JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2011; **23**:51–68
Published online 20 July 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.508

UML-based regression testing for OO software

Nashat Mansour*, †, Husam Takkoush and Ali Nehme

Department of Computer Science and Mathematics, Lebanese American University, Beirut, Lebanon

SUMMARY

In software maintenance, a system has to be regression tested after modifying it. The goal of regression testing is to ensure that modifications have not adversely affected the system. Regression test selection determines a subset of test cases, from the initial test suite, which concentrates on the parts of the system affected by the modification. Previous techniques have been mainly code-based and several of them have addressed procedural programs. When working with large and complex object-oriented systems, source code-based regression testing is usually costly. This paper proposes a programming-language-independent technique for regression test selection for object-oriented software based on Unified Modeling Language (UML 2.0) design diagrams. These diagrams are: the newly introduced interaction overview diagram, class diagrams, and sequence diagrams. We assume a test suite that contains both unit and system test cases. Based on the software changes reflected in the class and the interaction overview diagrams, our proposed technique selects test cases in phases. In the first phase, we select both unit and system test cases that directly traverse the changed methods and their calling methods. For the second phase, we present algorithms for detecting system level changes in the interaction overview diagram. If the change is at the action level, which is represented by a sequence diagram, only the test cases that execute changed methods will be selected. We apply our proposed technique to a few object-oriented subject applications and evaluate its precision and inclusiveness in addition to the number of selected tests; the results demonstrate the advantages of the technique. Copyright © 2010 John Wiley & Sons, Ltd.

Received 14 November 2008; Revised 23 April 2010; Accepted 17 May 2010

KEY WORDS: design-level testing; object-oriented regression testing; regression test selection; software maintenance; UML

1. INTRODUCTION

A program is regression tested after modifying it in the maintenance phase [1]. Regression testing can also be used in the testing release phase of software development. One important objective of regression testing is to ensure that the change made to a program does not cause adverse side effects. Retesting using all test cases that are generated during initial software development is too costly. Hence, regression test selection techniques aim to select a reduced subset of test cases from the initial test suite for retesting the modified software. This leads to a reduction in the cost of software maintenance.

The selection of appropriate test cases can be made in different ways and a number of regression test selection methods have been proposed. These methods are based on different objectives and techniques such as: firewalls [2]; slicing-based data-flow technique [3]; safe algorithm based on program's control graph [4]; test case reduction algorithms that account for the location of

*Correspondence to: Nashat Mansour, Department of Computer Science and Mathematics, Lebanese American University, Beirut, Lebanon.
†E-mail: nmansour@lau.edu.lb

Copyright © 2010 John Wiley & Sons, Ltd.

the modification made in the program and its effects [5]; optimization problem formulation and nature-inspired heuristics for minimizing the number of selected test cases [6]; concentrating on the glue code and using firewall analysis for component-based software [7]; state machine diagrams for web services [8].

