AN EMPIRICAL STUDY ON A REGRESSION TESTING TECHNIQUE FOR OO SOFTWARE

by

ALI TOUFIC NEHME

B.S. Business Computer, Saint-Joseph University, 2002

Project submitted in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science

Division of Computer Science and Mathematics

LEBANESE AMERICAN UNIVERSITY

June 2006

LEBANESE AMERICAN UNIVERSITY

Project approval Form

Student Name:	Ali Nehme	I.D.: 200202113
Project Title :	An Empirical Study on a Regression Testing Technique for OO Software	
Program :	M.S. in Computer Science	
Division/Dept :	Computer Science and Mathematics	
School :	Arts and Sciences - Beirut	

Approved/Signed by:

Project Advisor	Dr. Nashat Mansour	[redacted]
Member	Dr. Faisal Abu-Khzam	[redacted]

Date: June 27, 2006

Plagiarism Policy Compliance Statement

I certify that I have read and understood LAU's Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.

This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name:

Signature: Date:

I grant to the LEBANESE AMERCIAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or its students and employees. I further agree that the University may reproduce and provide single copies of the work to the public for the cost of reproduction.

To my father R.I.P.

Acknowledgment

The existence of this project is entirely due to the foresight of my supervisor Dr. Nashat Mansour. A special thanks to Dr. Abu-Khzam for being in my project committee.

I would like to express my appreciation for my project manager at the National Social Security Fund for his patience and support over the past years.

Finally, I would like to express my deepest gratitude for the constant support and understanding that I received from my mother, family and friends.

Abstract

Regression testing is an essential activity aiming to prove that the unmodified sections of a software application are unaffected by changes. This paper presents an empirical study on a regression test selection technique for object-oriented software. The technique is based on the Unified Modeling Language (UML) program design; more specifically it makes use of the class diagrams, interaction overview diagram and the sequence diagrams. Nine versions of three case studies test cases are used: the UML design is specified, test cases are drawn and then they are studied in terms of the number of selected test cases, inclusiveness and precision. We prove the feasibility of the technique and demonstrate the ability of the regression testing technique to select a small number of test cases for retest from the original set of test cases; We also provide evidence revealing high inclusiveness and precision rates; an amendment was also introduced to cope with a problem in detecting changes inside the fragments of the sequence diagrams.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Software testing is the process utilized to identify the completeness, accuracy and quality of computer software. The most important role of testing is executing an application with the objective of finding an error. We classify a test case as good when it has a high probability of finding a new error. Another objective of testing is proving that software functions are working with respect to the specification. The test information gathered from this phase provides an insight of the software consistency and its quality in general. Proving this aspect does not eliminate its counterpart, testing may fail to find bugs in the program thus failing to guarantee a bulletproof system.

Object-oriented software diverges from conventional procedural software in terms of analysis, design and actual coding; consequently specific testing support is needed. UML (Unified Modeling Language) is a standard object modeling language used in software engineering. This language makes use of graphical notations to depict the design of the software. This paper makes use of this language for its widely usability, standardization and its ability to solve recurring architectural problems

1.1 Types of Testing

Testing can be divided into several types: black-box testing is based on the output of the software in terms of specifications; it is equivalent to using the same interface the user

would have, the tester in this case is not aware of the internal operations of the application's software. In white-box testing, the developer has access to the application's source code. In this type of testing, the programmers tend to find faults or insure proper functionality of the applicant's components. This kind of testing, also known as glass-box testing, uses the control structure of the procedural design to obtain test cases.

Dealing with code-based testing can become a tedious and an unrealistic approach when aiming for reliable applications. This approach becomes harder to track when multiple developers are working on the same application; automated systems are used to be able to cope with this problem. Many of the tools that are found on the market have the ability to change the design when a change has been made to the code. (Leung and White, 1991) showed that a selective testing technique that is more cost-effective than the technique that tries to retest everything in one case: it's when the cost of choosing a compact subset of test cases to be run is lower than the cost of executing the test cases that the selective technique exclude. In other words, the cost of selecting regression test cases to be re-run must be lower than the cost of executing the remainder test cases for test selection to have a meaning.

1.2 Purpose of Regression Testing

Object-oriented software is usually built from stable components, such as classes that has been used and tested in previous applications. This reusability feature is one of the main reasons of coming up with object-oriented applications. Testing techniques should take advantage of this aspect in order to facilitate the testing phase that every software goes through.

After making modification to any software, retesting the whole system after each change is impractical. Regression testing ensures that the altered functions or code did not affect the functionality of existing working functionality; this is done by selecting a subset of previous test cases and then running them with the new system. "Regression testing is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements" (IEEE, 1990).

Regression testing largely reduces the overhead of testing by looking at the changes that took place in the system and comes out with a set of tests needed to be re-executed. The test data generation takes place when the existing test data has been altered or rearranged. Other than testing the accuracy of an application, regression testing is also used to track the quality of its output by looking at code size or execution time.

The empirical results presented in this report are based on a recent regression test cases selection algorithm (Takkouch, 2006). This algorithm makes use of the UML object modeling language; in particular it employs a new type of diagram (interaction overview diagram) found in its latest release.

1.3 Scope and Objective of the Project

This project report presents an empirical study of an algorithm for regression test cases selection based on Unified Modeling Language (UML) diagrams. The importance of this technique is that it is done at the design level without even accessing the source code. This algorithm makes use of three types of diagrams: class diagrams, sequence diagrams and interaction overview diagrams. Based on this algorithm, three case studies are

presented to measure the feasibility of this algorithm. Each case study has its first version which specifies the application in its initial state; three versions are then presented for each case study in order to do further experimentation to the regression testing algorithm. UML class diagrams illustrate the classes of the system, their inter-relationships, and the methods and attributes of those classes. The interaction overview diagram focuses on the overview of the flow of control of the interactions. Finally, sequence diagrams model the flow of logic within the system in a visual manner. When modifications are made to the application, it is reflected in the diagrams used in this technique.

An improvement to this technique is also presented; this improvement is made on the sequence diagram level.

The empirical study utilizes the technique to extract test cases from the diagrams presented in each case study. These test cases are then tested for accuracy and inclusiveness in order to assess the technique.

1.4 Report Outline

The remainder of the project paper is divided into seven chapters: chapter 2 present a set of previous work related to regression testing techniques. Chapter 3 first present recent works on UML based regression testing in addition to an overview of the technique used. An improvement for the technique was also presented in this chapter. Chapter 4 defines a framework used to analyze the regression test selection technique. Chapter 5 presents 3 case studies used for the assessment of the technique. Chapter 6 presents a summary of the empirical results and a discussion of these results and finally a conclusion is shown in Chapter 7.

Chapter 2

OO Regression Testing Techniques

Object-Oriented software testing needs to cope with novel problems brought up by the powerful features of OO languages. The new OO characteristics (such as encapsulation, inheritance and polymorphism) present great improvement to the software design and coding. These new enhancement raised new problems to the in the software maintenance and testing (Wilde and Huitt, 1992). All major software development companies have transitioned to the Object-Oriented paradigm. As more and more OO applications are being developed, the number of companies seeking innovative test techniques for OO software is in a rise.

2.1 Path Analysis Technique

Benedusi, Cimitile and De Carlini (1988) presented a selective regressive testing technique based on path analysis. This technique first takes as an input the set of the application's paths in the modified version denoted in algebraic expression. It then alters this extracted expression to end up with a cycle-free exemplar path. The next step is comparing both exemplar paths of the application before update and after, the technique then classifies the paths into four categories: new, modified, unmodified and canceled. The technique ends up by selecting all the tests that go through modified exemplar paths.

2.2 Dataflow Techniques

This type of technique is based on the dataflow analysis. In dataflow testing, a variable assignment in an application is testes by producing test that execute subpath from assignment to points where the method value is used. In general, these technique spots definition-use pairs, which are either modified or newly introduced in the modified application, and selects tests that implement these pairs. Uses in an application can be divided into two types computation uses or predicate uses. The first type arise when a variable is employed in a computation or output statement, the second type is used each time a variable is utilized in a predicate statement.

Largely, two approaches have been proposed: incremental and nonincremental. The first starts by processing one change, then picks the tests for that change and after that it update the dataflow and test information (incrementally), finally it repeats the same described process for the next change. Nonincremental techniques work on multiple modifications at the same time.

Conventional dataflow techniques uses control flow graph (CFG) representation of an application. In such representation, each node matches a statement executed and the edge depicts the flow of control between these statements. In 1988 and 1989 Harold and Soffa have presented such approach in addition to a testing tool.

A new improvement to the Control Flow Graph, (Landi and Ryder, 1992) and (Rothermel et al, 2000) uses Interprocedural Control Flow Graphs (ICFG). Since the CFG has a limitation of coping with only a single method, the ICFG has the ability to encode a group of related methods as long as they have a single entry point. The ICFG contains a Control Flow Graph (CFG) for the methods inside the program, each call site

in the program is represented as pair of nodes denotes as call and return nodes. Each call node is linked to the initializing and exit nodes of the call routine are each connected to an edge denoted call edge and return edge respectively. The main drawback of the ICFG is not being able to deal with all the OO features. The replacement of frames to be able to procure of the OO features did not cope with all the OO related characteristics.

2.3 Function Dependence Graph

This technique employs a high level of abstraction rather than involving the multifaceted relationship between statements (Wu et al, 1999). This technique will first identify the methods, attributes and dependence relationships that affected were affected by the application's modification. For a test case to be picked from the pool of test cases, it should execute one of the modified methods that will cause the application to behave differently. A Function Calling Graph (FCG) was also created in order to examine by which way the program behavior would be altered. The FCG will be generated for each test case in order to keep track of the method execution sequence.

This approach allows the effective selection of a regression test suite. Moreover, after evaluating the modified and original dependence relationships, data of new occurrences of this kind of relationships can be provided. Based on this data, new test case can now be generated to assess the modified application.

2.4 The Firewall Technique

In 1990, Leung and White presented a retesting technique that is classified as selective. This technique was targeted at interprocedural regression testing. This technique supported specification and code modifications. Leung and White methodology defines

where to place a *firewall* around modified code functions. This technique picks unit test cases for modified code that lies inside the firewall, the integration test cases are also selected if they fall inside the firewall. In 1992, Leung and White introduced an enhancement for their technique; it now has the possibility to support interactions engaging global variables. The firewall approach does not necessitate the use of any testing technique or even a specific coverage criterion.

Based on the work presented above, Hsia et al (1997) used the firewall technique to test Object-Oriented classes. To create the firewall, the relation between the classes was used. The main idea behind this innovative technique was to first determine the relations between classes and test cases, and then picks the set of test cases that goes through the modified classes.

2.5 Slicing Techniques

Agrawal et al (1993) described a family of selective technique for retest that use slicing. For each test belonging to the test cases pool, each technique defines a slice. The paper's authors propose several methods that identify the test cases of the regression test that were subject to modifications. t is defined as a test belonging to the set of tests in T ($t \in T$), P denotes the program being tested. The slice types discussed in the paper are presented below:

a. Execution slice: an execution slice for *t* only encloses the statement in P that was run by *t*.

b. Dynamic slice: a dynamic slice for *t* holds all the statements in the execution slice for *t* that has affected the output statement in the execution slice.

c. Relevant slice: the relevant slice is comparable to the dynamic slice; the difference is that it holds predicate statement (in t) that may cause the program to generate different output if it was altered.

d. Approximate relevant slice: this kind of slice is similar to the dynamic slice, it has an addition to it, and it holds the predicate statements in the execution slice for t.

When one of these slice techniques selected a slice for test, if this slice contains a modified statement, the method will select this test case.

Chapter 3

UML Based Regression Testing

The UML embodies a set of best engineering practices that were able to demonstrate successfulness when modeling large and complex systems. The industry is always on the run searching for methodologies to automate the production of systems and to improve quality and decrease cost. These techniques include component technology, patterns, visual programming and frameworks. UML was able to cope with all these need in addition to being compatible with systems designed for the web.

3.1 Related Work

3.1.1 UML Use Cases, Class and Sequence Diagrams

In 2003, Briand et al described a technique to classify regression test cases. The test cases falls into three categories: obsolete, retestable and reusable. The classification used the design data (UML Model) extracted from the systems, before and after update, and the data generated from the test cases. This technique uses the model before and after update to compare the two versions, it uses the UML diagrams: use case, class and sequence.

The method starts by comparing the application's class diagrams (before and after update); it then compares the sequence diagrams for each test case. After this comparison, it classifies the test cases based on their impact. If a test case led to an invalid execution sequence, it is considered obsolete. When the sequence of messages

from a test case remains valid, they are considered retestable and finally when the sequence of messages remains intact in the application after modification, it is considered reusable.

3.1.2 UML Collaboration, Sequence and State Chart Diagrams

Another approach by Wu et al (2003) for testing component-based software utilized several UML diagrams in order to model the internal behavior of components. The approach presented cover all content and context dependencies, it also picks all the transitions in the system.

In their paper, they presented two types of approaches, the first is based on the collaboration/sequence diagram and the subsequent is based on the state chart diagram. The former approach focuses on the interaction between multiple objects inside a use case; the latter will include all the potential sequences in a collaboration diagram in addition to the all possible combination of the state chart diagram. The state chart diagram is used since collaboration diagram in itself is not sufficient.

3.2 Technique Overview

The empirical studies of this paper were based on the UML based regression testing technique by Takkoush (2006). This technique uses three kinds of diagrams: class, sequence and interaction overview. The methodology presented presumes am accurate correlation between the UML design and the application's code.

3.2.1 Changes in Class Diagrams

The class diagram is a static structure diagram that illustrates the structure of a system by visualizing the system's classes and relationships among these classes. This diagram represents the blue print of the system. Figure 3.1 represents a class diagram of a Library System.

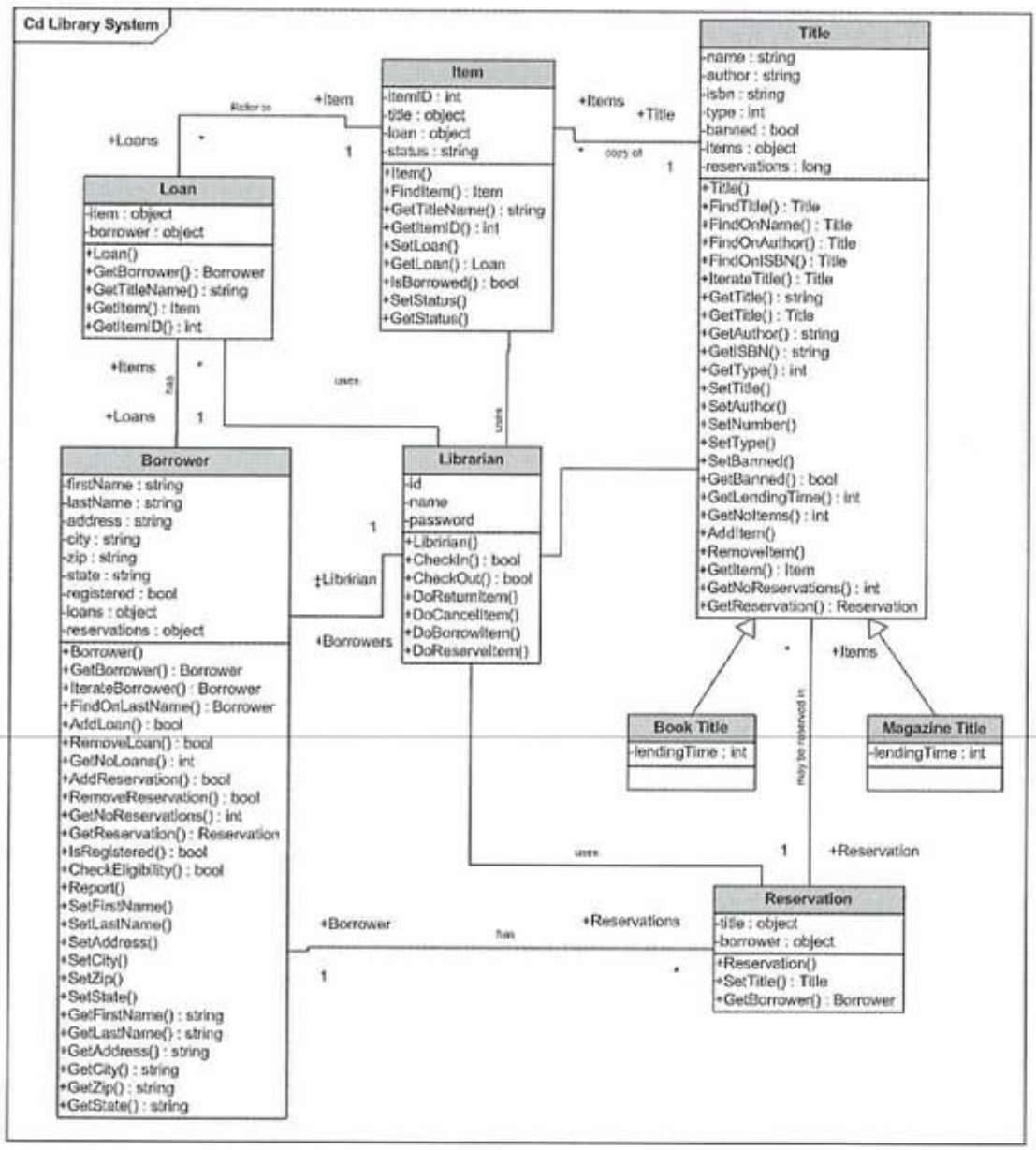

Figure 3.1 Library System (v1) class diagram

The changes that affect a class diagram turn it from the original version (CD) to its new modified version (CD'). The changes made on the class diagram level can have an effect on the attributes, methods, relationships and classes. The type of change that affects these modifications can be categorized into three types: new, deleted (suppressed from the system) and modified (that have encountered changes). All modifications that include a deletion change should remove all reference to the deleted change. The changes to relationships include adjustment in the kind of the relationship, the multiplicity or the direction of the link. We thus classify a class as changed if an attribute or method inside that class has been modified.

3.2.2 Changes in Interaction Overview Diagrams

The interaction overview (IO) diagrams are new to UML; they were introduced in the UML 2.0. This diagram is a mixture of interaction diagrams and activity diagrams. This type of diagram sum up the control flow of the whole application, it provides a big-picture overview how the interaction diagrams are connected from the point of view logic and process-flow. Figure 3.2 depicts an interaction overview diagram of a Library System.

