ROR
109	OperatorPanel.java	72	<code>if (password.compareTo("123456")==0)</code>	<code>if (password.compareTo("123456")!=0)</code>	ROR
110	OperatorPanel.java	96	<code>return new Money(20 * numberBills);</code>	<code>return new Money(20 + numberBills);</code>	AOR
111	EnvelopeAcceptor.java	49	<code>_inserted = false;</code>	<code>_inserted = true;</code>	LCR
112	EnvelopeAcceptor.java	49	<code>_inserted = false;</code>	<code>//_inserted=false;</code>	SDL

Appendix G: Diagnosability of Individual Mutant and Analysis

In this section we provide the diagnosability value for each mutant killed by the contracts (Table 26). After that we analyze the mutants whose diagnosability with the Highest Precise contracts are not 1 with the UML sequence diagram.

Table 26 Diagnosability of Individual Mutant

Mutant No.	With Oracle	With Highest Precise Contract	With Strongest simplification	With Intermediate Simplification
5	5.00	1	1	1
7	1.00	1	1	1
9	14.24	1	1	1
10	26.00	1	1	1
11	25.00	1	1	1
12	11.92	1	1	1
13	8.00	1	1	1
14	32.00	1	1	1
15	21.00	1	1	1
16	15.00	1	1	1
17	21.00	1	1	1
18	15.00	1	1	1
21	21.00	1	1	1
27	15.00	10	10	10
28	7.00	1	1	1
29	13.00	1	1	1
30	13.00	1	1	1
31	9.11	1	1	1
32	15.00	1	1	1
33	8.00	1	1	1
34	19.56	1	1	1
35	11.00	1	1	1
36	18.00	1	1	1
37	22.33	1	1	1
38	19.00	1	1	1
39	22.00	1	1	1
40	18.29	1	1	1
41	15.00	1	1	1
42	4.00	1	1	1
44	18.90	1	1	1
46	4.00	1	1	1
47	2.00	1	1	1
48	19.00	1	1	1
50	15.40	1	1	1
51	10.00	8	8	8
53	16.00	1	1	1
55	6.00	1	1	1
60	7.00	1	1	1
61	7.00	1	1	1
62	7.00	1	1	1

63	7.00	1	1	1
64	8.00	1	1	1
65	8.00	1	1	1
66	9.00	1	1	1
67	1.00	1	1	1
68	1.00	1	1	1
69	1.00	1	1	1
70	15.00	1	Killed by Oracle	1
71	14.00	1	Killed by Oracle	1
72	17.00	1	Killed by Oracle	1
74	15.41	1	1	1
75	12.00	1	1	1
76	12.00	1	1	1
77	18.00	1	1	1
78	8.00	7	7	7
79	8.00	1	1	1
81	14.00	1	1	1
82	25.77	1	1	1
83	25.77	1	1	1
84	5.00	1	1	1
85	5.00	1	1	1
86	19.00	1	1	1
87	17.00	1	1	1
89	17.00	1	Killed by Oracle	Killed by Oracle
91	15.40	1	1	1
92	14.00	1	1	1
93	23.19	6	6	6
95	2.00	1	1	1
96	10.00	1	1	1
97	10.27	1	1	1
98	2.00	1	1	1
99	16.00	1	1	1
101	16.00	2	2	2
103	1.00	1	1	1
105	12.00	1	1	1
107	27.80	1	1	1
109	3.00	1	1	1
110	28.00	1	1	1
111	29.00	1	1	1
Average	13.15	1.35	1.37	1.36

Mutant 27

Mutant Location: Bank.doTransfer(...), arguments' order of the method are exchanged.

Exception Information:

```
Jcontract: PreException: [_availableBalance.getValue() >=
_amount.getValue()], in thread "Thread-2"
at
TransferTransaction.finishApprovedTransaction$dbc$prem2(Transaction.java:
468)
```

```

at
TransferTransaction.finishApprovedTransaction$dbc$pre(Transaction.java:47
4)
at TransferTransaction.finishApprovedTransaction(Transaction.java:474)
at Transaction.doTransactionUseCase(Transaction.java:147)
at Session.doSessionUseCase(Session.java:85)
at ATM.serviceCustomers(ATM.java:180)
at ATMMain.run(ATMMain.java:96)
at java.lang.Thread.run(Unknown Source)

```

Sequence Diagram:

Figure 24 Sequence Diagram of Mutant 27

Diagnosis Flow Calculation:

1. Transaction.doTransactionUseCase()
2. TransferTransaction.getCustomerSpecification()
3. Bank.chooseAccountType()
4. ATM.getAmountEntry()
5. Status
6. TransferTransaction.setToBank()
7. Session.cardNumber
8. Session.PIN()
9. ATM.number()
10. Bank.doTransfer()

Diagnosability: 10

Analysis:

For this mutant although the contracts didn't catch the fault immediately, the fault is caught by the other contracts because the mutant changed the member variables' value.

Mutant 51

Mutant Location: Modify constant SUCCESS value from 0 to 1 in Status class.