In the last decade, regression testing techniques have been proposed for object-oriented software. Rothermel *et al.* [9] and Harrold *et al.* [10] use syntax-based inter-procedural control flow graphs and class control flow graph for regression testing C++ and Java programs. Taneja and Xie [11] propose an approach called DiffGen for generating regression tests. Their approach detects behavioral differences between two versions of a given Java class by checking observable outputs. Wu *et al.* [12] generate Affected Function dependency graphs based on the functions in the different classes and their behavior and non-behavior effects. Le Traon *et al.* [13] build a test dependency graph which is a directed graph of classes. Then, the graph is decomposed into connected components and the components are ordered. If a component is modified, all the components connected to it will be retested. Chen *et al.* [14] use an activity diagram like a control flow graph to describe the system requirements. The test paths that correspond to the affected graph nodes determine the tests to be rerun. Beydeda and Gruhn [15] build a graph for class level regression testing based on class specification and implementation information. Korel *et al.* [16] use extended finite state machines and carry out dependence analysis of this model based on data and control dependencies of the transitions. This analysis leads to selecting test cases. Wu and Offutt [17] use class, collaboration, and state machine UML diagram for regression testing component-based software. Briand *et al.* [18] use UML to design and classify the test cases into: retestable, reusable, and obsolete. Their approach considers changes by comparing class and sequence diagrams. After that comparison, use cases that have their sequence diagrams changed are determined and test cases related to these use cases are classified. Farooq *et al.* [19] have proposed a selective regression testing strategy that uses class diagrams and state machines for change identification and subsequent test selection. The changes are classified as class-driven and state-driven and the tests are classified as obsolete, reusable, and retestable. Pilskalns *et al.* [20] propose a regression testing technique based on UML sequence and class diagrams. Changes in these diagrams are mapped into vertices of an object method directed acyclic graph. Then, this graph is employed for classifying test cases, selecting the ones to be reused, and determining the new tests to be generated. Also, Batra [21] proposes a UML-based approach for regression testing software components. In this approach, a UML sequence diagram of a component is transformed into an extended control flow graph which is traversed for detecting change information. Then, tests are identified as reusable, obsolete or newly added. The resulting set of regression tests is suitable for detecting interaction and scenario faults.

The majority of the previous techniques are code-based and limited amount of work has been design-based, especially for object-oriented software. When working with large and complex systems, code-based testing and regression testing is quite costly and might not be feasible since it usually requires generating large dependence graphs. In this paper, we present a design-based regression test selection technique for objected-oriented software using standard UML 2.0 diagrams [22] that are not dependent on the particular programming language used. We use, for the first time, the newly introduced interaction overview diagram in UML 2.0 since it provides a modular overview of the system including control flow, action states, and interaction diagrams. We also employ information from class and sequence diagrams for further test selection. We have two test case pools for the unit and system test cases. Based on the software changes reflected in the class and the interaction overview diagrams, the technique for test case selection is mainly composed of two phases. In the first phase, we propose an algorithm for detecting the changed methods in the class diagram; this is done under the assumption that the developers update the design properly. For retesting, we select both unit and system test cases that directly traverse the changed methods and their dependent methods. In the second phase, we propose an algorithm for detecting system level changes in the interaction overview diagram. If the change is at the action level, which is basically a sequence diagram, only the test cases with changed method sequence will be selected. We apply our proposed technique to a few object-oriented subject programs and

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

the results show the advantages of this design-level language-independent regression test selection technique.

This paper is organized as follows. Section 2 presents the proposed technique and algorithms. Section 3 includes the empirical results. Section 4 concludes the paper.

2. REGRESSION TEST SELECTION

In this section, we present our technique for regression test case selection from system and unit test cases based on information derived from class and interaction overview diagrams. Interaction overview diagrams, introduced in UML 2.0, summarize the control flow of the entire system.

2.1. Notation and assumptions

In the remainder of this paper, we use the notation summarized in Table I and the following assumptions:

(i) T is the set of system test cases, and the following tables are maintained using instrumentation.
   (a) T.IO where for each $t_i \in T$, the table T.IO will contain the ordered traversal list of interaction overview artifacts.
   (b) T.SD where for each $t_i \in T$, the table T.SD will contain the ordered traversal list of SD in the form of: OBJECT.METHOD.

(ii) UT is the set of unit test cases where for each $ut_i \in UT$, ut_i method specifies the method tested.

(iii) Instrumentation is assumed to be used when running the initial tests in order to record the correct path coverage. Instrumentation probes would be inserted inside the methods and the control structures that include method calls.

(iv) There is one interaction overview diagram for the entire program.

(v) A sequence diagram is identified by its unique name in the interaction overview diagram, referred to as the signature of the sequence diagram.

(vi) SD messages are method calls. SD messages should be consistent with the called methods, hence the message will carry the same name as the invoked method, the message will be referenced as: Object.MethodName().