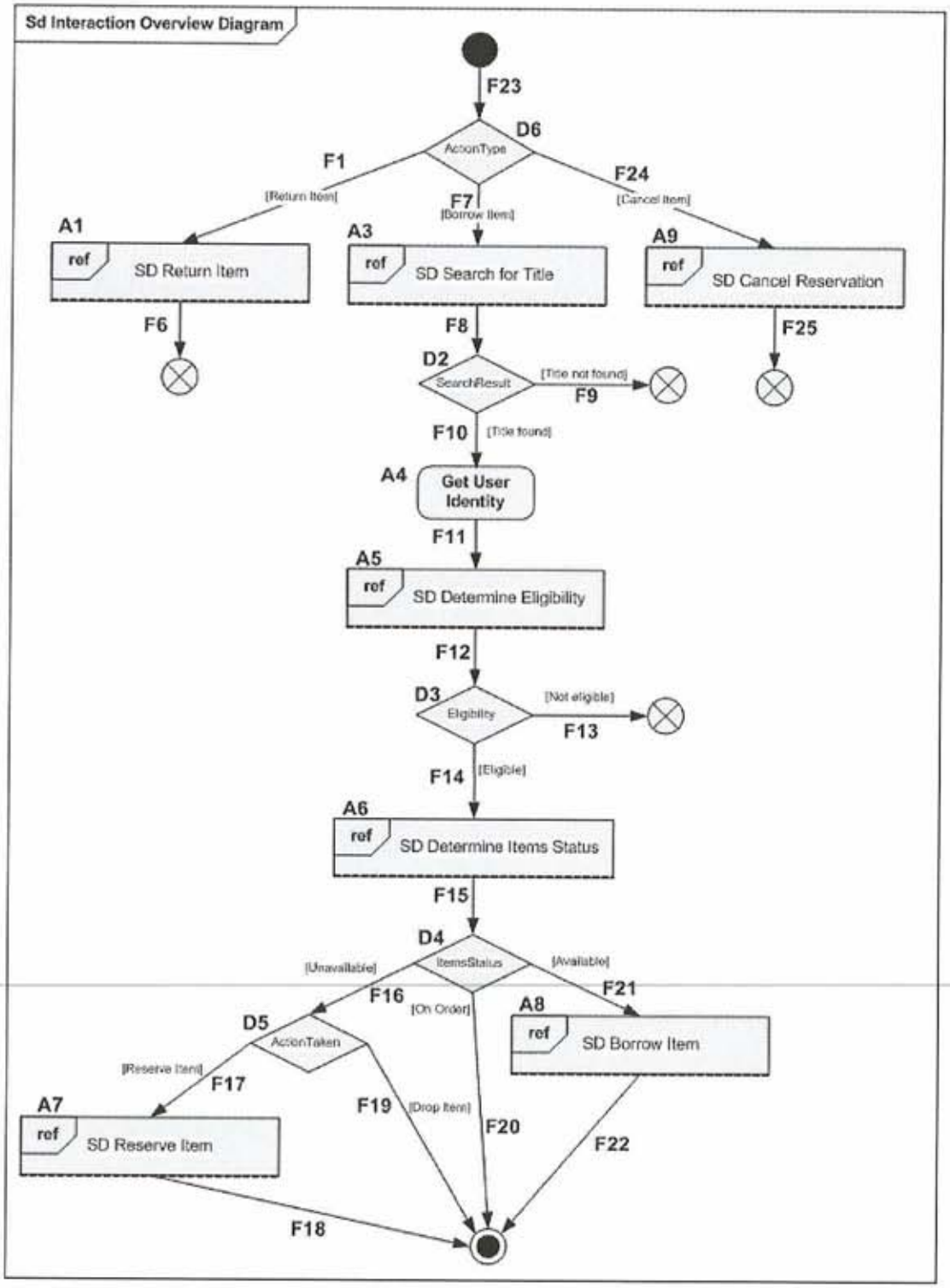

Figure 3.2 Library System (v1) interaction overview diagram

The interaction overview diagram consists of several components: activity, initial nodes, activity final, fork, join and flow/edge …

When a change occurs, it may reflect the interaction overview diagrams in terms of behavioral changes. The updates in this type of diagram can be classified into 2 major parts: the changes in the interaction between the sequence diagrams and changes in the sequence diagrams. The flow can be modified by changes the condition, source or target. Decision and flow final cannot be modified; they can only be inserted or removed from the diagram.

3.2.3 Changes inside Sequence Diagrams

The sequence diagram (SD) illustrates the sequence of messages, which are exchanged between roles that implement the behavior of the system, ordered in time. Sequence diagrams focuses on identifying the behavior within the system. The sequence diagram after the update is denoted by SD'.

The main components constituting the sequence diagrams are the lifeline and the messages. The first represents participants in the system, there are many time of participants: actors, object or any entity. The second are considered as methods invokers. Figure 3.3 is a representation of a UML sequence diagram, messages are the horizontal method calls and the lifelines are presented in vertical lines.

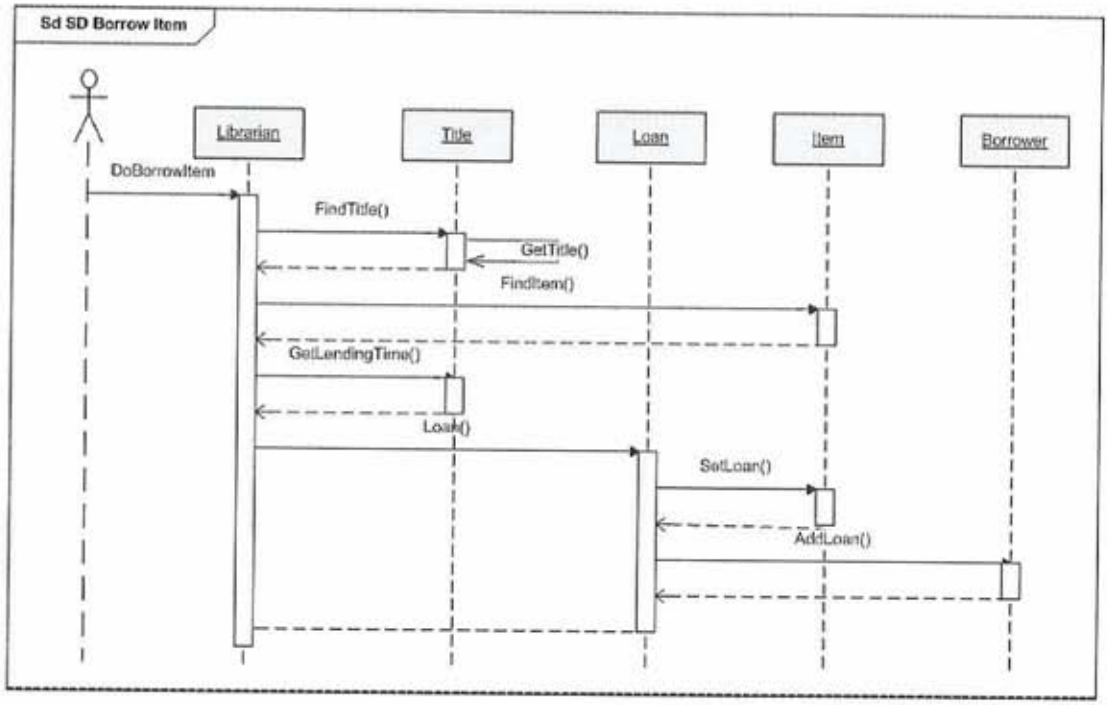

Figure 3.3 Library System (v1) Borrow Item SD

3.2.4 Test Cases Selection

The regression testing technique (Takkoush 2006) we use to perform our empirical study is based on two diagrams: the interaction overviews diagram that recapitulates the flow of the whole application's software and the class diagram that reflect the modifications on the method level. Sequence diagram are present in the interaction overview diagram and represent components of this diagram.

This technique can be categorized in 2 parts: the part that detect and classify the changes based on the class diagram and interaction overview diagram, and the other part that classify the test cases based on their coverage. The algorithm is divided into two parts the first will produce unit test cases and integration test cases for retest based on the class diagram. The next one only picks tests that are based on the modifications that affected the interaction overview diagram.

3.2.4.1 Based on Class Diagrams

The test selection based on changes in class diagrams are divided into three parts (*GenerateChangedMethods*) and can be summarized as follow:

a. Directly changed methods (*GenerateChangedMethods*): We denote by M the set of changed methods. By comparing methods signatures before and after update, if a method signature has been changed, it is directly added to M.

b. Integration test case selection from changed method (*CDIntegrationTestSelection*): This step will go through the methods already classified in M, and will select from the ordered path list per sequence diagram (denoted by T.SD) that reference the methods belonging to M.

c. Unit test case selection from method changes (*CDUnitTestSelection*): This algorithm will select the unit test cases in M and uses the transitive closure approach to select all the unit test cases that the method is accessible from.

3.2.4.2 Based on Interaction Overview Diagrams

The test case selection based on changes in the interaction overview diagram is also divided into three parts:

a. Interaction overview test case selection (IOTestCaseSelection): It compares every SD with its appropriate SD' in IO and IO'. It then marks the changed sequence diagrams as marked; the comparison inside the SD is done lifeline by lifeline.

b. Interaction overview diagram based classification (*IOBasedClassifiaction*): This step consists of traversing the original and updated IO diagrams to spot the changes along the paths and thus selected the test cases going through the modified paths. All changed flow, edge, decision…will be marked for retest.

c. Sequence diagrams based reduction (*SDBasedReduction*): This step is a refinement step, it will re-examine the integration level test cases for retest and check if it's still valid in the changed SD.

3.3 Technique Improvement

Earlier versions of UML (1.X) failed to cover the logic of some sequences being modeled. This lack of functionality was a real setback to this standard modeling language. The new UML 2.0 tackled this problem by adding z new notation called Combined Fragment. This innovative feature is employed to group a set of messages in order to show conditional flow in the sequence diagram.

The Combined Fragment convention allows users to define expressions of interaction fragments in the sequence diagrams. It does that by assigning an interaction operator and an equivalent interaction operand. In the UML 2.0, eleven combined fragment were supplied:

- Alternative fragment (*alt*): models if…then…else statements.
- Option fragment (*opt*): models switch statements.
- Parallel fragment (*par*): models simultaneous processing.
- Loop fragment (*loop*): includes a series of messages which are repeated.

- Break fragment (*break*): models a substitute sequence of events that is executed as an alternative of the rest of the diagram
- Weak sequencing fragment (*seq*): includes a number of sequences where the messages have to be executed before the next segments are launched (the sequence doesn't have to share a lifeline.
- Strict sequencing fragment (strict): includes a set of messages that should be executed in a specified order.
- Negative fragment (*neg*): includes an invalid set of messages
- Critical fragment (*critical)*: includes a non-interruptible sub-process with self-contained combined fragments.
- Ignore fragment (*ignore*): define a set of messages of no interest if it appears in the current context.
- Assertion fragment (*assert*): means that any sequence not explicitly designated as an operand of the assert fragment is invalid

The mostly used combined fragments are: the alternative fragment, the option fragment and the loop fragment.

The technique presented by Takkouch (2006) defines a test selection algorithm based on interaction overview diagrams. This algorithm (*IOTestCaseSelection*) generates a set of change SDs, by examining the SDs with the same signatures in IO and IO', and noting the similarities or differences between them. According to his algorithm, the sequence diagrams are compared lifeline by lifeline, the lifelines are compared by the messages and the messages order.

While this algorithm selects most of the modifications, it fails to identify changes made in the operand on the combined fragment level. In order to cope with this setback, an amendment should affect this algorithm. This algorithm must take into considerations all modification to the combined fragment, since the change in the fragment will cause a change in integration level tests. A change in the operand may cause the test case path to fail.

```
Algorithm: IOTestCaseSelection.
Input:     T=Set of integration level tests.
           IO=Original interaction overview diagram.
           IO'=Updated interaction overview diagram.
Output:    T'=Set of tests selected from T for retest.
Description: The algorithm generates a set of integration test cases to be selected for
             retest based on their traversal of a changed section in the IO diagram
SystemTestSelection(IO,IO',T):T'
  begin
    T'=∅  --set of all tests from T marked for retest based on IO changes
    T"=∅ --set of all tests from T marked as candidates
    ChangedSD=∅ --set of all changed sequence diagrams

    --compare the SDs of each interaction overview diagram
    ∀ sd ∈ IO having the same signature as sd' ∈ IO'
      begin
      ∀ ll ∈ sd ∪ sd'
        if ll does not exits in either sd or sd' then
          --mark sd as changed
          ChangedSD=ChangedSD ∪ {sd}
        else
          begin
          if sd.ll and sd'.ll have different messages or message order
          --mark sd as changed
          ChangedSD=ChangedSD ∪ {sd}
          end
      ∀ cf ∈ sd ∪ sd'
        if cf does not exits in either sd or sd' then
          --mark sd as changed
          ChangedSD=ChangedSD ∪ {sd}
        else
          begin
          if sd.cf and sd'.cf have different operands
          --mark sd as changed
          ChangedSD=ChangedSD ∪ {sd}
          end
        end

    --traverse the IO diagram and generate a set of test cases for retest
    --and another as candidates for refinement in SDBasedReduction
    (T',T")=IOBasedClassification(IO, IO', T, ChangedSD)
    T'=T' ∪ SDBasedReduction(T, T", ChangedSD)
  end
```

Figure 3.4 Improved IOTestCaseSelection algorithm

The cost of this algorithm was not affected by the modification made on the combined fragment level. The comparison of the SDs is O(sd * a). sd represents the number of sequence diagram in IO, a is the average number of sequence diagrams in the SDs.

The algorithm can now procure from the new modification made to the UML 2.0 sequence diagrams, it will take into consideration all the modifications and not only the change made on the messages or messages sequences. This new improvement will certainly result in an increase in locating the tests that will cause the modified application to produce a different output, thus exposing more faults inside the system.

Chapter 4

Framework for Analyzing Regression Testing Technique

In order to track the modification made to the system, we first hold the original version of the class and the interaction overview diagrams before modification, presuming a suitable correlation, involving changes in the application's source code and the application's UML design, has been kept. Interaction overview diagrams enclose other type of activity diagram, this algorithm only focus on the sequence diagrams, meaning that a copy of the sequence diagrams before update should also be kept.

We will consider an initial set T $=\{t_1, t_2, \ldots . t_N\}$ of N test cases used in the initial development of the program; this set will be saved prior to our regression testing. The test cases are determined by structural (glass box) testing techniques; in particular the programs are modeled using the Unified Modeling Language (UML). Class diagrams and interaction overview diagrams are applied to pick the unit and the integration tests from the original test suite.

After a program has been modified, the algorithm will select a subset of test cases denoted as R with the objective of providing confidence that no adverse effects have been caused by these modifications.

4.1 Test Cases Selected

This first criterion returns the numbers of test cases in *R* selected by the technique from the set of test cases *T*, to be re-executed for regression testing. The output would be a set

of unit test and integration level test. $\#R$ will denote this number of test cases and will be presented as a percentage value from N, the cardinality of T.

In order to express the use of the refinement that occurs at the end of the technique, we will denote by $\#R_1$ the number of test cases selected for rerun before the refinement step, and by $\#R_2$ the number after it.

4.2 Inclusiveness

The *Inclusiveness* measures to which degree the technique selects tests that will cause the modified program to produce different output than the original program, and thereby exposing the faults caused by the modifications (Rothermel and Harrold, 1996). This criterion aims at selecting the modification-revealing (*mr*) tests. When a test case does include modification-revealing tests, and the regression testing method selects s of these tests (denoted by *smr*), the inclusiveness will be given by the following formulas:

$$\text{Inclusiveness} = 100\left(\frac{smr}{mr}\right) \quad \text{for mr} \neq 0$$

$$\text{Inclusiveness} = 100\% \qquad \text{for mr} = 0$$

Equation 4. 1 Inclusiveness formula

Many conclusions can be drawn from the inclusiveness rate, the method is considered safe when it selects all the modification revealing tests. When this technique fails to accomplish full coverage, it is considered non safe.

4.3 Precision

Precision determines the ability of a technique to avoid choosing tests that won't cause the modified program to produce different output than the original program. In other words, precision assess the capacity of a technique to exclude tests that are non-modification-revealing (*nmr*). The excluded tests that are non-modification-revealing are denoted by *onmr* where o stands for omitted. The precision rate is given by the following formulas:

$$\text{Precision} = 100\left(\frac{onmr}{nmr}\right) \quad \text{for nmr} \neq 0$$

$$\text{Precision} = 100\% \quad \text{for nmr} = 0$$

Equation 4. 2 Precision formula

Precision can be useful to prove that a technique is not accurate by proving a case where this method selects tests that are non-modification-revealing. It is also found valuable when comparing different test selection techniques.

Chapter 5
Case Studies

In order to evaluate this technique, we will test the technique against three applications: a Library System, an ATM application that handles both Automated Teller Machine (A.T.M.) and Point-of-Sale (P.O.S) transactions, and a Learning Center application that handles courses registration and an exam system. For each application we present three versions meaning that each application encountered three modifications (in additional to the original version). A separate set of UML models is provided for each version; the diagrams presented are the ones before and after update. Each application will be described and then the changes that were performed on each modification will be presented along with the results of the regression testing technique. The modifications made were realistic, logical changes.

In this section the UML-based regression testing method is applied to all nine applications, the results are shown in the following sections. The full technique is only presented for the first version, for the other technique only the class and interaction overview diagrams are shown. Further information about these case studies are documented the Appendixes at the end of this report.

5.1 Library System

The library system is considered simple compared with real systems. However, it has proven enough complexity to allow several problems in the area of testing.

The application is a support system for a library. This library provides items (books and magazines) to borrowers listed in the system. The system guaranties that only registered users are allowed to borrow items from the library, it also insures that the borrowers can only performs services that are associated with themselves, same goes for librarians. Borrowers can only search for books or magazines, all the other interactions are done through the librarian. Old books and old magazines are taken out of the library when they are out of date, new items are regularly purchased. Once a book is purchased, users can check its availability and can reserve a copy if the book was unavailable. A user can also cancel a previous reservation thus allowing the person behind in the list queue to take his place.

The original system consists of 8 classes, 29 attributes, 74 methods and 12 relationships (2 of which are inheritance relationships). This system is made up of 3 main subsystems: The Return Item, Borrow Item and the Cancel Reservation subordinate system.

5.1.1 Library System v2

The first change on this application was adding a new functionality to the system. A new fine system was conceived, all late borrowers are now subject to fine. The amount differs according to the item type (book or magazine) and is computed by the number of days, meaning that more the borrower exceeded his due date, the more fees he will have to pay. If a user ran late on returning an item, he will have to pay the penalties due and a receipt will be issued to the borrower.

5.1.1.1 Application Changes

Changes in the Class Diagram

The original class diagram of the Library System was presented in Figure 3.1, it illustrates the classes constituting the system.