Exception Information:

```
Jcontract: PreException: [amount.getValue() <= _currentCash.getValue()],  
in thread "Thread-2"  
at CashDispenser.dispenseCash$dbc$pre(CashDispenser.java:65)  
at CashDispenser.dispenseCash(CashDispenser.java:69)  
at ATM.dispenseCash(ATM.java:272)  
at WithdrawlTransaction.finishApprovedTransaction(Transaction.java:288)  
at Transaction.doTransactionUseCase(Transaction.java:147)  
at Session.doSessionUseCase(Session.java:85)  
at ATM.serviceCustomers(ATM.java:180)  
at ATMMain.run(ATMMain.java:96)  
at java.lang.Thread.run(Unknown Source)
```

Sequence Diagram:

Figure 25 Sequence Diagram of Mutant 51

Diagnosis Flow Calculation:

1. ATM.dispenseCash()
2. WithdrawlTransaction.finishApprovedTransaction()
3. Transaction.doTransactionUseCase()
4. WithdrawlTransaction.getTransactionSpecificsFromCustomer()
5. Bank.chooseAccountType()
6. Am.getMenuChoice()
7. Atm.checkIfCashAvailable()
8. Status

Diagnosability: 8

Analysis:

Mutant 51 is changing the definition of the constant value. The contracts can't address the fault directly because we also use the constant's name in the contracts.

Mutant 78

Mutant Location: a new attribute that has the same name in parent's class is redefined in child's class.

Exception Information:

```
Jcontract: PreException: [atm != null], in thread "Thread-2"
at Bank.chooseAccountType$dbc$pre(Bank.java:360)
at Bank.chooseAccountType(Bank.java:365)
at
DepositTransaction.getTransactionSpecificsFromCustomer(Transaction.java:3
44)
at Transaction.doTransactionUseCase(Transaction.java:135)
at Session.doSessionUseCase(Session.java:89)
at ATM.serviceCustomers(ATM.java:185)
at ATMMain.run(ATMMain.java:106)
at java.lang.Thread.run(Unknown Source)
```

Sequence Diagram:

Figure 26 Sequence Diagram of Mutant 78

Diagnosis Flow Calculation:

1. DepositTransaction.getTransactionSpecificationFromCustomer
2. Transaction.doTransactionUseCase
3. Session.doSessionUseCase
4. ATM.getPIN
5. Transaction.chooseTransaction
6. ATM.getMenuChoice
7. DepositTransaction.DepositTransaction

Diagnosability: 7

Analysis:

This mutant is caught by the contract when the added attribute is used in the contract because the redefined attribute is not initialized.

Mutant 93

Mutant Location: break statement is deleted in method doSessionUseCase() of Session class.

Exception Information:

```

Jcontract: PreException: [$ppre.put ("_availableBalance [_accountNumber [
cardNumber ] [ from ]].getValue()",new java.lang.Long(_availableBalance
[_accountNumber [ cardNumber ] [ from ]].getValue()));\n...]: generated:
java.lang.ArrayIndexOutOfBoundsException, in thread "Thread-2"
    at Bank.doTransfer$dbc$ppre(Bank.java:0)
    at Bank.doTransfer(Bank.java:279)
    at TransferTransaction.sendToBank(Transaction.java:456)
    at Transaction.doTransactionUseCase(Transaction.java:141)
    
```

```

at Session.doSessionUseCase(Session.java:89)
at ATM.serviceCustomers(ATM.java:185)
at ATMMain.run(ATMMain.java:106)
at java.lang.Thread.run(Unknown Source)

```

Sequence Diagram:

Figure 27 Sequence Diagram of Mutant 93

Diagnosis Flow Calculation:

1. TransferTransaction.sendToBank()
2. Transaction.doTransactionUseCase
3. Transaction.getTransactionSpecifcsFromCustomer
4. ChooseAccoutType
5. Status
6. Session.doSessionUseCase

Diagnosability: 6

Analysis: The new added attribute is caught by the precondition of Bank.doTransfer() when the precondition calls this attribute.

Mutant 101

Mutant Location: The input argument of method `_atm.cashDispense()` is changed when this method is called in method `withdrawalTransaction.finishApprovedTransaction()`.

Exception Information:

```

Jcontract: PreException: [amount.getValue() <= _currentCash.getValue()],
in thread "Thread-2"

```

```

at CashDispenser.dispenseCash$dbc$pre(CashDispenser.java:65)
at CashDispenser.dispenseCash(CashDispenser.java:69)
at ATM.dispenseCash(ATM.java:277)
at WithdrawalTransaction.finishApprovedTransaction(Transaction.java:292)
at Transaction.doTransactionUseCase(Transaction.java:152)
at Session.doSessionUseCase(Session.java:89)
at ATM.serviceCustomers(ATM.java:185)
at ATMMain.run(ATMMain.java:106)
at java.lang.Thread.run(Unknown Source)

```

Sequence Diagram:

Figure 28 Sequence Diagram of Mutant 101

Diagnosis Flow Calculation:

1. ATM.dispenseCash
2. WithdrawalTransaction.finishApprovedTransaction

Diagnosability: 2

Analysis: The diagnosability is 2 because `ATM.cashDispense()` is a delegation method.