Table I. Notation.

Notation	Description
CD	Class diagram
CD'	Modified class diagram
SD	Sequence diagram
T	Set of all system tests and their respective path in the IO and SD diagrams
T.IO	Ordered path list in the interaction overview diagram for each system test case
T.SD	Ordered path list per sequence diagram frame for each system test case
UT	Set of the unit tests and their covered methods
Md	Total number of methods in the class diagram
M	Set of changed methods
SD.M	list of methods involved in SD
IO	Interaction overview diagram
IO'	Modified interaction overview diagram
T'	Set of all tests selected from T for retest
T"	Set of tests (from T) marked as candidates for retest
UT'	Set of all tests from UT selected for retest
t	A single system test case
ut	A single unit test case

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

(vii) Upon updating methods in the source code, their version number in the class diagram should be updated as well.
(viii) A list of methods involved in an SD, SD.M, is maintained before and after updating the IO diagram.

2.2. Overview of the technique

Our proposed technique is based on the UML diagram representation. We consider the CD and CD', IO and IO', the set of system test cases (T) and the set of unit test cases (UT). The technique for test case selection consists of two phases: the first involves detecting changes and selecting tests based on the class diagram and interaction overview diagram; the second involves selecting test cases based on their coverage in the interaction overview diagram. The two phases are summarized in the following:

(I) Test case selection based on changes in class diagram: First, we determine the set of changed methods, M, from class diagram changes. Then, we use M to generate the set of unit and system tests that need to be re-tested.
  (a) Determine the set of changed methods M based on changes in the class diagram affecting one or more UML entities (Association, Class, Method, etc.). We consider each class in both class diagrams, CD and CD', and compare the methods' versions. If there is a change, we add this method to M.
  (b) Select system test cases: for every method in M, select from T.SD the test cases that have a reference call to the changed methods.
  (c) Select unit test cases:
    (i) Select all the unit test cases that test the methods in M.
    (ii) Select all the unit tests for the methods that M's methods are reachable from. In an SD, the flow of methods specifies which methods are called before others. However, a change in method m, would imply that the methods calling m need to be retested as well. Therefore, for every SD, we consider all methods whose calls precede those in M in order to rerun their unit tests.

(II) Test case selection is based on changes in the interaction overview diagram. The comparison algorithm works by traversing IO and IO' in parallel:
  (a) Compare every SD in IO and IO' and mark all the changed SDs as changed. This is performed by doing a lifeline by lifeline and messages comparison between the SD in IO and that in IO'.
  (b) Select tests based on IO changes: Compare IO and IO', if there is a changed flow/edge, decision, decision final, or SD signature, then all test cases traversing the changed element are marked for retest.
  (c) If an SD signature is not changed, but the SD itself is marked as changed, then we mark the test cases that traverse the changed SDs as candidates for selection and retest.
  (d) Reduction: For each test case t marked as candidate for selection, do the following: for every SD in the path of t marked as changed, do the following:
    (i) If there is no changed message (including those in combined fragments of UML 2.0 notation, if they exist) or order on the traversed path of t inside the SD, as per T.SD, then move to the next SD in the path list of t in T.IO.
    (ii) Otherwise, mark t as selected for retest.

The algorithms that implement this regression test selection technique are presented in the following subsections.

2.3. Test selection based on class diagrams

We first generate the set of changed methods from the class diagram, and then perform selection of test cases from unit and system level based on directly changed and indirectly affected methods.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

```
Algorithm: CDTestCaseSelection.
Input:     T=Set of system level tests.
           UT=Set of unit tests.
           M=Set of changed methods.
           IO=Original interaction overview diagram.
           IO'=Updated interaction overview diagram.
Output:    T'=Set of tests selected from T for retest.
           UT'=Set of tests selected from UT for retest.
Description: The algorithm selects two subsets of test cases from UT and T for retest.
CDTestCaseSelection(T,UT, M, IO, IO'):T',UT'
  begin
  T'=∅  --set of all tests from T marked for retest
  M = FindChangedMethods(CD, CD')
  T' = CDSystemTestSelection(M, SD.M, T)
  UT' = CDUnitTestSelection(M, UT, IO')

  return T,UT
  end
```