For the new enhancement to take place, several additions were made on the class level, a date attribute was created to save the item due date denoted as dueDate, In order to read and write to this attribute, GetDueDate() and SetDueDate() were added to the Loan class. The dudeDate is filled by adding the appropriate lendingTime, depending on the item type, and adding the value to the current date of borrowing. The amountPerDay attribute was added to both BookTitle and MagazineTitle Classes. These two attributes allow saving the amount of money due for each day exceeding the dueDate. Two methods were added to the Title class, the first is GetAmountPerDay() and the second is SetAmountPerDay(), these two methods are used to manipulate the amountPerDay attribute mentioned earlier. Another modification was done on the method level, it affected the GetLendgingTime() method inside the Title class. The resulting Class Diagram after the modification (v2) is shown in Figure 5.1

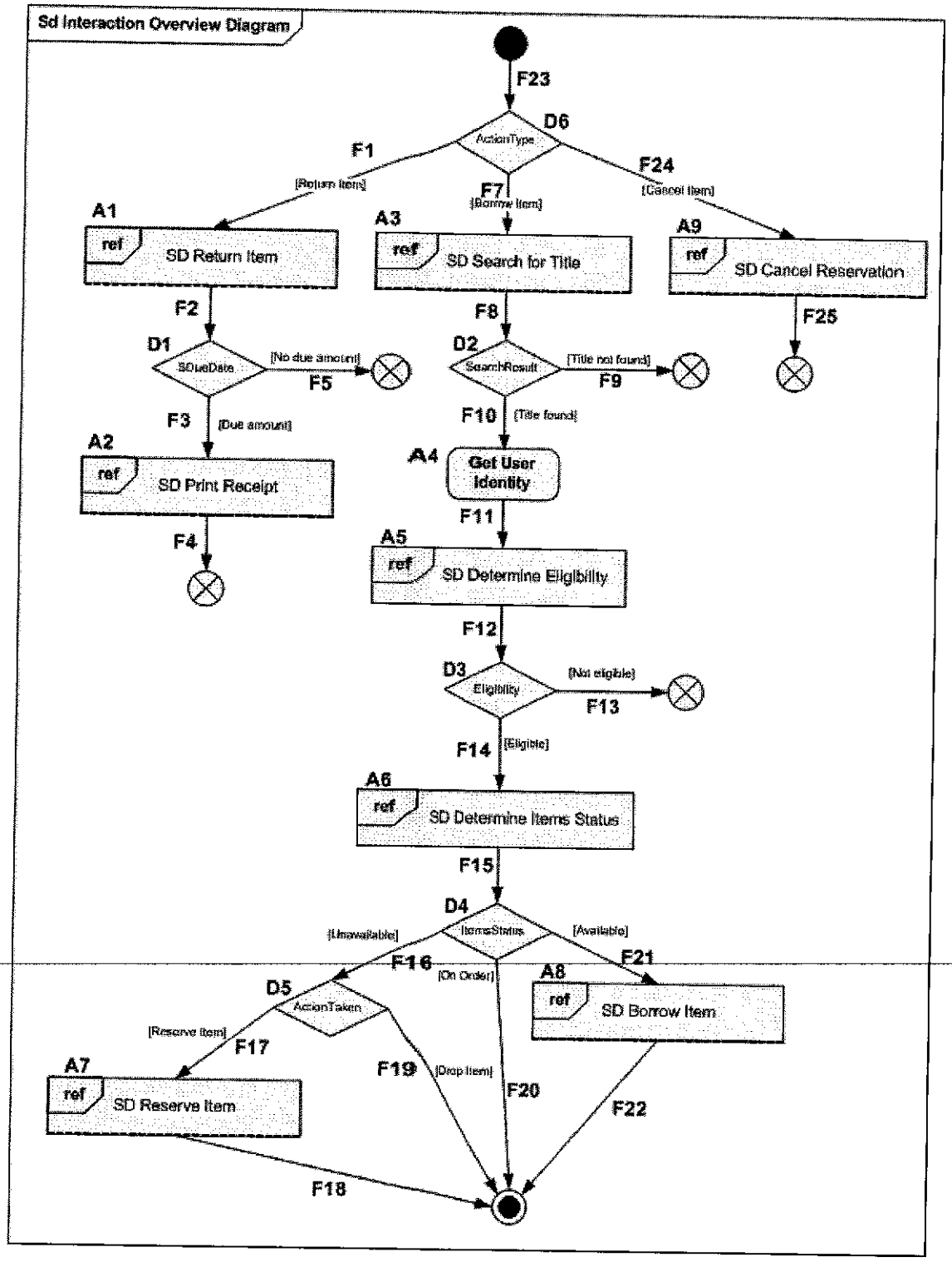

Figure 5.4 Library System (v2) interaction overview diagram

5.1.1.2 Original Set of Test Cases

The original set of tests is divided into two parts: the integration level test and the unit test cases. The integration level test cases with their respective path in the interaction overview diagram (T.IO) are shown in the Table 5.2. The path of each integration test case inside the sequence diagrams (T.SD) is presented in the Table 5.3. This table presents the path that each test case undergo while passing in the sequence diagrams, each test case will have multiple entries since each path pass through several sequence diagram. Finally, the unit test cases for the methods of the Library System is shown in Table 5.4. T.SD

Table 5.2 Library System (v1) integration test cases in the IO diagram (T.IO)

Test Case	Description	Test Path
T1	Valid Return Item	F23, D6, F1, A1, F6.
T2	Valid Reserve Item	F23, D6, F7, A3, F8, D2, F10, A4, F11, A5, F12, D3, F14, A6, F15, D4, F16, D5, F17, A7, F18.
T3	Invalid Borrow Item, item unavailable, drop item	F23, D6, F7, A3, F8, D2, F10, A4, F11, A5, F12, D3, F14, A6, F15, D4, F16, D5, F19.
T4	Invalid Borrow Item, item unavailable, on order	F23, D6, F7, A3, F8, D2, F10, A4, F11, A5, F12, D3, F14, A6, F15, D4, F20.
T5	Valid Borrow Item	F23, D6, F7, A3, F8, D2, F10, A4, F11, A5, F12, D3, F14, A6, F15, D4, F21, A8, F22.
T6	Invalid Borrow Item, title not found	F23, D6, F7, A3, F8, D2, F9.
T7	Invalid Borrow Item, not eligible, exceeds number of loans	F23, D6, F7, A3, F8, D2, F10, A4, F11, A5, F12, D3, F13.
T8	Invalid Borrow Item, not eligible, user not registered	F23, D6, F7, A3, F8, D2, F10, A4, F11, A5, F12, D3, F13.
T9	Valid Cancel Reservation	F23, D6, F24, A9, F25.

Table 5.3 Library System (v1) integration test cases in the sequence diagram (T.SD)

Test Case	SD	Path
T1	A1	Librarian.DoReturnItem, Title.FindOnISBN, Title.FindOnISBN.Return, Item.GetItemID, Item.GetItemID.Return, Item.GetLoan, Item.GetLoan.Return, Loan.GetBorrower, Loan.GetBorrower.Return, Borrower.RemoveLoan, Borrower.RemoveLoan.Return
T2	A3	Title.FindOnName, Title.IterateTitle, Title.FindOnName.Return, [opt: Find by author], Title.FindOnAuthor, Title.IterateTitle, Title.FindOnAuthor.Return, [alt: Title not found], Borrower.Report
T2	A5	Borrower.CheckEligibility, Borrower.GetBorrower, Borrower.GetNoOfLoans, Borrower.CheckEligibility.Return, [alt: No Loans>=3], Borrower.IsRegistered, Borrower.IsRegistered.Return, [alt: Registered=false], Borrower.Report
T2	A6	Title.FindOnISBN, Title.GetNoItems, Title.GetNoReservations, [alt: No Items Found], [loop: more items], Item.GetItemID, Item.IsBorrowed, Item.GetItemID.Return, Title.GetItem, Item.GetStatus, Item.GetStatus.Return, [alt: status=available], [alt: no available items], Borrower.Report
T2	A7	Librarian.DoReserveItem, Reservation.Reservation, Reservation.SetTitle, Reservation.Reservation.Return, Borrower.AddReservation, Borrower.AddReservation.Return
T3	A3	Title.FindOnName, Title.IterateTitle, Title.FindOnName.Return, [opt: Find by author], Title.FindOnAuthor, Title.IterateTitle, Title.FindOnAuthor.Return, [alt: Title not found], Borrower.Report
T3	A5	Borrower.CheckEligibility, Borrower.GetBorrower, Borrower.GetNoOfLoans, Borrower.CheckEligibility.Return, [alt: No Loans>=3], Borrower.IsRegistered, Borrower.IsRegistered.Return, [alt: Registered=false], Borrower.Report
T3	A6	Title.FindOnISBN, Title.GetNoItems, Title.GetNoReservations, [alt: No Items Found], [loop: more items], Item.GetItemID, Item.IsBorrowed, Item.GetItemID.Return, Title.GetItem, Item.GetStatus, Item.GetStatus.Return, [alt: status=available], [alt: no available items], Borrower.Report
T4	A3	Title.FindOnName, Title.IterateTitle, Title.FindOnName.Return, [opt: Find by author], Title.FindOnAuthor, Title.IterateTitle, Title.FindOnAuthor.Return, [alt: Title not found], Borrower.Report
T4	A5	Borrower.CheckEligibility, Borrower.GetBorrower, Borrower.GetNoOfLoans, Borrower.CheckEligibility.Return, [alt: No Loans>=3], Borrower.IsRegistered, Borrower.IsRegistered.Return, [alt: Registered=false],

		Borrower.Report
T4	A6	Title.FindOnISBN, Title.GetNoItems, Title.GetNoReservations, [alt: No Items Found], Title.FindOnISBN.Return, Borrower.Report
T5	A3	Title.FindOnName, Title.IterateTitle, Title.FindOnName.Return, [opt: Find by author], Title.FindOnAuthor, Title.IterateTitle, Title.FindOnAuthor.Return, [alt: Title not found], Borrower.Report
T5	A5	Borrower.CheckEligibility, Borrower.GetBorrower, Borrower.GetNoOfLoans, Borrower.CheckEligibility.Return, [alt: No Loans>=3], Borrower.IsRegistered, Borrower.IsRegistered.Return, [alt: Registered=false], Borrower.Report
T5	A6	Title.FindOnISBN, Title.GetNoItems, Title.GetNoReservations, [alt: No Items Found], [loop: more items], Item.GetItemID, Item.IsBorrowed, Item.GetItemID.Return, Title.GetItem, Item.GetStatus, Item.GetStatus.Return, [alt: status=available], Title.FindOnISBN.Return, [alt: available items], Borrower.Report
T5	A8	Librarian.DoBorrowItem, Title.FindTitle, Title.GetTitle, Title.FindTitle.Return, Item.FindItem, Item.FindItem.Return, Title.GetLendingTime, Title.GetLendingTime.Return, Loan.Loan, Item.SetLoan, Item.SetLoan.Return, Borrower.AddLoan, Borrower.AddLoan.Return, Loan.Loan.Return
T6	A3	Title.FindOnName, Title.IterateTitle, Title.FindOnName.Return, [opt: Find by author], Title.FindOnAuthor, Title.IterateTitle, Title.FindOnAuthor.Return, [alt: Title not found], Borrower.Report
T7	A3	Title.FindOnName, Title.IterateTitle, Title.FindOnName.Return, [opt: Find by author], Title.FindOnAuthor, Title.IterateTitle, Title.FindOnAuthor.Return, [alt: Title not found], Borrower.Report
T7	A5	Borrower.CheckEligibility, Borrower.GetBorrower, Borrower.GetNoOfLoans, Borrower.CheckEligibility.Return, [alt: No Loans>=3], Borrower.Report
T8	A3	Title.FindOnName, Title.IterateTitle, Title.FindOnName.Return, [opt: Find by author], Title.FindOnAuthor, Title.IterateTitle, Title.FindOnAuthor.Return, [alt: Title not found], Borrower.Report
T8	A5	Borrower.CheckEligibility, Borrower.GetBorrower, Borrower.GetNoOfLoans, Borrower.CheckEligibility.Return, [alt: No Loans>=3], Borrower.IsRegistered, Borrower.IsRegistered.Return, [alt: Registered=false], Borrower.Report
T9	A9	Librarian.DoCancelReservation, Borrower.FindOnLastName, Borrower.FindOnLastName.Return, Reservation.Reservation, Reservation.Reservation.Return, Title.GetTitle,

	Title.GetTitle.Return, Title.GetReservation, Title.GetReservation.Return, Borrower.RemoveReservation, Borrower.RemoveReservation.Return

Table 5.4 Library System (v1) unit test cases (UT)

Test Case	Method
UT1	Loan.Loan
UT2	Loan.GetBorrower
UT3	Loan.GetTitleName
UT4	Loan.GetItem
UT5	Loan.GetItemID
UT6	Item.Item
UT7	Item.FindItem
UT8	Item.GetTitleName
UT9	Item.GetItemID
UT10	Item.SetLoan
UT11	Item.GetLoan
UT12	Item.IsBorrowed
UT13	Item.SetStatus
UT14	Item.GetStatus
UT15	Title.Title
UT16	Title.FindTitle
UT17	Title.FindOnName
UT18	Title.FindOnAuthor
UT19	Title.FindOnISBN
UT20	Title.IterateTitle
UT21	Title.GetTitle
UT22	Title.GetTitle
UT23	Title.GetAuthor
UT24	Title.GetISBN
UT25	Title.GetType
UT26	Title.SetTitle
UT27	Title.SetAuthor
UT28	Title.SetNumber
UT29	Title.SetType
UT30	Title.SetBanned
UT31	Title.GetBanned
UT32	Title.GetLendingTime
UT33	Title.GetNoItems
UT34	Title.AddItem
UT35	Title.RemoveItem
UT36	Title.GetItem
UT37	Title.GetNoReservations

UT38	Title.GetReservation
UT39	Librarian.Librarian
UT40	Librarian.CheckIn
UT41	Librarian.CheckOut
UT42	Librarian.DoReturnItem
UT43	Librarian.DoCancelItem
UT44	Librarian.DoBorrowItem
UT45	Librarian.DoReserveItem
UT46	Reservation.Reservation
UT47	Reservation.SetTitle
UT48	Reservation.GetBorrower
UT49	Borrower.Borrower
UT50	Borrower.GetBorrower
UT51	Borrower.IterateBorrower
UT52	Borrower.FindOnLastName
UT53	Borrower.AddLoan
UT54	Borrower.RemoveLoan
UT55	Borrower.GetNoLoans
UT56	Borrower.AddReservation
UT57	Borrower.RemoveReservation
UT58	Borrower.GetNoReservations
UT59	Borrower.GetReservation
UT60	Borrower.IsRegistered
UT61	Borrower.CheckEligibility
UT62	Borrower.Report
UT63	Borrower.SetFirstName
UT64	Borrower.SetLastName
UT65	Borrower.SetAddress
UT66	Borrower.SetCity
UT67	Borrower.SetZip
UT68	Borrower.SetState
UT69	Borrower.GetFirstName
UT70	Borrower.GetLastName
UT71	Borrower.GetAddress
UT72	Borrower.GetCity
UT73	Borrower.GetZip
UT74	Borrower.GetState

5.1.1.3 Test Case Selection

Selection Based on the Class Diagram

As section 3.2.1 indicates, the changes that affected the class diagram is the method GetLendingTime(), the comparison is done by evaluating the signature of the two class diagrams. The set of changed methods, by traversing the *GenerateChangedMethods* algorithm is M={GetLendingTime()}.

The sequence diagram referencing M, by running the *CDIntegrationTestSelection* algorithm is A8: SD Borrow Item (Figure 3.3). By going through the list of integration test cases path in the sequence diagrams (T.SD), the integration test cases traversing A8 on the I.O level is T5.

CDUnitTestSelection algorithm selects the unit test cases from both directly and indirectly changed methods. The unit test case executing M directly being UT32, the indirectly affected methods are extracted by running the transitive closure for each method in each sequence diagram. Thus, the indirectly changed methods affect by the change of GetLendingTime() are: DoBorrowItem(), FindTitle(), GetTitle() and FindItem(). The unit test cases testing these methods (Table 5.4) are respectively: UT44, UT16, UT22 and UT7.

The selected test cases based on changes in the sequence diagram are: T5, UT7, UT16, UT22, UT32 and UT44.

Selection Based on the Interaction Overview Diagram

After comparing the interaction overview diagram before and after update, the set of changed SDs is A1: SD Return Item (Figure 5.2 and Figure 5.3).

The *IOBasedClassification* algorithm will output the set of test cases picked from T for retest and the test cases selected from T as possible candidates for retest (for further analysis and reduction). The direct change on the IO diagram was the addition of F2, F3, F4, F5 and the deletion of F6. D1 was also added. The set of test cases from T.IO selected for retest is the one traversing A1 which is T1.

The set of candidate test cases going through A1 is T1.

The final step in this technique is making sure if the past list of these candidates in the set of changed SDs in T.SD is still valid. The results are:

- (T1, A1)=Invalid Path. (*Librarian.DoReturnItem, Title.FindOnISBN, Title.FindOnISBN.Return, Item.GetItemID, Item.GetItemID.Return, Item.GetLoan, Item.GetLoan.Return, Loan.GetBorrower, Loan.GetBorrower.Return, Borrower.RemoveLoan, Borrower.RemoveLoan.Return*).The old path skips the GetDueDate() method, thus failing to remain the same since a new step has been added to the path.

Based on the changes in IO, the selected integration test case for regression testing is T1. This technique has selected five unit test and two integration test; these results were selected out of 83 test cases.

5.1.2 Library System v3

The second logical modification requires an account for each borrower. The system now has to support a separate account for each borrower. If a borrower, for some reason, does not pay the amount owed when he returns his overdue book, he can added in his own account for later payment. This new feature affected another procedure, the next time this user wants to borrow an item; his eligibility will be affected if he has an amount of money to be paid. He will not have the ability of borrowing any item before reimbursing the amount due to the library. If he chose to pay the amount due, a receipt will be printed.

The change on the class level was done by adding amountDue attribute and two methods to manipulate this attribute: GetAmountDue() and SetAmountDue(). Obviously, these additions were made to the Borrower class where the amount due is kept in his account for later use. The changes are depicted in the class diagram (v3) in Figure 5.5.

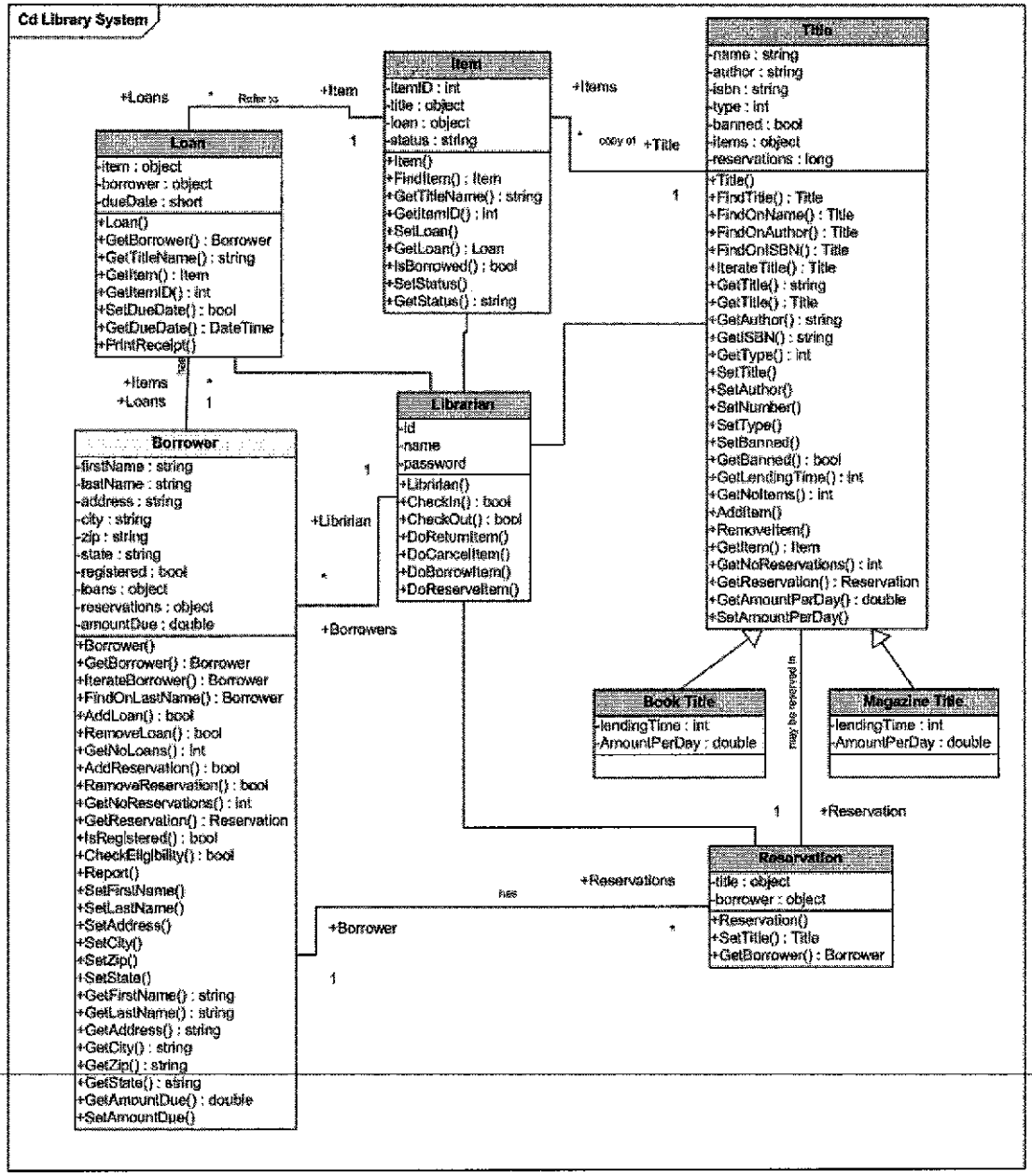

Figure 5.5 Library System (v3) class diagram

This version of the library system resulted in the addition of 1 attributes and 2 methods, Table 5.5 summarizes the changes made to the system:

Table 5.5 Library System classes change results (v2-v3)

	Total *before update*	**Added**	**Modified**	**Deleted**	**Total** *after update*
Attributes	32	1	0	0	33
Methods	79	2	0	0	81
Relationships	12	0	0	0	12
Classes	8	0	1	0	8

The changes in the system resulted in the addition of one decision (*PaymentMethod*) on the IO level. The update resulted in the addition of 3 flows: F5, F6 and F7. One flow was deleted: F4. The figure below (Figure 5.6) illustrates the system with the new modifications.

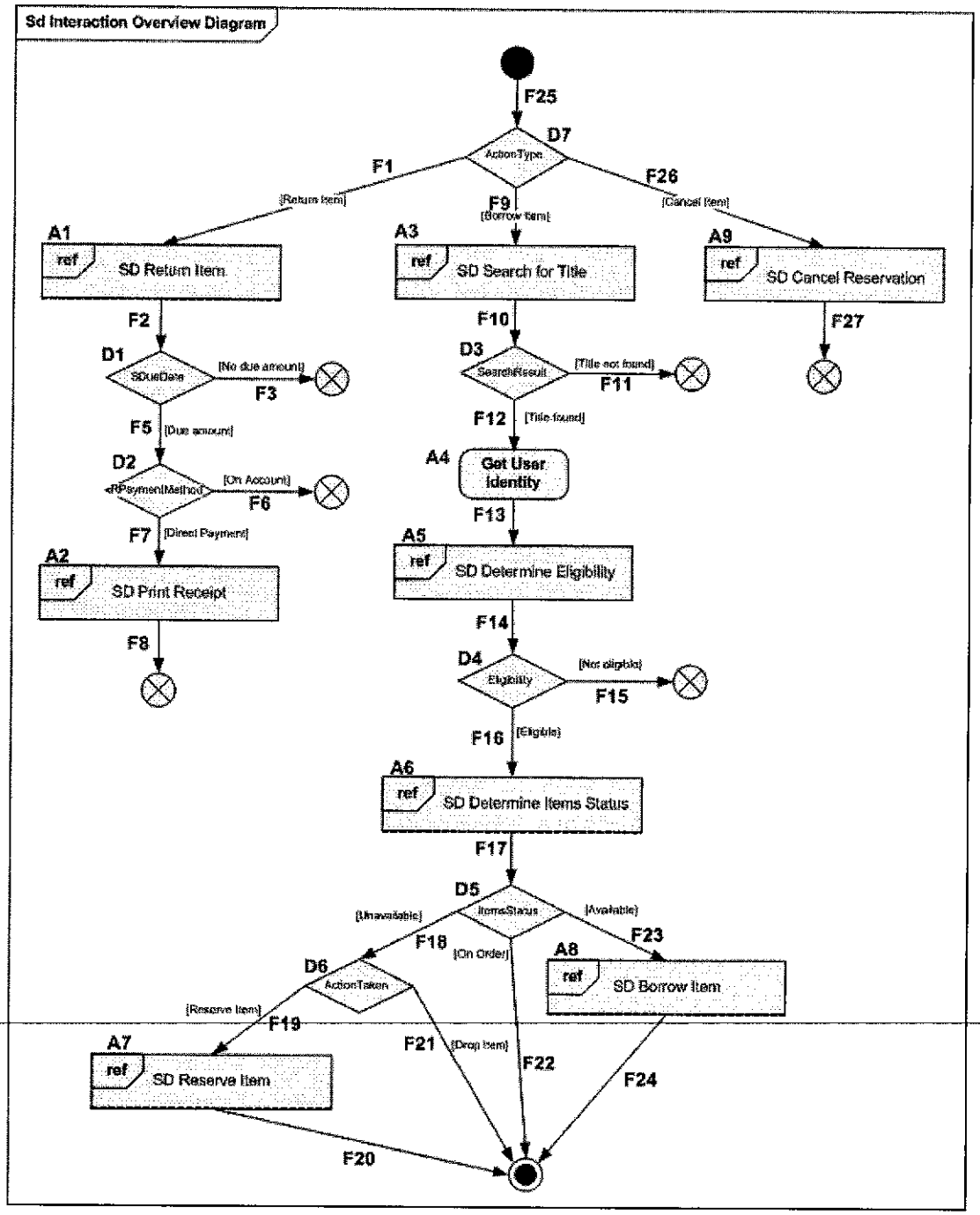

Figure 5.6 Library System (v3) interaction overview diagram

5.1.3 Library System v4

The last modification impose more restriction on the borrowing process, due to overbooking in the library and the insufficient number of available books, the administration of the library decide to decrease the number of allowed books to be borrowed down to one item per borrower. Another enhancement was added to the system to speed up the borrowing lifecycle, after an item is return and in case it has a reservation, an email notification is now sent to the borrower to collect his reservation.

The change on the class diagram level results the addition of a new attribute: email. This string attribute was accompanied by three methods: EmailBorrower(), GetEmail() and SetEmail(). The change on this level was done for the borrower notification feature; the number of allowed book modification only caused changes on the sequence diagram level. The class diagram after the modification is show in Figure 5.7, as seen in the class diagram; the modification only affected the Borrower class.

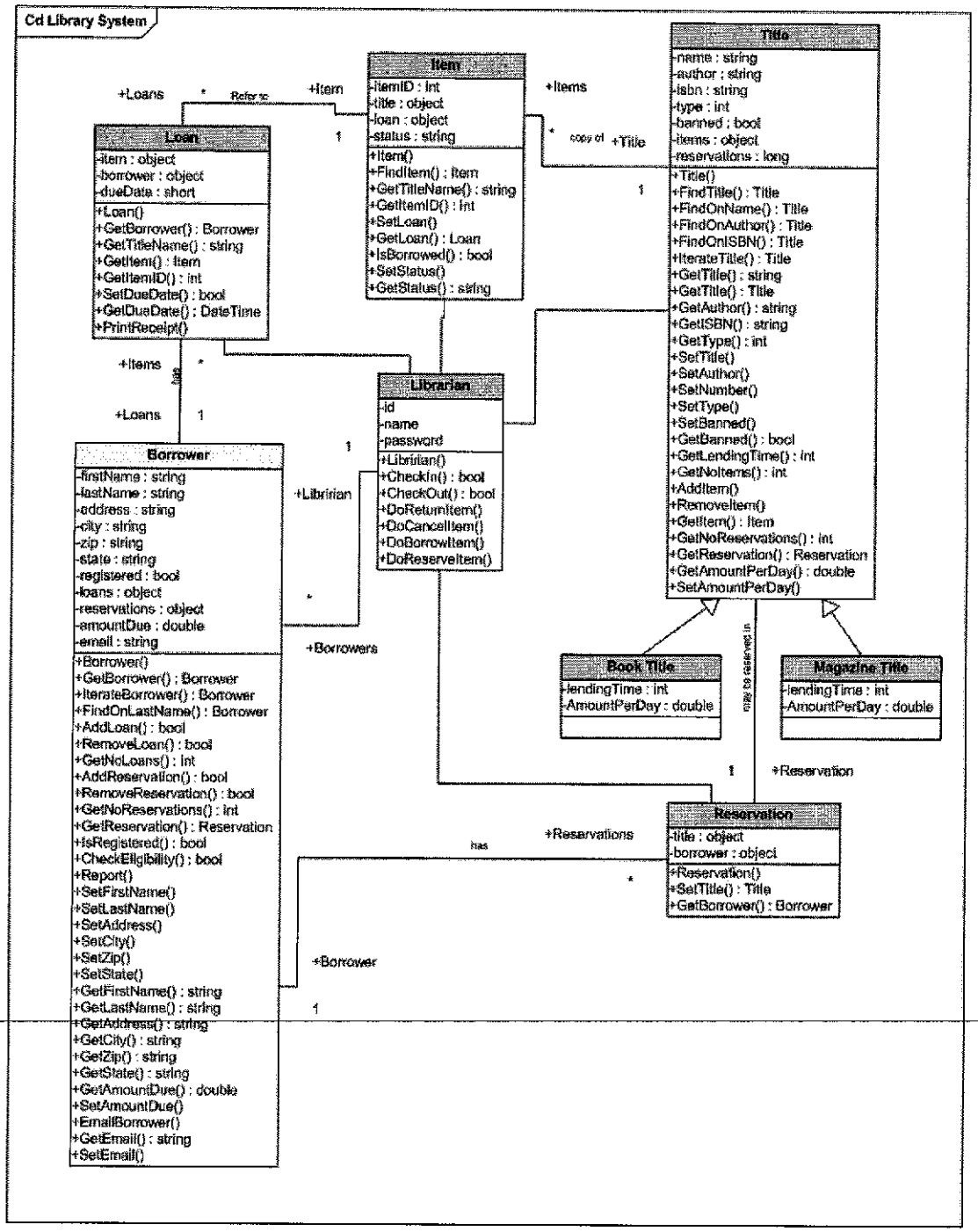

Figure 5.7 Library System (v4) class diagram

The result modification on the class diagram is shown below:

Table 5.6 Library System classes change results (v3-v4)

	Total *before update*	Added	Modified	Deleted	Total *after update*
Attributes	33	1	0	0	34
Methods	81	3	0	0	84
Relationships	12	0	0	0	12
Classes	8	0	1	0	8

In this version, no modifications were made to the Interaction Overview diagram. All modifications in the Sequence Diagrams were done internally, thus transparent to the Interaction Overview Diagram. The IO before update is shown in Figure 5.6.

5.2 ATM Application

The Automated Teller Machine (ATM) application is considered a critical application since it requires precision to handle the money or reservation transactions.

The ATM are distributed at all branches and linked to the main office's mainframe which stores the data of all customer accounts. The ATM allows the customers of the bank to access the details of their accounts; they are allowed to view their balance and cash in money, if they have enough credit. The customers can also use their ATM cards at the Point of Sales (POS) to buy goods or even use it to reserve an amount of money. When the card is purchased, the customer is given a 4-digit Pin code which will be used each time he wants to make a transaction

The original version of this application is made with 11 classes, 23 attributes, 40 methods and 12 relationships. The system is divided into 2 subsystems: the Automated Teller Machine (ATM) and the Point of Sale (POS).

The original class diagram of the ATM application is illustrated in Figure 5.8

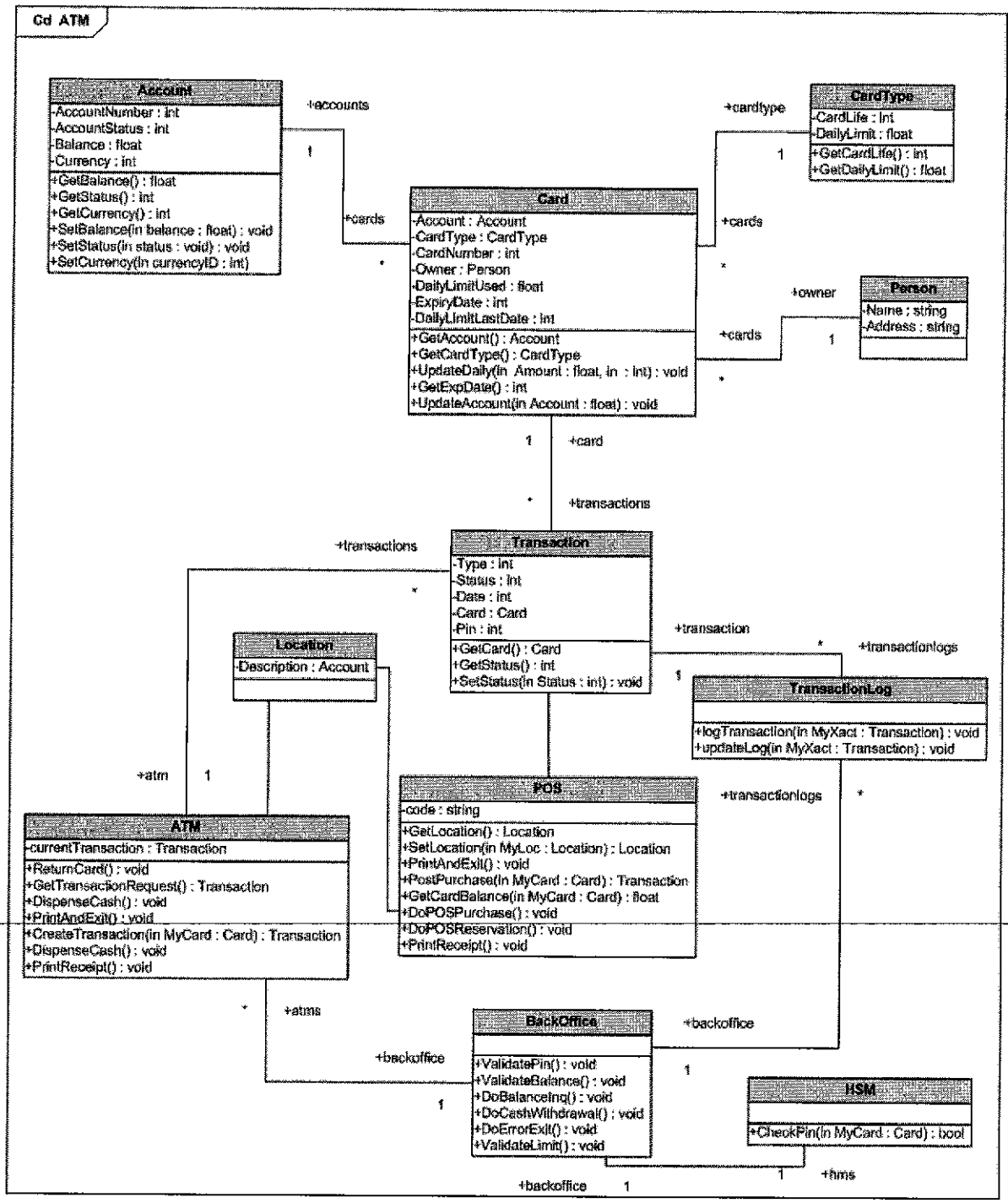

Figure 5.8 ATM (v1) class diagram

The interaction overview of this system (Figure 5.9) is composed of 6 referenced sequence diagrams: three for the ATM transactions and the others for the POS transactions. Two actions also exists one for each transaction type. Finally, the system encloses 7 decisions nodes to indicate conditions.

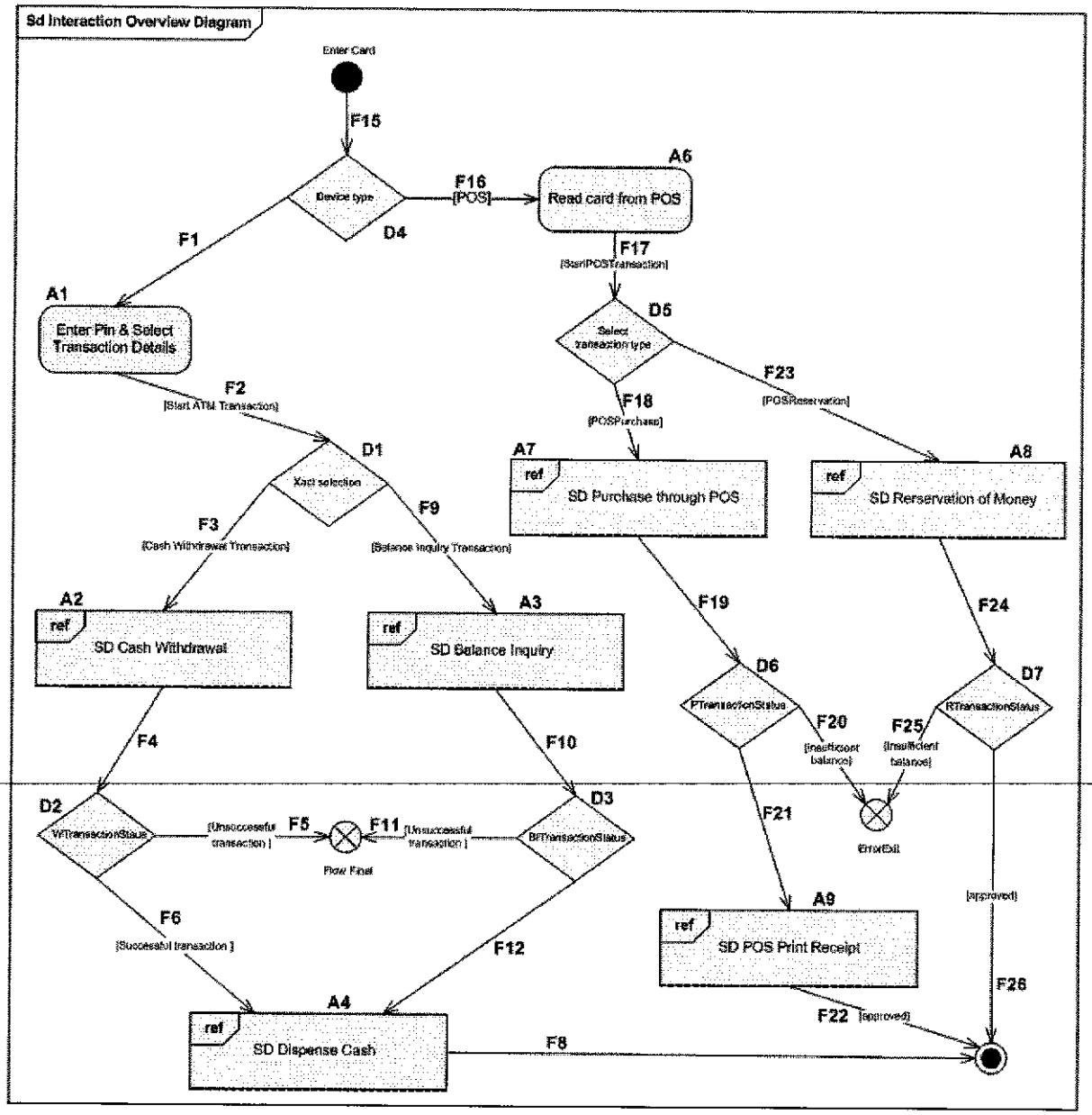

Figure 5.9 ATM (v1) interaction overview diagram

5.2.1 ATM Application v2

The first modification to the system was the addition of a monthly limit restriction for each customer. To cope with this change, 3 attributes were added: monthlyLimitUsed, monthlyLimitLastDate and monthlyLimit. The first is used to save how much of the monthly allowed amount the customer has used, the second stores the last date of the monthly limit entered value and the monthlyLimit saves the maximum allowed amount per month per customer. Two methods were also introduced, one to update the monthly value (UpdateMonthly()) and the second to retrieve this amount (GetMonthlyLimit()). The last change on the method level is done inside the ValidateLimit() method. The class diagram after update (v2) is shown in Figure 5.10.