Figure 1. Class Diagram Test Case Selection Algorithm.

```
Algorithm: CDUnitTestSelection(M, UT, IO')
Input:     M=Set of changed methods.
           UT=Set of unit tests.
           IO'=Updated interaction overview diagram.
Output:    UT'=Set of unit tests selected from UT for retest.
Description: Select the unit test cases for methods in M and those whose calls precedes M in the same
             SD.
CDUnitTestSelection(M, UT, IO')
  begin
  UT'=∅  --set of all tests from UT marked for retest

  -- select the unit test cases for indirectly affected functions
  ∀ sd ∈ IO'
    begin
    M'=determine_calling_methods(M)  --set of methods indirectly affected by methods in M

  --select the unit test cases for the directly and indirectly affected methods
  ∀ m ∈ M' ∪ M
      -- select the unit test cases from UT that test m
        UT'=UT'∪ {ut ∈ UT: ut is a test case that tests method m}

  return UT'
  end
```

Figure 2. Class Diagram System Test Selection.

This is summarized in Figure 1. The algorithm FindChangedMethods() determines the set of the directly changed methods in the system. This algorithm is based on the initial and changes class diagrams by comparing the signature of methods for each class and updating the set of changed methods, M. The cost of this method is $O(Md)$. The algorithm CDSystemTestSelection() selects system test cases, T', for retest. It is based on the set of the directly changed methods, M, in the system. Every system test case traversing methods in M should be selected for retest. CDSystemTestSelection has one loop that selects test cases from T.SD that traverses the changed methods. For every row (path sequence) in T.SD we check whether any element of M exists in the T.SD path. This will cost $O(t.sd*l*\#T)$, where $t.sd$ is the number of rows in T.SD, l is the maximum length of paths traversed in T.SD, and $\#T$ is the number of all system tests.

Figure 2 gives the algorithm, CDUnitTestSelection, for selecting unit test cases, UT', for retest. We select the unit tests for the directly changed methods. Also, the algorithm determines the indirectly changed methods. This is done by determining the methods that call the directly changed methods. CDUnitTestSelection has two main loops, one to generate the indirectly changed methods for each SD and the other is for selecting unit test cases of the changed methods. The cost of finding the indirectly changed methods for all SDs is $O(\#SD*Md)$, where $\#SD$ is the number of SDs in the

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

```
Algorithm: IOTestSelection.
Input:      T=Set of system 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 system test cases to be selected for retest based on their
             traversal of a changed section in the IO diagram
IOTestSelection(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
      end

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

Figure 3. System Test Case Selection Algorithm.

IO diagram. Selecting unit test cases from UT that test the directly and indirectly changed methods costs $O(\#UT * Md)$, where $\#UT$ is the number of unit tests. Therefore, CDUnitTestSelection costs $O((\#SD + \#UT) * Md)$.

2.4. Test selection based on interaction overview diagram

Figure 3 summarizes the steps for selecting test cases based on the IO diagram. Since the IO diagram is a graph, IO comparison is done by doing a breadth first traversal of both graphs, original and updated, in parallel starting from the start node. If there is a change detected in an IO artifact, the test cases traversing that changed artifact will be selected for retesting. However, if a test case passes through a changed SD, then that test case is marked for refinement/reduction. Finding a set of changed SDs is done by comparing SDs with the same signature in IO and IO'. SDs (SD for IO and SD' for IO') are compared lifeline by lifeline. Lifelines are compared by ordered messages comparison between the lifeline in SD and its corresponding lifeline in SD'. If there is a change in one lifeline, we mark SD as changed. IOTestSelection will first compare every SD diagrams in IO with its respective diagram in IO'. Then, we invoke two methods: IOBasedClassification followed by SDBasedReduction. Comparing all the SDs of the interaction overview diagram with each other costs $O(\#SD * \#SDa)$ where $\#SD$ is the number of SDs in IO and $\#SDa$ is the maximum number of SD artifacts in the SDs. The costs of calling IOBasedClassification and SDBasedReduction will be discussed next.

IOBasedClassification algorithm, presented in Figure 4, will perform test case selection based on the interaction overview diagram before and after the update. This algorithm traverses the original and updated IO diagrams in parallel detecting changes along the path and selecting test cases that traverse that change. Based on the change type, the algorithm will either classify the test cases as selected for retest or as candidates for further refinement/reduction. Candidates are usually test cases that traverse a changed SD. This is because if an SD is internally changed, test cases

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

```
Algorithm: IOBasedClassification.
Input:     T=Set of system level tests.
           G=Nodes or branches in the original interaction overview diagram, IO.
           G'=Nodes or branches in the updated interaction overview diagram, IO'.
           ChangedSD=set of changed SDs
Output:    T'=Set of tests selected from T for retest.
           T"= Set of tests selected from T as candidates for retest.
Description: This algorithm starts at the start node of both IO and IO'; and traverses the edges and
nodes. If an edge is deleted or changed, all the test cases having that edge in their path will be marked
for retest. Similarly if there is a change in the decisions. If a test case passes through a changed SD,
then that test case is marked for refinement.
IOBasedClassification (G, G', T): T', T"
begin
  if G is marked as visited
    return; -- no need to go into loops
  else
    mark G as visited
  if signature(G)=signature(G') then -- same node
    begin
    switch G.type:
    case action:
          if (G ∈ ChangedSD)
              --G is a changed SD with the same signature
              --mark all test cases traversing G as candidates for selection
              ∀ t ∈ T with G in the path of t.IO, T'=T"∪ { t }
          [T',T"]= IOBasedClassification (G.outedge, G'.outedge, T)
    case edge:
          --move to the targets and continue processing
          [T',T"]= IOBasedClassification (G.target, G'.target, T)
    case decision:
          --mark all test cases traversing G as selected for retest
          ∀ decision d of G
              if d ∈ decision(G')
                  [T',T"]= IOBasedClassification (G. d, G'. d, T)
              else
              -- deleted decision, mark all test cases traversing this decision as
              -- selected for retest
                  begin
                  ∀ t ∈ T with d in the path of t.IO do
                      T'=T'∪ { t } --Retest
                      T"=T"-{ t } --in case already marked as Candidate
                  end
    default: return [T',T"]
    end
  else
    begin
    ∀ t ∈ T with G in the path of t.IO do
          T'=T'∪ { t } -- retest
          T"=T"-{ t } -- in case already marked as Candidate
    end
end
```

Figure 4. IO-Based Classification/Selection Algorithm.

traversing that SD might not necessarily traverse the change inside that SD. IOBasedClassification, is a graph coverage algorithm, it costs O(e * *#T* * l), where e is the number of edges in *IO*, *#T* is the number of system test cases, and l is the maximum length of the test paths per test case in T.IO.

After classifying the test cases based on the IO diagram as candidates for further analysis, the SD-based reduction algorithm (Figure 5) examines these candidates for the purpose of excluding those that do not cover changed elements. For each candidate test case, we consider the changed SDs traversed by this test and check whether the SD path covered by this test case in T.IO is different from the SD path in IO'. That is, SDBasedReduction compares method paths in changed SD for a candidate test case. The comparison particularly concentrates on changed messages in the SD and changed order of messages. Any of these changes leads to selecting the relevant candidate test for retesting. The computational cost is O(*#T"* * *#SD'* * l'), where *#T"* is the number of

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