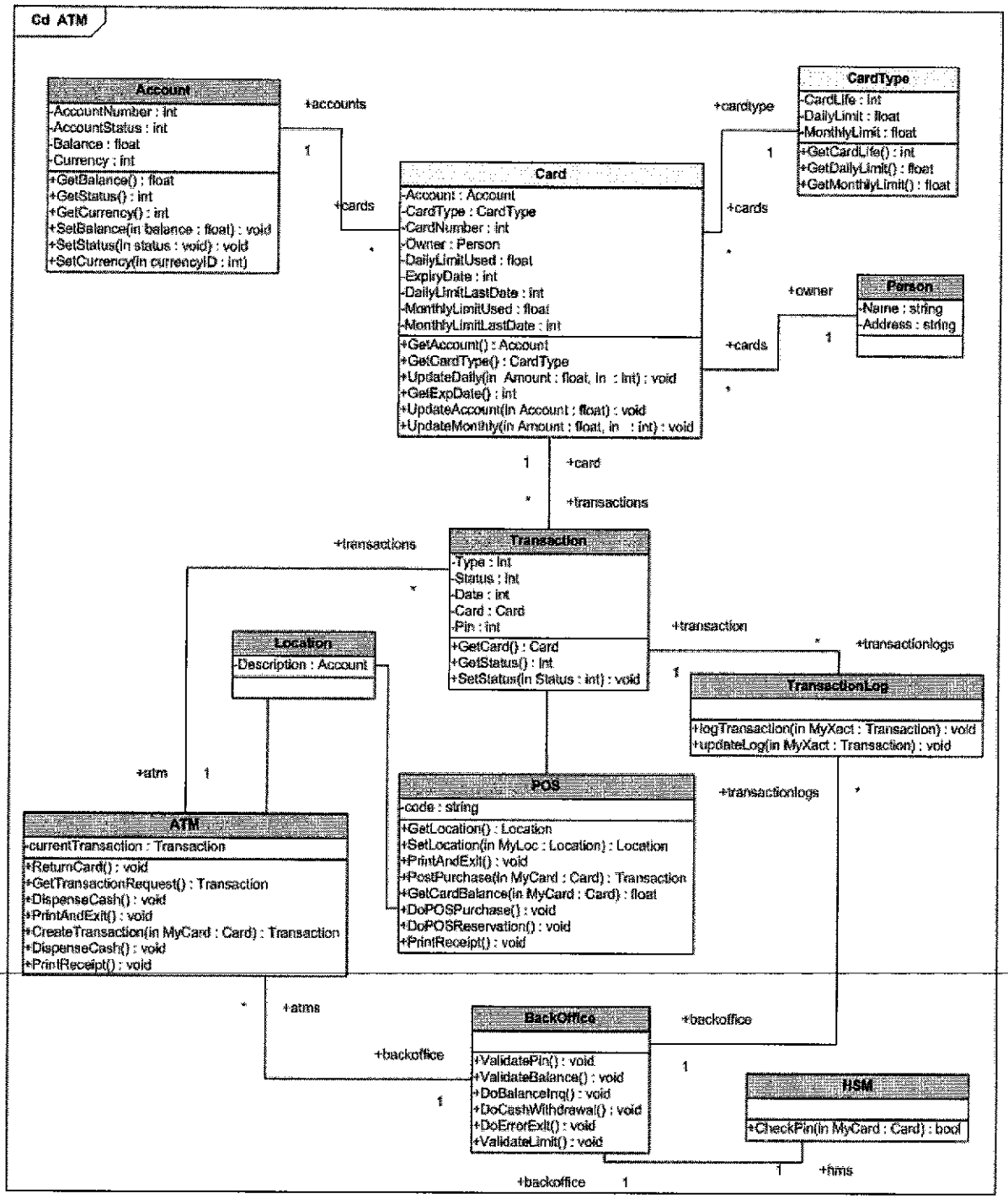

Figure 5.10 ATM (v2) class diagram

The table below (Table 5.7) reviews the changes made to the class diagram:

Table 5.7 ATM classes change results (v1-v2)

	Total *before update*	**Added**	**Modified**	**Deleted**	**Total** *after update*
Attributes	23	3	0	0	26
Methods	40	2	1	0	42
Relationships	13	0	0	0	13
Classes	11	0	3	0	11

The changes on the IO level resulted in the addition of one sequence diagram (SD Print Receipt). The new SD caused the addition of 3 flows: F11, F7 and F12. Two flows were also removed: F6 and F8. Figure 5.11 depicts the interaction overview diagram (v2) after the first modification.

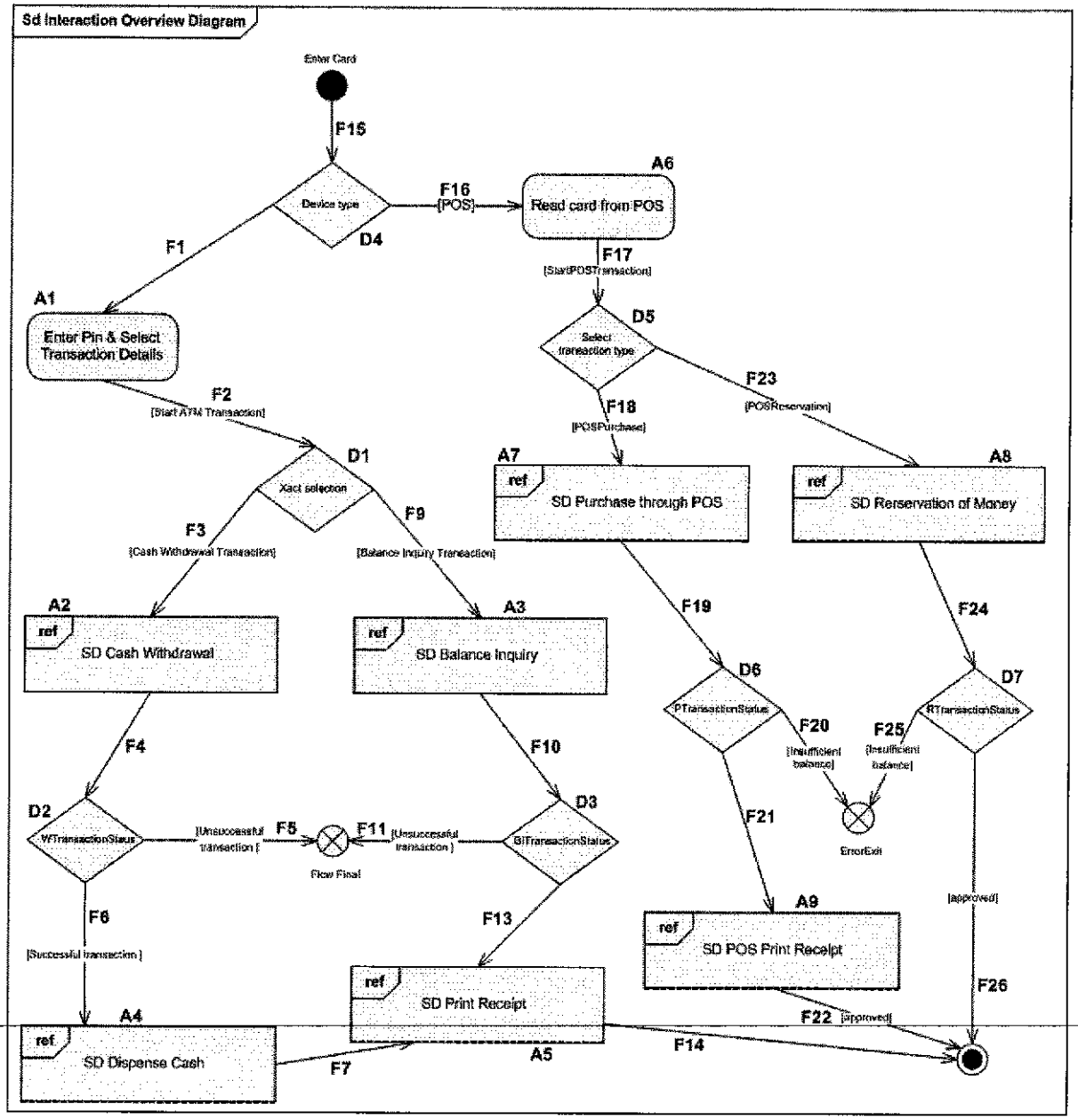

Figure 5.11 ATM (v2) interaction overview diagram

5.2.2 ATM Application v3

Fearing forgery, the information technology committee has decided to enhance their security policy by allowing two erroneous pin retries. This means when a card owner enters more than two invalid pin, his/her card will be held in the ATM machine and has to be pick it up from his bank branch. This logical change translates into the following changes: In the Card class, the PinRetries attribute was added to store the number of times the customer has erroneously tried to enter his pin, another change in this class on the method level is the addition of SetPinRetries() and GetPinRetries() to read and store the pin retries value from the Card class diagram. Another change was imposed to hold the card when the number of pin retries has exceeded its limit. The addition was made to the ATM class with the addition of the HoldCard() method.

The class diagram illustrating this change (v3) is shown in Figure 5.12.

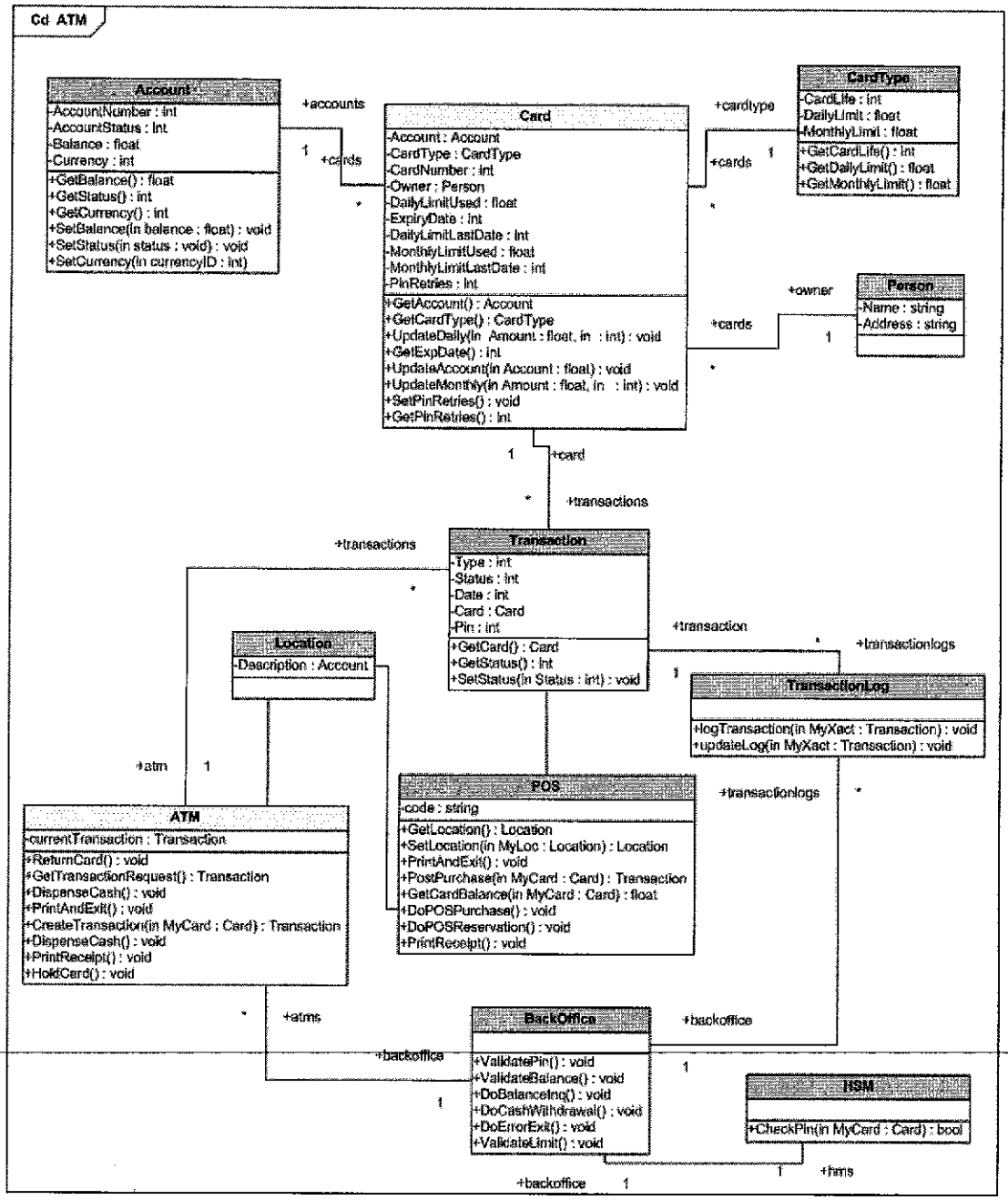

Figure 5.12 ATM (v3) class diagram

The summary of changes on the class level is depicted in Table 5.8:

Table 5.8 ATM classes change results (v2-v3)

	Total *before update*	Added	Modified	Deleted	Total *after update*
Attributes	26	1	0	0	27
Methods	43	3	0	0	46
Relationships	13	0	0	0	13
Classes	11	0	2	0	11

Changes were made inside the SD Cash Withdrawal; since the change was internally, it was transparent to the Interaction Overview Diagram. Figure 5.11 shows the interaction overview diagram before this modification.

5.2.3 ATM Application v4

The third change was imposing a draw limit when reserving money from the POS. This new restriction allows customers to set an upper boundary for the reservation of money. Due to high number of held cards, the administration decided to raise the allowed number of erroneous pin codes by one. A receipt now will also be printed when the customer makes a reservation.

As a result of this change, the ATM class was subject to modifications: the DrawLimit attribute now store the higher bounder for the reservation of money amount. Two methods were also used to operate this attribute: GetDrawLimit() and SetDrawLimit(). The updated class diagram (v4) is shown in Figure 5.13.

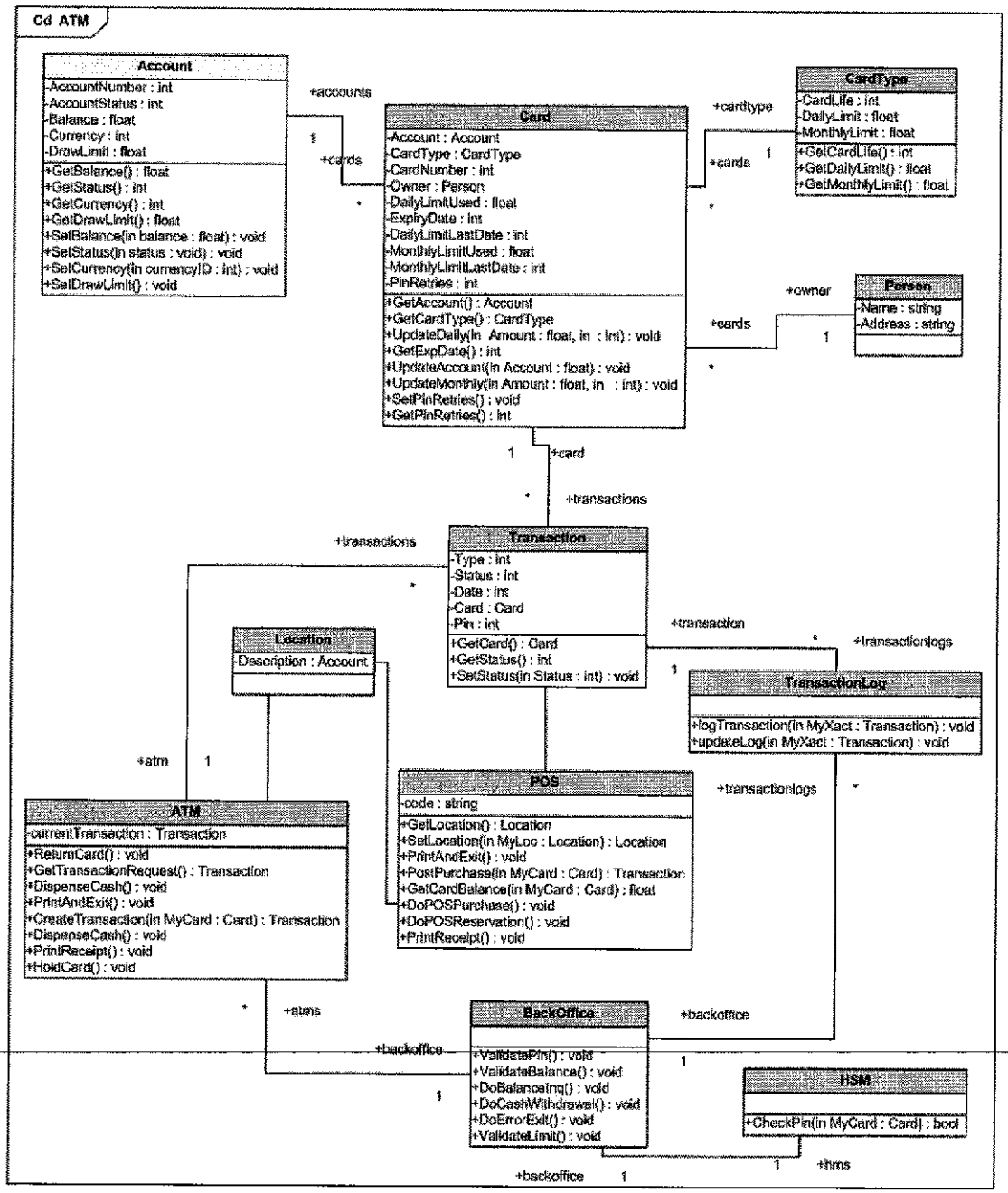

Figure 5.13 ATM (v4) class diagram

The resulting change summary on the class level is shown in Table 5.9

Table 5.9 ATM classes change results (v3-v4)

	Total *before update*	**Added**	**Modified**	**Deleted**	**Total** *after update*
Attributes	27	1	0	0	28
Methods	46	2	0	0	48
Relationships	13	0	0	0	13
Classes	11	0	1	0	11

Changes on the IO level resulted in the addition of one flow: F27 and the deletion of flow F26. Figure 5.14 depicts the interaction overview diagram with the newly introduced changes.