```
Algorithm: SDBasedReduction.
Input:      T=Set of system level tests.
            T"= Set of tests selected from T as candidates for retest.
            ChangedSD=set of changed SDs
Output:     T'=Set of tests selected from T for retest.
Description: The SD based reduction algorithm goes through each SD classified from T, T" and
checks the traversal flow if changed between IO and IO' at the SD-level.
SDBasedReduction (T, T"): T'
begin
  ∀ t ∈ T"
    begin
    ∀ sd ∈ (ChangedSD ∩ set of SDs in the path of t.IO)
      --sd is marked as changed, so go to sd level
      begin
      if t.sd path is the same as in SD-level path in IO'.sd
        continue
      else
        --mark t for retest
        begin
        T'=T'∪ { t } (Retest)
        T"=T"-{ t } (in case already marked as Candidate)
        end
      end
    end
end
```

Figure 5. SD-Based Reduction Algorithm.

candidate tests for retest, *#SD'* is the number of changed SDs, and l' is the maximum method-call path traversed per SD from T.SD.

3. EMPIRICAL RESULTS AND DISCUSSION

The empirical procedure is composed of the following steps: design the application with UML diagrams; generate a suite (TS) of N unit and system test cases; determine and save the relationship between the UML diagrams (viz. CD, IO, and SD) and each test case; introduce a change into the application; reflect the change in the UML-design diagrams; determine which test cases in TS are modification-revealing with respect to the change; run the regression testing algorithms; compute the number of selected test cases (#R) as a percentage of N, inclusiveness (Inc), and precision (Pre) as metrics for assessing the results. Inclusiveness measures the percentage of modification-revealing tests selected by the regression testing technique, whereas precision measures the percentage of non-modification-revealing tests omitted in the selected tests.

For empirical evaluation, we use three systems with different versions to make up seven subject applications, which are illustrated in Table II; Md is the number of methods, C is the number of classes, and NIO is the number of nodes in the IO diagram. These systems were developed by students as class projects and the changes considered were found to be introduced by the students for updating the systems. An example of one system and the application of our technique is given in the Appendix. Since there are no existing techniques that are comparable to our technique in the UML diagrams employed, we compare our results with retest-all (i.e., #R=100% of N) and randomly select strategies. In the latter, the number of randomly selected test cases is made equal to the number selected by our regression testing technique (i.e., #R).

Table III gives the results for the seven applications, where N is the number of test cases in the initial test suite TS, and mr is the number of modification-revealing tests in TS; a test case is modification-revealing if it causes the output of the two applications (before and after change) to be different. The columns of results (in percentages) correspond to those produced by our technique, randomly select method, and retest-all method.

The results in Table III show the advantages of our proposed technique. The number of selected test cases for retesting ranges from 2 to 18% of N, which is a relatively small percentage. That