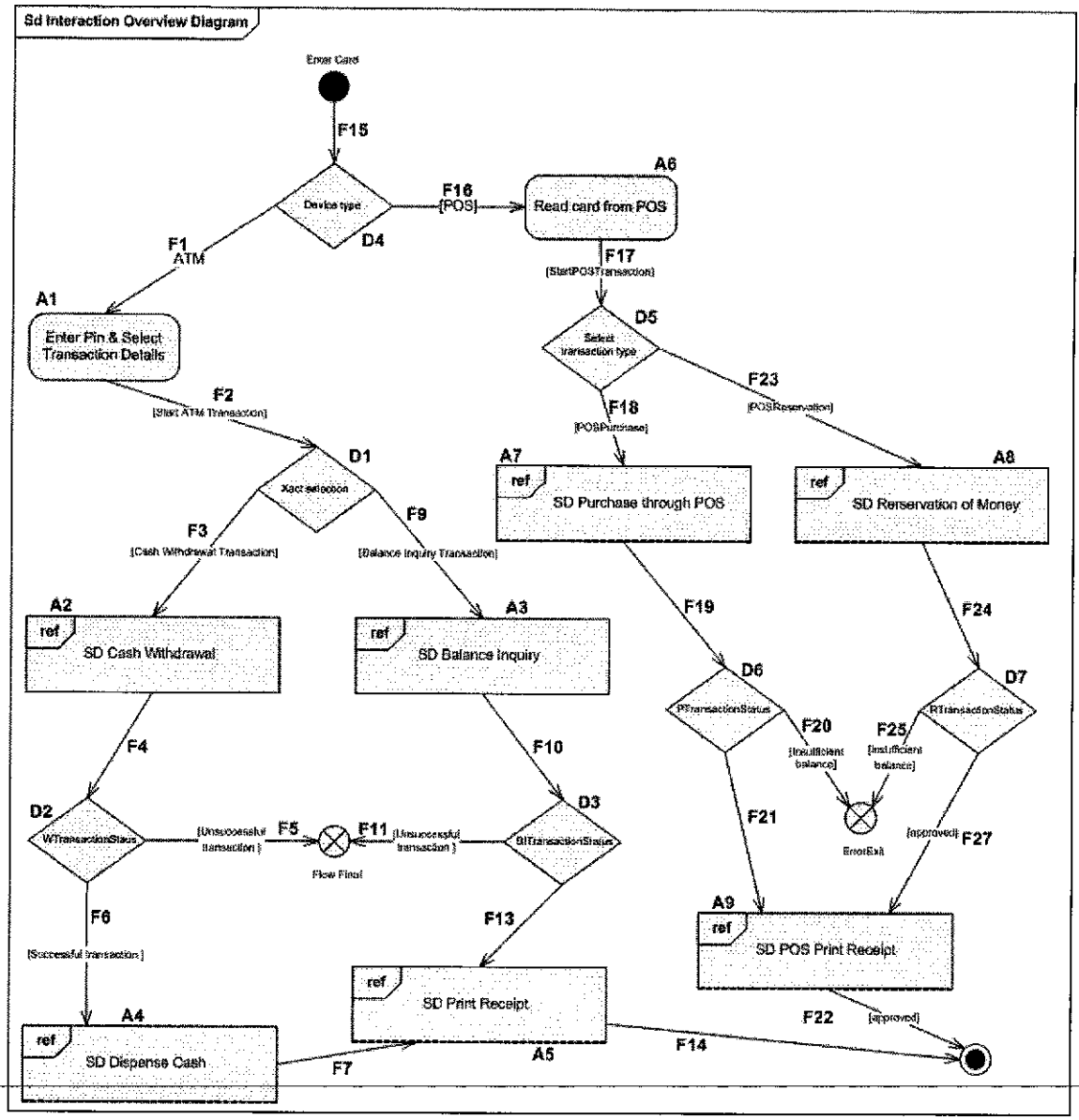

Figure 5.14 ATM (v4) interaction overview diagram

5.3 Learning Center System

The last case study is a Learning Center automation system; it is composed of 2 subsystems: the first is the course registration and the second is the exam taking. All user are welcomed to the center everyday to register for courses, the applications should provide the employee at the center with the required information. User can also apply and take the certification exams at this center; passing students will be given a certificate after they pass their exam.

The original system is made of 14 classes, 50 attributes, 62 methods and 13 relationships (2 of which are inheritance). The class diagram before any modification done to the system is shown in Figure 5.15:

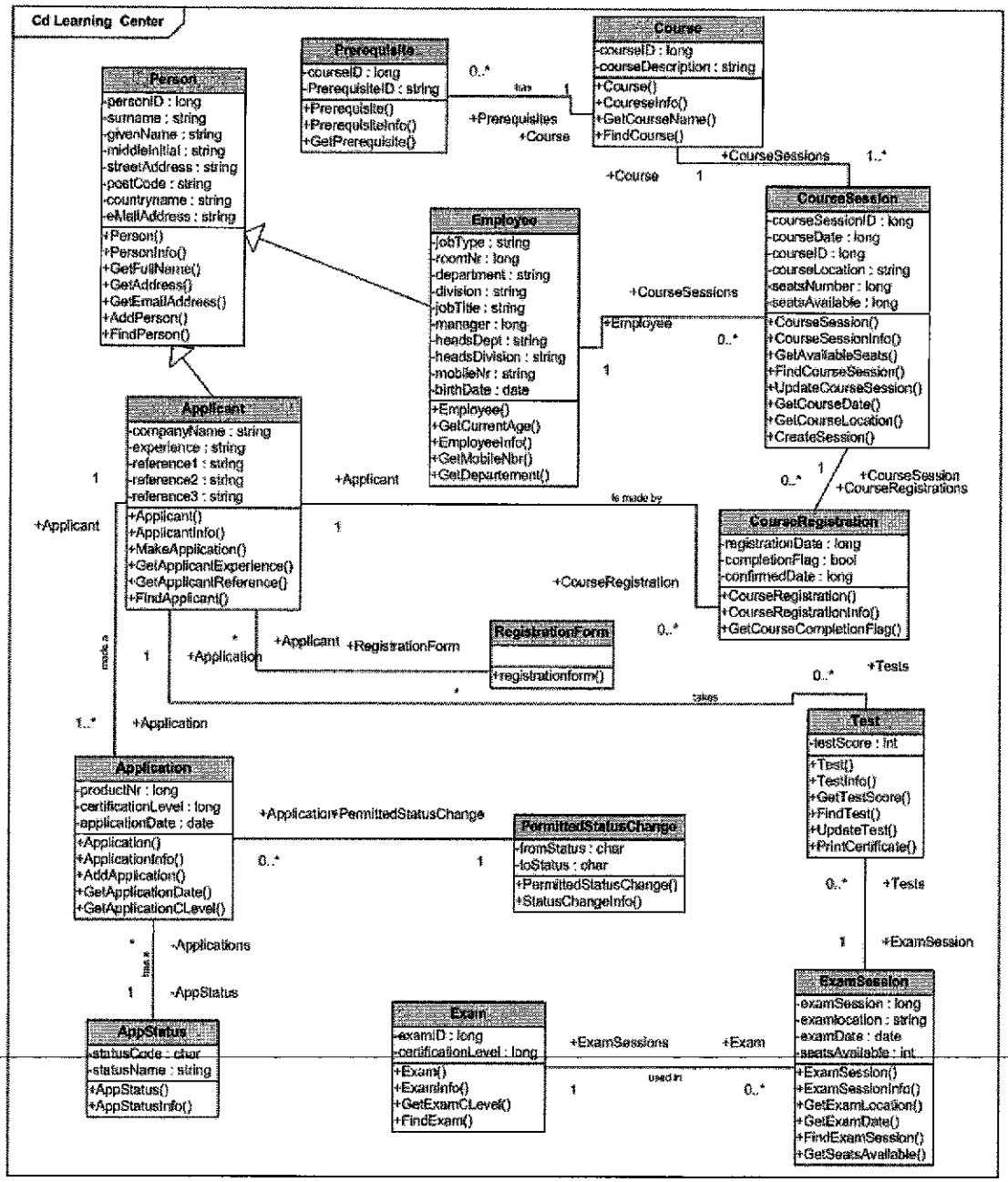

Figure 5.15 Learning Center (v1) class diagram

The interaction overview diagram is composed of 11 Referencing sequence diagrams and 6 decisions nodes. The illustration of the interaction overview diagram of this system is depicted in Figure 5.16.

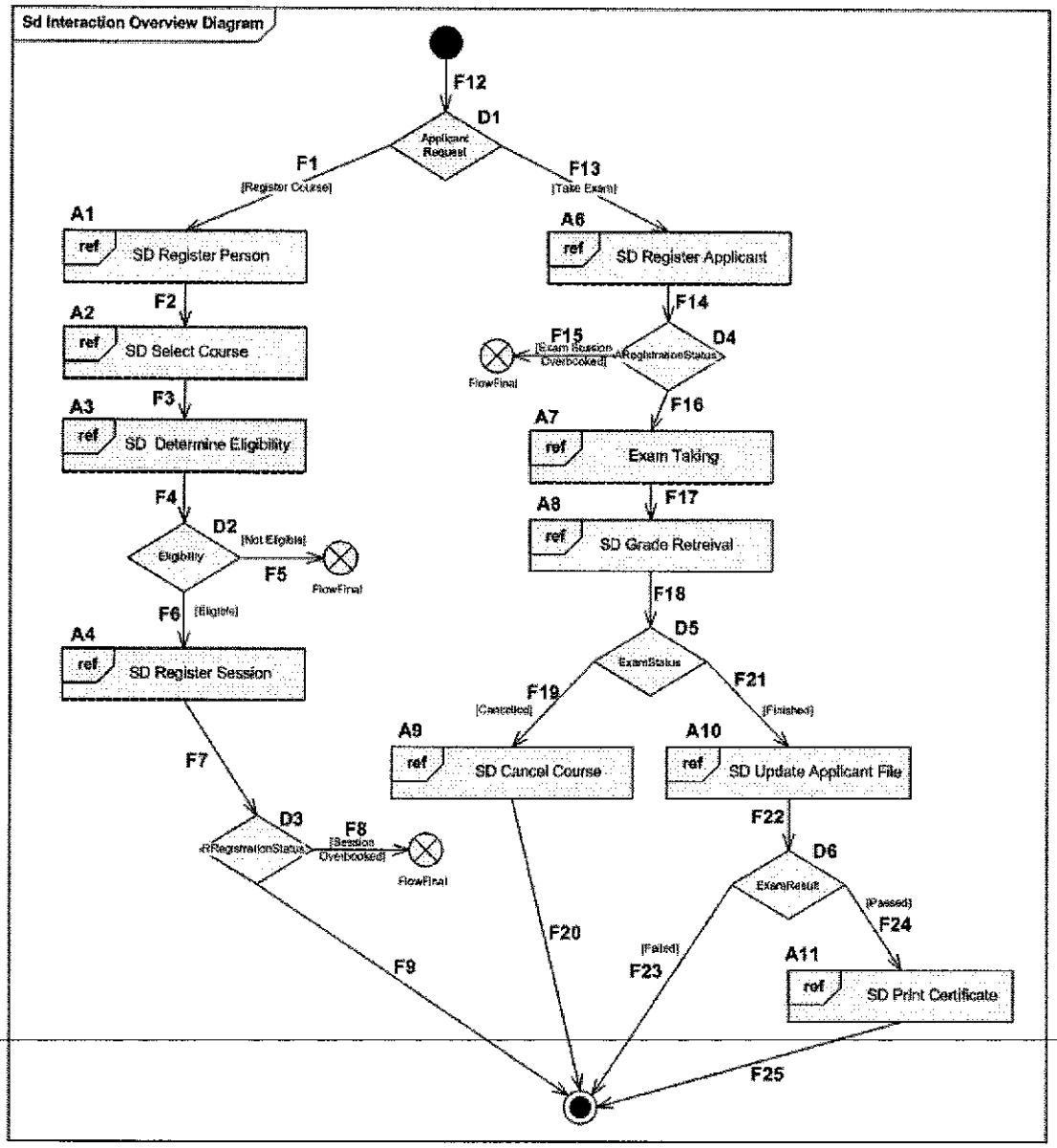

Figure 5.16 Learning Center (v1) interaction overview diagram

5.3.1 Learning Center v2

The first feature added to the system was the ability to send and print the user registration system. Another enhancement was conceived, which was the addition of a passing grade for each exam.

The translation of these changes on the class level imposed the addition of three attributes: the first is in the CouseSession class: infoSent is used as a flag to be raised when the information is sent to the applicant. In the Title class, testResult was added to save the result of each test made by the applicant, and finally the passing grade for each exam caused a modification in the Exam class by adding the passingGrade attribute. Three methods were also added, PrintInfo() and SendInfo() to the CourseSession class and GetPassingGrade() to the Exam class which returns the passing grade of a specific exam. Two methods were also affected by this change, GetTestScore() and FindExam(). Figure 5.17 denotes the class diagram (v2) after the update.

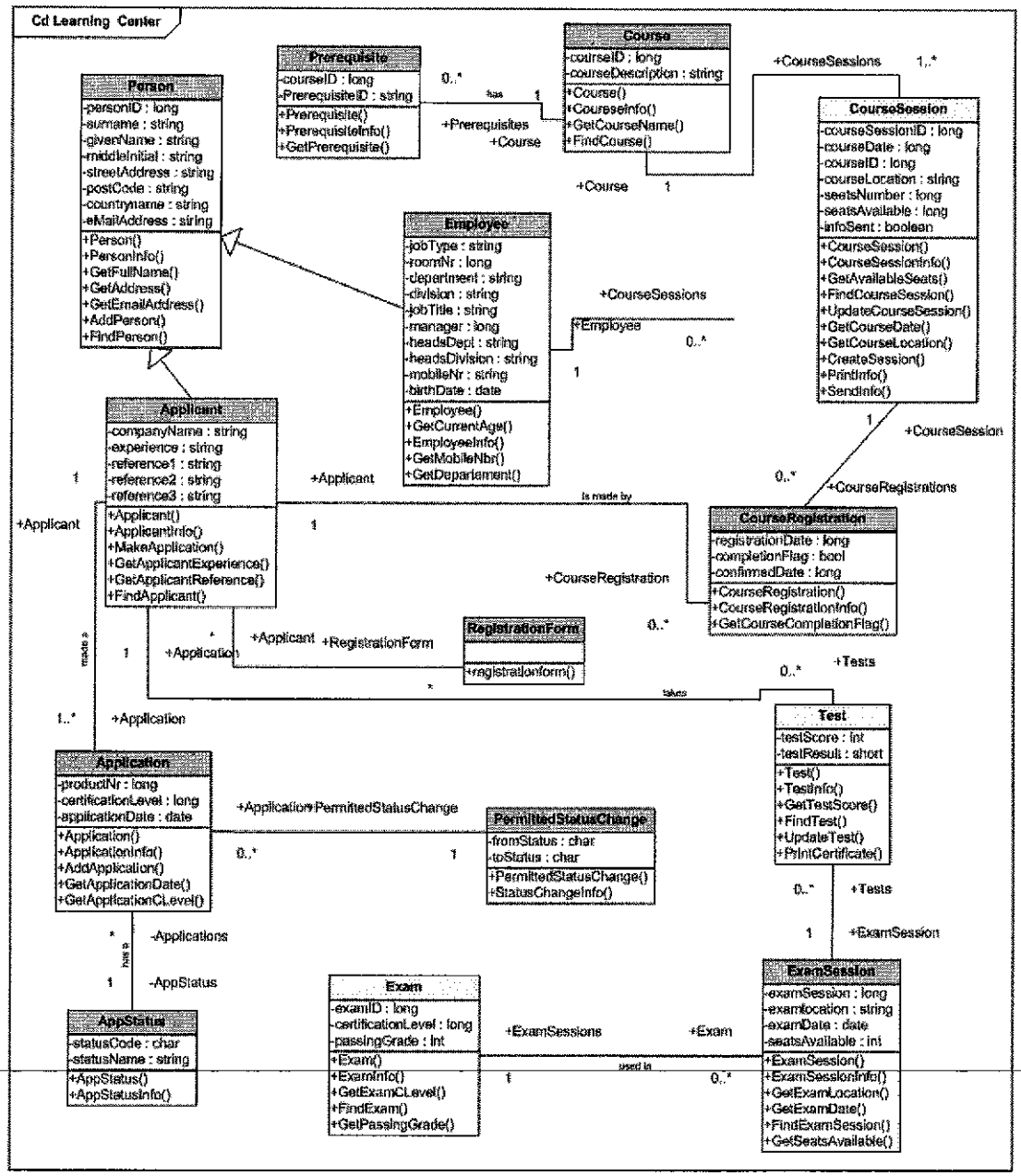

Figure 5.17 Learning Center (v2) class diagram

Table 5.10 is a summary of the changes made on the class level:

Table 5.10 Learning Center classes change results (v1-v2)

	Total *before update*	**Added**	**Modified**	**Deleted**	**Total** *after update*
Attributes	50	3	0	0	53
Methods	62	3	2	0	65
Relationships	13	0	0	0	13
Classes	14	0	3	0	14

The changes accompanying this diagram resulted in the addition of one sequence Diagram (SD Print & Send Registration Info). This change led to the deletion of one flow: F9 and the addition of F10 and F11. The figure below (5.18) depicts the interaction overview diagram (v2) of the Learning Center Application after update.

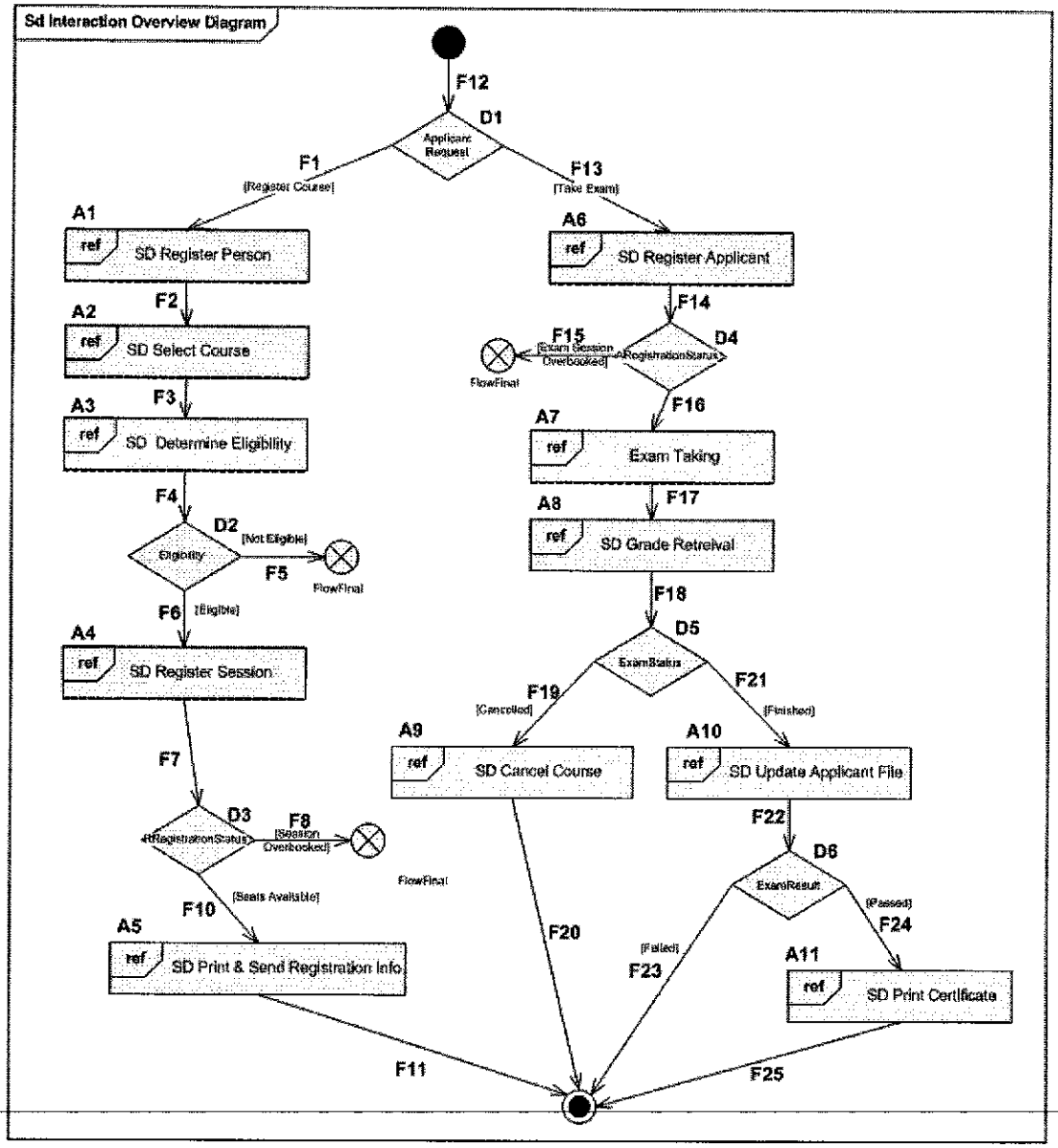

Figure 5.18 Learning Center (v2) interaction overview diagram

5.3.2 Learning Center v3

The manager of the learning center has decided to restraint the option of allowing the applicant to create a session whenever he wants. The manager now selects the days that the applicants can chose from to create a new session. These values are inserted in a table where applicant can chose from. Every course creation attempt that is not included in the

course schedule list will fail. This new step will allow the manager to fix on specific dates for allowed sessions.