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table II. Description of the subject applications.

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

Table III. Results for the nine subject applications.

			Proposed technique			Randomly select			Retest-all		
App.	N	mr	#R%	Pre	Inc	#R%	Pre	Inc	#R%	Pre	Inc
P1V1	83	3	8	95	100	8	91	0	100	0	100
P1V2	90	7	8	100	100	8	92	0	100	0	100
P2V1	55	6	15	96	100	15	89	17	100	0	100
P2V2	55	1	2	100	100	2	98	0	100	0	100
P3V1	74	10	18	95	100	18	83	20	100	0	100
P3V2	77	2	10	92	100	10	89	0	100	0	100
P3V3	82	9	11	99	100	11	90	22	100	0	100

is, the proposed technique does reduce the time and effort, required for regression testing after program modification, in comparison with retest-all that blindly selects all tests.

The inclusiveness results (all 100%) indicate that our algorithms select all modification-revealing tests similar to the retest-all strategy. In contrast, the randomly select strategy performs poorly on inclusiveness since it selects tests without accounting for their relationship to the modification made. However, our technique is based on a safe strategy (selecting all modification-revealing tests) which might not be practical for large systems and large test suites. For large systems, it might be necessary to classify modification-revealing tests and select high-priority ones only; however, such prioritization may not be obvious. On the other hand, the precision results (92–100%) show that our technique performs well on omitting non-modification-revealing tests from the subset selected for retesting.

We note that the proposed technique assumes that the software developers update the design diagrams in a timely way, every time they introduce changes into the software system. This assumption becomes also a limitation for the proposed technique if, in practice, such timely updating does not occur. Another limitation is that concurrency methods are not accounted for in our technique. We also note a threat to the validity of the results given by the fact that the selected systems, developed by students, may not be strong representatives of the software domain.

4. CONCLUSION

We have presented a regression test selection technique for object-oriented software. This technique is design based and employs recent standard UML 2.0 diagrams. Empirical results have shown its capability to reduce the regression testing effort and to safely select modification-revealing tests and to omit non-revealing ones. Further, since our technique is based on UML-design and not on code, it also has the following advantages: (a) it works when design documentation only is available, (b) it makes test selection more manageable and leads to faster processing since design-based graphs and information is less complex than the corresponding code-based information, (c) it is independent of particular programming languages, and (d) it employs recent UML 2.0 diagrams, especially the newly introduced interaction overview diagram.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