The logic transformation made to the system resulted in the following changes on the class diagram level: CourseSchedule class was added to the system, it is composed of two attributes: CourseID and CourseDate, these attributes will be employed to store the information of the allowed sessions. The methods of this new class are in the number of four: CourseSchedule(), CourseScheduleInfo(), GetCourseSchedule() and SetCourseSchedule(). A change encountered the CourseSession class which resulted in a modification inside the GetAvailableSeats(). Figure 5.19 depicts the Learning Center (v3) class diagram.

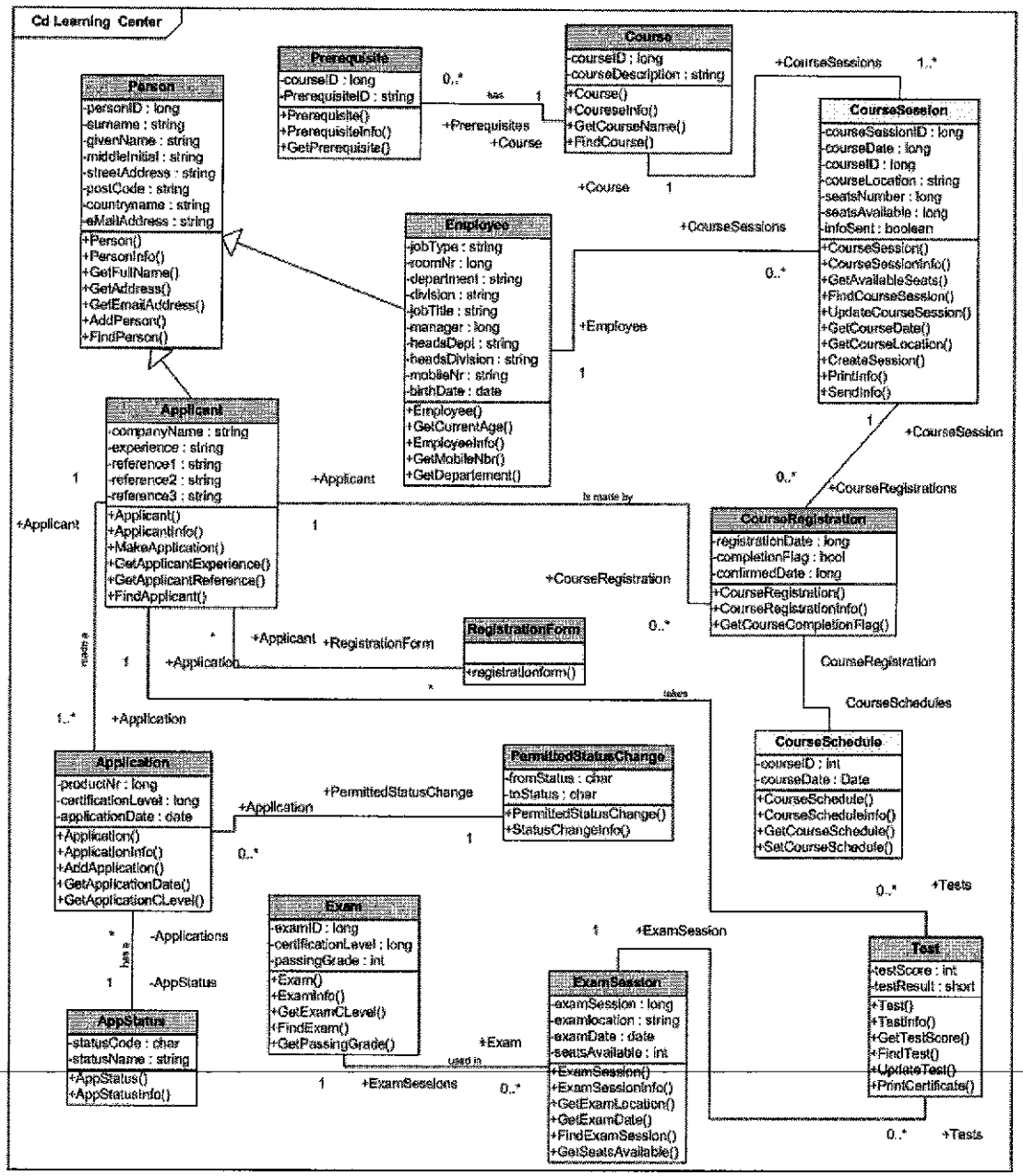

Figure 5.19 Learning Center (v3) class diagram

The changes that affected the class diagrams are summarized in Table 5.11

Table 5.11 Learning Center classes change results (v2-v3)

	Total *before update*	Added	Modified	Deleted	Total *after update*
Attributes	53	2	0	0	55
Methods	65	4	1	0	69
Relationships	13	1	0	0	14
Classes	14	1	1	0	15

This change did not cause the interaction overview diagram to change. The interaction overview diagram of the Learning Center Application before the update (which remained the same) is shown in Figure 5.18.

5.3.3 Learning Center v4

The last modification made to the system was the ability for the applicants to create a new course session. The applicants can now create sessions at will as long as it falls in the dates of allowed sessions. The candidates no longer have to choose from existing sessions or wait for the sessions to be booked; they have more flexibility to choose their courses based on their convenient time table.

As a result of this change, only one method was changed. The GetTestScore() inside the Test class was modified. The final version of the class diagram (v4) is shown in the figure below (Figure 5.20):

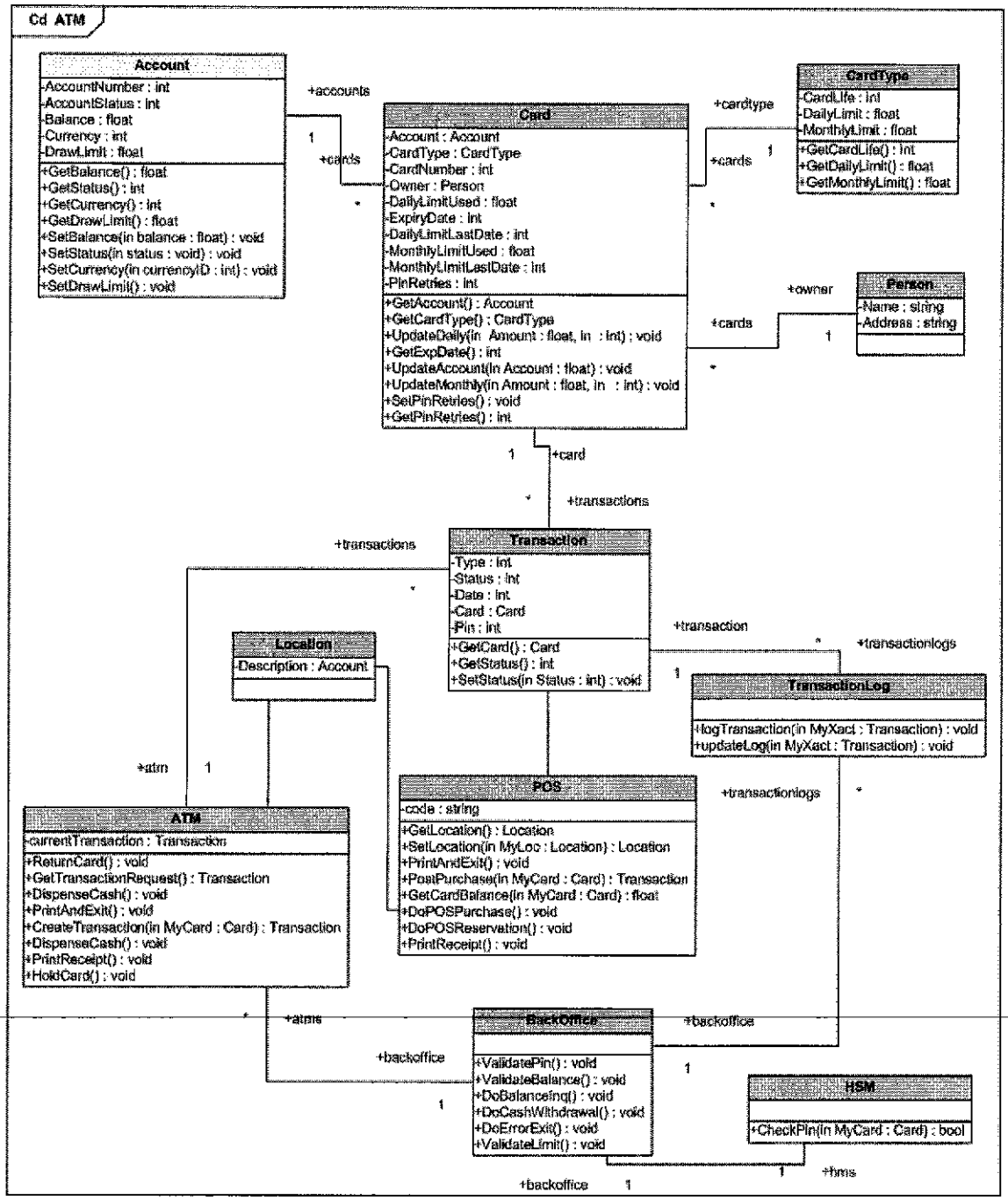

Figure 5.20 Learning Center (v4) class diagram

The resulting change summary on the class level is shown in Table 5.9:

Table 5.12 Learning Center classes change results (v3-v4)

	Total *before update*	**Added**	**Modified**	**Deleted**	**Total** *after update*
Attributes	55	0	0	0	55
Methods	69	0	1	0	69
Relationships	14	0	0	0	14
Classes	15	0	1	0	15

This change did not cause the interaction overview diagram to change. The diagram can be seen in Figure 5.18.

Chapter 6

Empirical Results and Discussion

6.1 Methodology

The empirical study procedure can be divided into seven steps:

- Design the application with three UML diagrams: Class Diagram, Interaction Overview Diagram and Sequence Diagrams.
- Create a set of initial unit and integration test cases
- The application is then subject to a logical change
- Translate the change in the appropriate UML diagrams
- Run the regression testing technique
- Compute #R, Inclusiveness and precision
- Assess and compare the results

Figure 6.1 depicts the methodology using a UML activity diagrams, starting at the design phase of the UML model and ending with the assessment of the method and comparing it with several other regression test selection techniques.

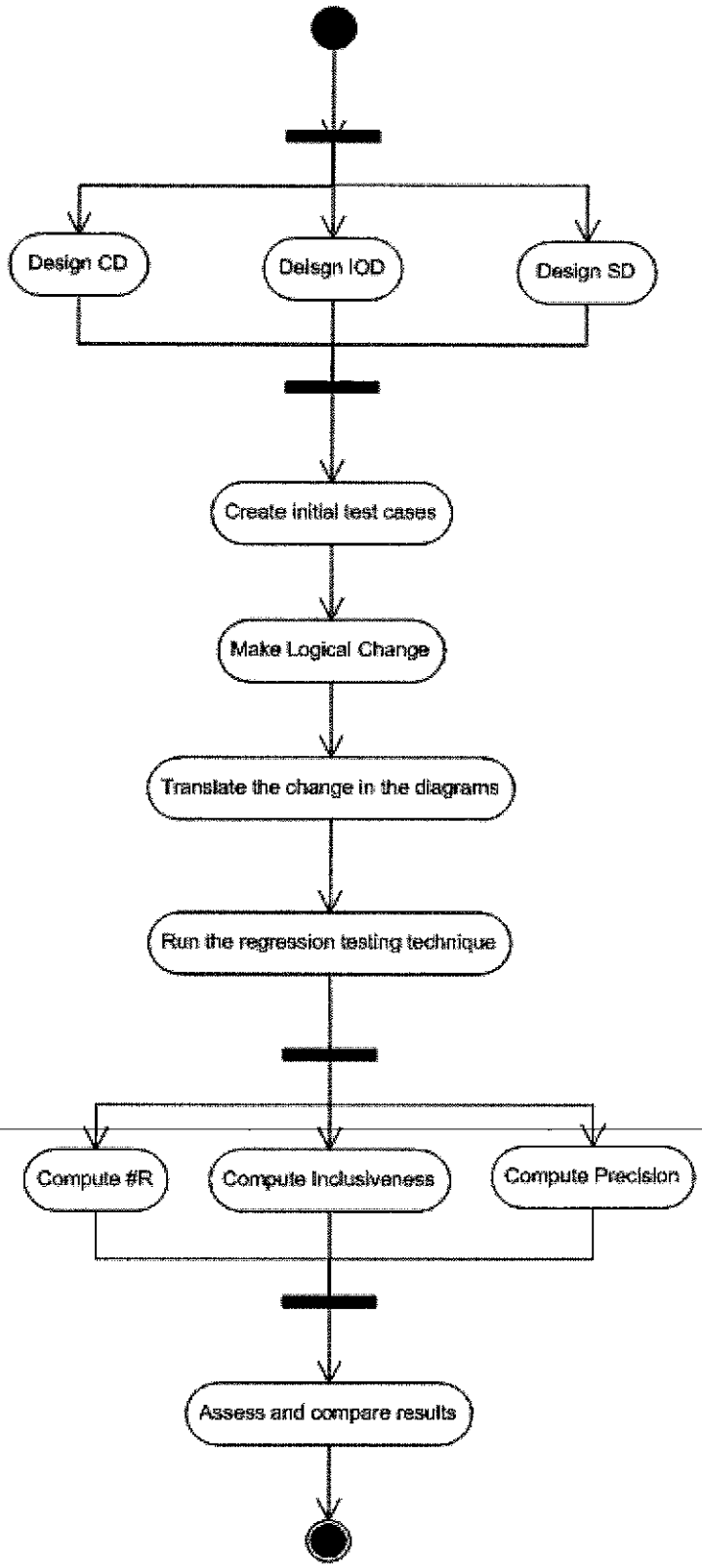

Figure 6. 1 Methodology in Activity Diagram

6.2 Results

In this section, we present a summary of the results from the nine examples (P1V1-P3V3) provided in chapter 5. For each example, the class diagram, interaction overview diagram and the initial suite of test cases were manually generated. Instrumentation was used on the UML design level: a Tool was used to create the UML diagrams. The nine applications are summarized in Table 6.1. For each example, three criteria were provided to give a notion on the case study complexity: M denotes the number of methods; C stands for the number of classes and NIO for the number of nodes inside the UML interaction overview diagram. Typically, each logical change defines a new version of the case study; the modification results in several adjustments on the design level.

One way to quantitatively evaluate this method is providing the $\#R_1$ and $\#R_2$ figures (discussed in section 4.1). Table 6.1 presents the numbers and percentages of the selected test cases where N denotes the total number of original test cases for each application. S_1 and S_2 are the number of test cases selected by the regression testing technique; S_1 refers to the count of the test cases before the refinement step while S_2 points to number after this step

Table 6.1 Description and complexity of the case studies

App ID	App Description	M	C	NIO
P1V1	Library System v1	74	8	13
P1V2	Library System v2	79	8	15
P1V3	Library System v3	81	8	16
P2V1	ATM Application v1	43	11	15
P2V2	ATM Application v2	43	11	16
P2V3	ATM Application v3	46	11	16
P3V1	Learning Center v1	62	14	16
P3V2	Learning Center v2	65	15	17
P3V3	Learning Center v3	69	15	17

As a result of Table 6.2, Table 6.3 summarizes the previous table, the least and largest numbers of selected test cases are presented for the method before and after refinement.

Table 6.2 Number of selected test cases

App ID	N	S_1 (#R_1)	S_2(#R_2)
P1V1	83	7 (**8.4%**)	7 (**8.4%**)
P1V2	90	10 (**11.1%**)	7 (**7.8%**)
P1V3	95	3 (**3.2%**)	3 (**3.2%**)
P2V1	55	9 (**16.4%**)	8 (**14.5%**)
P2V2	55	4 (**7.3%**)	1 (**1.8%**)
P2V3	59	3 (**5.1%**)	2 (**3.4%**)
P3V1	74	15 (**20.3%**)	13 (**17.6%**)
P3V2	77	8 (**10.4%**)	8 (**10.4%**)
P3V3	82	9 (**11%**)	9 (**11%**)

Table 6.3 Aggregate results for the selected test cases

Criteria	#R_1 (%)	#R_2 (%)
Least #R_x	3.2	1.8
Largest #R_x	20.3	17.6

Table 6.4 and Table 6.5 illustrates the precision and inclusiveness figures for the nine examples, they are computed using the Equation 4.1 and 4.2. *mr* stands for modification revealing, a test case is called modification revealing if it causes the output of the two applications (before and after update) to be different. *nmr* stands for non modification revealing, if the test case causes the application to have the same output. *onmr* stands for omitted non modification revealing representing the number of non modification revealing test cases omitted by the regression testing technique. *smr* represents the selected modification revealing by the technique, in other words it's the number of test cases, selected by the technique, that are supposed to cause the applications (before and after update) to differ in terms of output. The precision and inclusiveness results of the

applications before refinement are shown in Table 6.4; the results after refinement are depicted in Table 6.5.

Table 6.4 Precision and Inclusiveness results before refinement

App ID	N	mr	nmr	onmr	smr	Precision (%)	Inclusiveness (%)
P1V1	83	3	80	76	3	95	100
P1V2	90	7	83	81	7	97.6	100
P1V3	95	11	84	84	3	100	27.3
P2V1	55	6	49	46	6	93.9	100
P2V2	55	1	54	51	1	94.4	100
P2V3	59	3	56	55	2	98.2	66.7
P3V1	74	10	64	59	10	92.2	100
P3V2	77	2	75	69	2	92	100
P3V3	82	9	73	71	7	98.6	77.8

Table 6.5 Precision and Inclusiveness results after refinement

App ID	N	mr	nmr	onmr	smr	Precision (%)	Inclusiveness (%)	Inclusiveness after imp. (%)
P1V1	83	3	80	76	3	95	100	100
P1V2	90	7	83	83	7	100	100	100
P1V3	95	11	84	84	3	100	27.3	100
P2V1	55	6	49	47	6	95.9	100	100
P2V2	55	1	54	54	1	100	100	100
P2V3	59	3	56	56	2	100	66.7	100
P3V1	74	10	64	61	10	95.3	100	100
P3V2	77	2	75	69	2	92	100	100
P3V3	82	9	73	71	7	98.6	77.8	100

Table 6.6 describes the set of test case selection techniques used. Each method, used for comparison purposes, is given a different reference so it can be identified later on in Table 6.8. This last table summarizes the results of the comparison between the method in hand and methods that generates random test cases. For each of these methods, N, the initial number of test cases, is provided in addition to the previously stated criteria: #R

(number of selected test cases), inclusiveness and precision (Further details can be found in Table A.2).

Table 6.6 List of the test cases selection techniques

Reference	Description
1	Regression testing technique results
2	Regression testing technique results after improvement
3	Randomly selected test cases by type (same #R)
4	Randomly selected test cases (same #R)
5	Select-All test cases
6	Randomly selected test cases (#R = 20%)
7	Randomly selected test cases (#R = 50%)

Table 6.7 summarizes the test cases selected results from the original redundant test cases. More specifically, Reference 1 represents the results from the regression test selection technique and Reference 2 denotes the results from the redundant test case selection technique. Full specifications of these numbers are found in Table A.2.

The results extracted from the regression test selection technique (Table 6.8) are denoted as the first reference, the second reflects the results of our improvement discussed earlier. The rest of the table represents the results of random selected test cases. A random number generator (Daniels, 2003) was used to specify the selected test cases, this is done by providing x numbers as input in addition to the desired interval. In our case, x represents the number S of selected test cases and the interval ranges from 1 to N. The output will be a set of numbers that will be used as the test cases identifier. Reference 3 denotes a set of randomly selected test cases having the same #R as the regression test selection technique. In this case, the number of test cases selected is based on the tests type; for example, if the original test case contained a unit test cases and b

integration test cases, the same numbers is used for the randomly selected test cases. On the other hand reference 4 is different than the previous results in the sense that it does not take the test type into consideration, the number of test cases is equal to the original set as a whole. Reference 5 is a Select-All test case selection technique; it retests all the original test cases. In Reference 6 and 7, the number of selected test cases is fixed, 20% for the first and 50% for the second, the test cases are still selected randomly.

Table 6.7 Comparison with redundant test cases

Reference	App ID	N	#R	Pre.	Inc.
1	P1V1	83	8	95	100
8	P1V1	93	12	95	100

Table 6.8 #R, precision & inclusiveness summary for the test case selection techniques

AppID	N	Reference 1			Reference 2			Reference 3			Reference 4			Reference 5			Reference 6			Reference 7		
		#R	Pre	Inc	#R	Pre	Inc	#R	Pre	Inc	#R	Pre	Inc	#R	Pre	Inc	#R	Pre	Inc	#R	Pre	Inc
P1V1	83	8	95	100	8	95	100	8	93	33	8	91	0	100	0	100	20	80	33	51	49	33
P1V2	90	8	100	100	8	100	100	8	96	57	8	92	0	100	0	100	20	78	14	50	52	29
P1V3	95	3	100	27	3	100	100	3	98	9	3	96	0	100	0	100	20	80	18	51	50	55
P2V1	55	15	96	100	15	96	100	15	88	33	15	89	17	100	0	100	20	78	0	51	49	67
P2V2	55	2	100	100	2	100	100	2	98	0	2	98	0	100	0	100	20	80	0	51	48	0
P2V3	59	3	100	67	3	100	100	3	96	0	3	96	0	100	0	100	20	79	0	51	48	33
P3V1	74	18	95	100	18	95	100	18	86	40	18	83	20	100	0	100	20	80	20	50	50	50
P3V2	77	10	92	100	10	92	100	10	89	0	10	89	0	100	0	100	21	79	0	51	51	0
P3V3	82	11	99	78	11	99	100	11	90	22	11	90	22	100	0	100	21	78	11	50	52	33

6.3 Discussion of Results

The number of test cases (N) is computed by adding the number of unit test cases to the number of integration test cases. For this technique, the number of unit test cases represents the number of methods inside the application, each method should be tested separately to measure its accuracy and make sure it is fault free. These case studies are considered small to medium size applications, Table 6.1 provide a rough complexity comparison between them. The larger the number of nodes and classes is, the more complex the system.

The integration test cases are extracted from the interaction overview diagram and the sequence diagrams; they represent all possible paths inside the application; the integration path goes through different nodes till its ending point (flow final or activity final). Two equivalent paths running the IO are considered different if they pass through different lanes inside the sequence diagrams. Table 6.2 depicts the values of N for all nine examples; the original set of test cases represents the minimum number of tests yet having full coverage meaning that each application path will be tested at least one time. This choice of test cases insures a lack of redundancy in the tests; each method and test path is executed once.

The number of selected test cases is considered small (Table 6.2 and Table 6.3), the numbers ranged from 1.8% to 17.6%. This small number of selected test cases is expected since the technique used selects only the necessary test cases executing the modified segments. As shown in Table 6.2, the results after the refinement step were lower 56% of the times with an average of 2.2% decrease in $\#R$ before and after refinement. The largest decline occurred in P2V2 where the number of selected test cases

dropped from 7.3% to 1.8%. For the rest of the examples, the refinement step did not affect the results; the number of selected test cases remained the same. The importance of this step can be derived from the numbers mentioned above; the reduction in terms of *#R* is considerable.

An additional number of initial test cases were used to one example to study the effect of using redundant test cases on the regression test case selection technique. Results have shown (Table 6.7) that the precision and inclusiveness rates were not affected by the added test cases, it only affected the number of selected test cases: #R has increased from 8% to 12%. Since more test cases are added to the original set, more test cases will be selected by the technique. This means that adding redundant test cases will only lead to a higher #R ratio thus adding more work to execute these test cases, leading to a decrease in the technique's efficiency. In other examples, the precision may change by adding more test cases, but this would be relatively small since we are increasing both the number test cases and the number of original set of test cases. The inclusiveness will not change because we are already choosing test cases that cover all possible paths.

Tables 6.4 and 6.5 depict the precision and inclusiveness outcome before and after the refinement step. These tables illustrate in detail how the values were computed, the formulas are found in Equation 4.1 and 4.2. In five of the examples, this step increased the precision with an average of 3%. In the other cases, no change was spotted on the precision rate. The inclusiveness rate did not change in any of the example since this reduction step only deals with paths that are selected for refinement, meaning that this

rechecking step will only eliminate path that are valid in the application; thus not affecting the inclusiveness.

The final results of the technique are shown in Table 6.5, the precision rates are considered high. Most of the time, this method omits tests that are non modification revealing. The lowest precision rate was in P3V2 with a value of 92%, nevertheless this value is still considered high. The regression testing technique used in these results can be considered safe due to its high precision ratios. The technique will select the least necessary testing for most of the cases.

The inclusiveness results are depicted in Table 6.5. 67% of the time, this technique was able to achieve 100% inclusiveness. In the remaining times (in 3 examples), it fails to pull off with high results; this is due to the fact that this technique fails to unveil changes made inside the combined fragment.

After the improvement made to the technique (section 3.3), the results have shown remarkable outcome, the inclusiveness rate has arisen to full coverage for all test cases. The technique now selects all the affected test cases without having an impact on the precision results. This means that the improvement suggested did not entail any negative effect on the technique.

To further assess the value of this technique; a comparison was made with randomly selected test cases. In contrast to the studied technique, both random test case selection techniques with the same number of selected test cases (Table 6.8, reference 3 and 4) have showed inferior results. Even thought the randomly test cases by test type showed better results than the one that takes the whole test cases as one unit (since the probability, of finding modification-revealing test cases and omitting non-modification-

revealing, is higher when dealing with each type alone than the system as a whole), both have lower *smr* and *onmr*. For all nine case studies versions, the regression test selection technique has showed better results in terms of precision and inclusiveness. In some cases (for ex. P2V2), the inclusiveness rate had a total failure outcome going from 100% to 0%, the reason is that random selection of test cases has an equal chance of selecting any test case, thus having the possibility of failing to pick the ones with changes. In this case, the number of test cases selected for retest is bound to the one used in our technique; since our technique has a low value of #R, the number selected test cases for retest is low which can justify the low inclusiveness and the relatively good precision rates. The less the method select test cases, the higher probability it has to be precise since precision measures to which a technique can omit non-modification-revealing tests.

Select-all test have full coverage of the test cases, they re-test all the original test cases, and they insure very high rates of inclusiveness. As pleasant as it looks, re-testing everything will cause the method of having the lowest precision rates. As Table 6.8 (Reference 5) shows, the inclusiveness rates are at their highest levels thus having better results than our technique used. This has resulted in the lowest precision possible which may be relevant for small applications but is surely not feasible for bigger applications. Some application can have more than 300.000+ tests (Briand, 2003) and re-executing this amount of test cases every time a change occurs is surely impossible. Even for small applications, doing this additional work on every change is unnecessary and impractical most of the times.

One other way to select random test cases is to fix the number selected test cases (#R) and then generate these tests randomly. Two rates were used #R=20% and #R=50%

(Table 6.8 reference 6 and 7). The lower the #R, the more precise the technique is, the less chance it has to detect modification-revealing tests (lower inclusiveness). In the first case, with #R=20%, the precision was around 80% but have very low inclusiveness rates. When a higher number of test cases were selected (reference 7), the precision rate dropped and the inclusiveness has risen. The precision rates in these two cases are much lower than our regression testing technique, even the inclusiveness rates are much worse in both cases (except for the P1V2 in the case of #R=50% where the random test cases had a 2% higher rate).

6.4 Limitation and suggestions

This technique requires the system to have only one interaction overview diagram for the whole application; this limitation may be doable for small or even medium sized applications but it surely fails for larger ones. Large system cannot be modeled in one interaction overview diagram, several may be needed.

An iterative approach for this type of testing should be applied, interaction overview diagrams with the same signature should be checked for changes then all interactions inside each IO should also be checked for changes.

6.5 Technique Complexity

In 2006, Takkoush discussed the time complexity of each of the algorithm, when combined, forms the complexity of his technique for regression testing. Table 6.6 summarizes the complexity of each algorithm.

Table 6.9 Complexity summaries

Algorithm	Complexity	Descrption
GenerateChangedMethods	O(*M*)	M: # of methods
CDIntegrationTestSelection	O(*t.sd* * *l* **m*)	*t.sd*: # of rows in T.SD *l:* is the average length of paths pass through in T.SD *m*: # changed methods
CDUnitTestSelection	O(*sd* * (*e* * *n* + *m*)) + O(*m'* * *ut*)).	**O(*sd* * (*e* * *n* + *m*))**: indirectly changed methods for each SD. *e*: average edge count in the SD *n*: average # of nodes in the SD *m*: # changed methods **O(*m'* * *ut*)**: Selecting the unit test cases from UT that test the changed methods
IOTestCaseSelection	O(*sd* * *a*)	*sd*: the count of the SD in IO *a*: average # of SD in the SDs
IOBasedClassification	O(*e* * *t* * *l*)	*e*: #of edges in *IO* *t*: # of integration test cases in T.IO *l:* average length of path per test case in T.IO
SDBasedClassification	O(*t''* * *sd* * *l*)	*t'':* # of candidate test cases for retest *sd:* total # of changed SD *l:* average method-call path traversed per SD from T.SD.

Chapter 7

Conclusion and Future Works

This paper has presented an empirical study based on a regression testing technique for OO software. This technique uses the UML design to model the application; no need to access the code, the method works on the design level. The motivation behind this work is to study the accuracy and the feasibility of the method. The method was studied in term of precision, inclusiveness and number of selected test cases.

Several case studies were fully documented and presented, test cases were drawn and then the technique was applied to each of the examples. The case studies varied in specification and size. Results have shown that a single change in the logic may have complex impact on the selection of test cases, thus the need of automation in order to minimize all human errors. The results drawn from these examples seemed promising; the drawback was its inability to spot changes inside the fragments.

In addition to the empirical work presented in this paper, an amendment was made to the technique; it now allows the detection of change made inside the combined fragment. This improvement impacts the inclusiveness results, since all test case that failed to cover the change were due to modification of operand inside the fragment.

Due to lack of previous work on testing OO software with this kind of UML diagrams, a comparison was made with randomly selected test cases. The comparison has shown significant difference in results. The technique used has outdone all the methods that select random test cases and demonstrated better accuracy and inclusiveness than all other techniques. The technique selects fewer test cases with a higher precision and

inclusiveness. The average percentage of the selected test cases is around 8.6% which is considered a fairly low percentage. Precision's average is as high as 97.5% and an inclusiveness rate of 85.8%. The inclusiveness rate changed dramatically after the amendment, it hit the full coverage rate of 100% in all nine examples.

Working with these examples was very time consuming and may become unrealistic with application having a larger size. Future work should include automating both the technique and the extraction of test cases. The tool can also check for consistency between diagrams in order to reduce UML design faults related to manual work.

One other aspect can be investigated: compare UML design level regression testing with that done at the source code level.

References

Agrawal, H., Horgan, J.R., Krauser, E.W. and London, S. (1993). Incremental Regression Testing. *ICSM '93: Proceedings of the Conference on Software Maintenance* (p.p 348-357). Washington DC : USA.

Benedusi, P., Cimitile, A. and De Carlini, U. (1988, October). Post-maintenance testing based on path change analysis. *Proceedings of the Conference on Software Maintenance* (pp. 352-361).

Briand, L.C., Labiche, Y., Buist, K. and Socca, G. (2003, August). Automating Impact Analysis and Regression Test Selection Based on UML Designs. *18th IEEE International Conference on Software Maintenance (ICSM'02)* (pp. 252-261).

Daniels, P. (2003). Random Number Generator [Online]. Available: http://www.mdani.demon.co.uk/para/random.htm

Harrold, M. J. and Soffa, M. L. (1988, October). An incremental approach to unit testing during maintenance. *Proceedings of the Conference on Software Maintenance* (pp. 362-367).

Harrold, M. J. and Soffa, M. L. (1989, May). An incremental data flow testing tool. *Proceedings of the Sixth International Conference on Testing Computer Software.* Washington DC.

Harrold, M. J., Jones J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A. and Gujarathi, A (2001, October). Regression test selection for Java software. Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications. (pp. 312-326).

Hsia, P., Li, X., Kung, DC., Hsu, CT., Li, L., Toyoshima, Y., and Chen, C. (1997). A technique for the selective revalidation of OO software. *Software Maintenance: Research and Practice* (pp. 217-233).

IEEE-Computer Society (Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12, 1990)

Landi, W.A. and Ryder, B.G. (1992, June). A safe approximate algorithm for interprocedural pointer aliasing. *Proceedings of SIGPLAN '92 Conference on Programming Language Design and Implementation.* (pp. 235-248).

Leung , H. K. N. and White, L. J. (1990, November). A study of integration testing and software regression at the integration level. *Proceedings of the Conference on Software Maintenance.* (pp. 290-300).

Leung , H. K. N. and White, L. J. (1991, October 15-17). A Cost Model to Compare Regression Test Strategies. *Proc. Conference on Software Maintenance* (pp. 201-208). Sorrento: Italy.

Leung , H. K. N. and White, L. J. (1992). A firewall concept for both control-flow and data-flow in regression integration testing. *International Conference on Software Maintenance*. (pp. 262-271).

Mansour, N. and Bahsoon, R. (2002) Reduction-based methods and metrics for selective regression testing. *Information and Software Technology*, 44 (7). 431-443.

Rothermel, G., Harrold, M.J. (1996, August). Analyzing Regression Test Selection Techniques. IEEE Transactions on Software Engineering, 22(8):529-551.

Rothermel, G., Harrold, M.J. and Dedhia, J. (2000, June): Regression Test Selection for C++ Software. *Journal of Software Testing, Verification, and Reliability*, 10(2), 77-109.

Takkoush , H. (2006). UML Based Regression Testing Technique for OO Software. Unpublished thesis, Lebanese American University, Lebanon.

Wu, Y., Chen, M. and Kao, H (1999) Regression Testing on object-Oriented Programs *Tenth International Symposium on Software Reliability.*

Wu, Y., Chen, H. and Offutt, J. (2003, November). UML-based integration testing for component-based software. *Second International Conference on COTS-Based Software Systems (ICCBSS 2003)* (p.p 251-260). Ottawa:Canada

Appendix A

Comparison with Randomly Selected Test Cases

Table A.1 List of the test cases selection techniques

Reference	Description
1	Regression testing technique results
2	Regression testing technique results after improvement
3	Randomly selected test cases by type (same #R)
4	Randomly selected test cases (same #R)
5	Select-All test cases
6	Randomly selected test cases (#R = 20%)
7	Randomly selected test cases (#R = 50%)
8	Regression testing technique results with redundant test cases

Table A.2 #R, precision and inclusiveness results for the test case selection techniques

Reference	App ID	N	#R	mr	nmr	onmr	smr	Pre.	Inc.
1	P1V1	83	8	3	80	76	3	95	100
	P1V2	90	8	7	83	83	7	100	100
	P1V3	95	3	11	84	84	3	100	27
	P2V1	55	15	6	49	47	6	96	100
	P2V2	55	2	1	54	54	1	100	100
	P2V3	59	3	3	56	56	2	100	67
	P3V1	74	18	10	64	61	10	95	100
	P3V2	77	10	2	75	69	2	92	100
	P3V3	82	11	9	73	71	7	99	78
2	P1V1	83	8	3	80	76	3	95	100
	P1V2	90	8	7	83	83	7	100	100
	P1V3	95	3	11	84	84	3	100	100
	P2V1	55	15	6	49	47	6	96	100
	P2V2	55	2	1	54	54	1	100	100
	P2V3	59	3	3	56	56	2	100	100
	P3V1	74	18	10	64	61	10	95	100
	P3V2	77	10	2	75	69	2	92	100
	P3V3	82	11	9	73	71	7	99	100
3	P1V1	83	8	3	80	74	1	93	33
	P1V2	90	8	7	83	80	4	96	57
	P1V3	95	3	11	84	82	1	98	9
	P2V1	55	15	6	49	43	2	88	33
	P2V2	55	2	1	54	53	0	98	0
	P2V3	59	3	3	56	54	0	96	0
	P3V1	74	18	10	64	55	4	86	40
	P3V2	77	10	2	75	67	0	89	0
	P3V3	82	11	9	73	66	2	90	22
4	P1V1	83	8	3	80	73	0	91	0
	P1V2	90	8	7	83	76	0	92	0
	P1V3	95	3	11	84	81	0	96	0
	P2V1	55	15	6	49	42	1	89	17
	P2V2	55	2	1	54	53	0	98	0
	P2V3	59	3	3	56	54	0	96	0
	P3V1	74	18	10	64	53	2	83	20
	P3V2	77	10	2	75	67	0	89	0
	P3V3	82	11	9	73	66	2	90	22
5	P1V1	83	100	3	80	0	3	0	100
	P1V2	90	100	7	83	0	7	0	100
	P1V3	95	100	11	84	0	3	0	100
	P2V1	55	100	6	49	0	6	0	100
	P2V2	55	100	1	54	0	1	0	100
	P2V3	59	100	3	56	0	2	0	100
	P3V1	74	100	10	64	0	10	0	100
	P3V2	77	100	2	75	0	2	0	100
	P3V3	82	100	9	73	0	7	0	100
6	P1V1	83	20	3	80	64	1	80	33
	P1V2	90	20	7	83	65	1	78	14
	P1V3	95	20	11	84	67	2	80	18
	P2V1	55	20	6	49	38	0	78	0
	P2V2	55	20	1	54	43	0	80	0
	P2V3	59	20	3	56	44	0	79	0
	P3V1	74	20	10	64	51	2	80	20
	P3V2	77	21	2	75	59	0	79	0
	P3V3	82	21	9	73	57	1	78	11
7	P1V1	83	51	3	80	39	1	49	33
	P1V2	90	50	7	83	43	2	52	29
	P1V3	95	51	11	84	42	6	50	55
	P2V1	55	51	6	49	24	4	49	67
	P2V2	55	51	1	54	26	0	48	0
	P2V3	59	51	3	56	27	1	48	33
	P3V1	74	50	10	64	32	5	50	50
	P3V2	77	51	2	75	38	0	51	0
	P3V3	82	50	9	73	38	3	52	33
8	P1V1	93	12	7	86	82	7	95	100