The proposed technique assumes that the system has only one interaction overview diagram for the whole application. This would be realistic for small- or medium-sized applications but it becomes a limitation for large ones. Hence, the technique should be developed further to account for multiple or a hierarchy of interaction overview diagrams. Also, further work may investigate the level of details that should be included in interaction overview diagrams and its effect on the results. Furthermore, the scalability of the proposed technique should be empirically established on larger real-world software systems.

APPENDIX A: CASE STUDY—LIBRARY SYSTEM

The application is a support system for a simple library that provides items (books and magazines) to borrowers listed in the system. The system allows users to borrow items from the library and ensures that borrowers can only perform services that are associated with themselves, similarly for librarians. Borrowers can only search for books or magazines, and 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, and new items are regularly purchased. After 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 next in the list to take his/her place. The system consists of 8 classes, 29 attributes, 74 methods and 12 relationships.

The first change to this application is adding a new functionality to the system. A new fine system is included, where all late borrowers will be subject to fines which depends on the item type and the number of days.

Changes in the class diagram

For implementing the above-mentioned cahnge, a few additions are made at the class level. A dueDate attribute, GetDueDate(), and SetDueDate() are added to the Loan class. The dueDate is determined based on the lendingTime and the date of borrowing. The amountPerDay attribute is also 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 are added to the Title class: GetAmountPerDay() and SetAmountPerDay(). These two methods are used to manipulate the amount-PerDay attribute. The modified class diagram is shown in Figure A1 and the changes are listed in Table AI.

Changes in sequence diagram

The new system functionality resulted in a change inside the Return Item sequence diagram. The modification resulted in the addition of the GetDueDate() and the GetAmountPerDay() method when the date to return a book has been exceeded.

The change inside the GetLendingTime() method did not affect the SD Borrow Item since it occurred inside the method itself. This change is transparent to the sequence diagram and can only be noticed by signature comparison between the old and the new version of the application. The change inside the SD Return Item is mostly given by an *alternative* fragment, shown in Figure A2, which is used to denote the case where the borrower returns an overdue book.

Changes in the interaction overview diagram

At the interaction overview diagram level, the modifications to the system result in the addition of one sequence diagram (SD Print Receipt). The introduction of this SD resulted in the addition of 4 flows: F2, F3, F4 and F5. Only one flow was deleted: F6. A new decision (D1) was added to the system to denote whether the loan exceeded its due date or not. Figure A3 shows the IO diagram after the modification.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

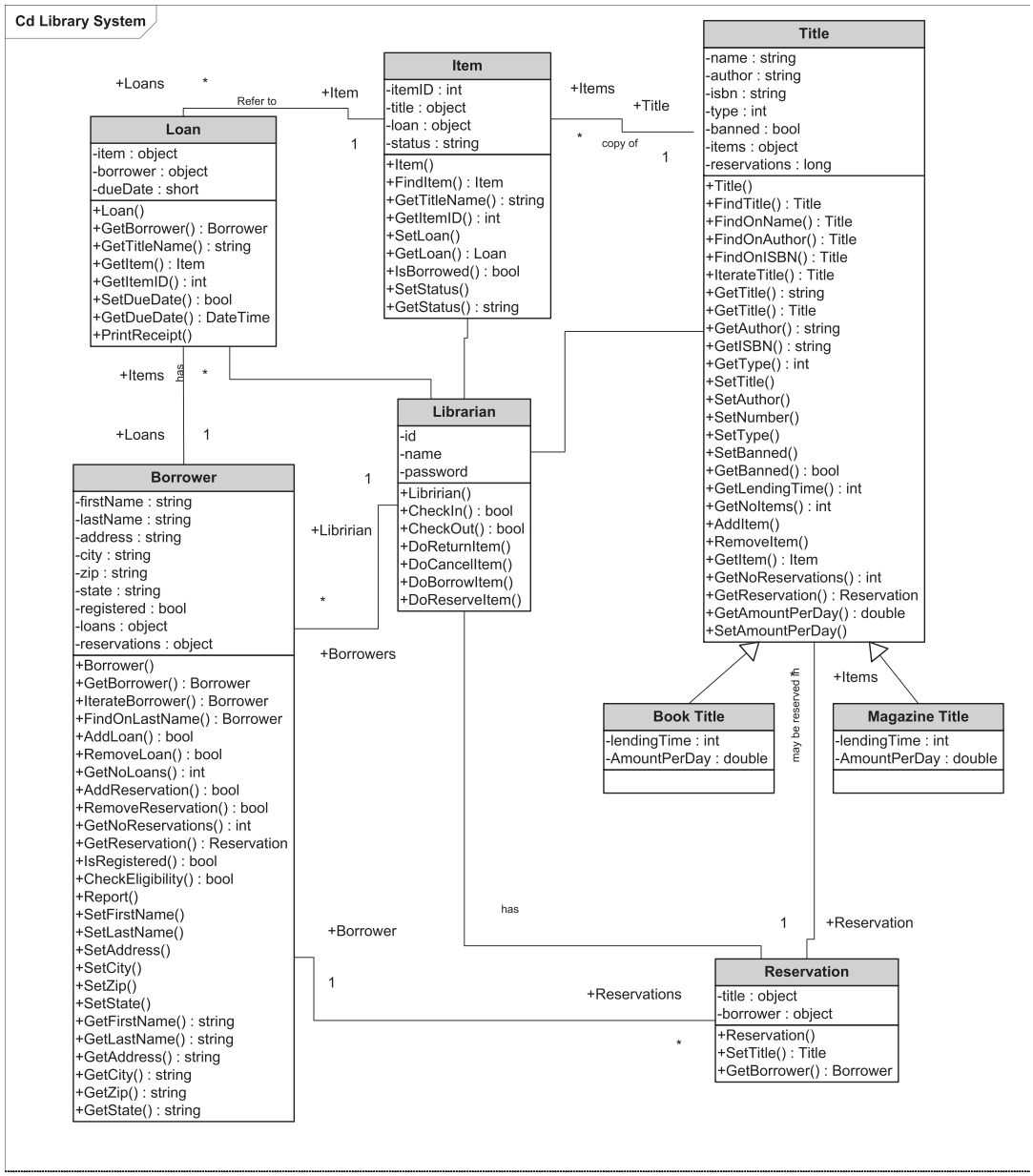

Figure A1. Modified library system.

Table AI. Library system changes.

	Total before update	Added	Modified	Deleted	Total after update
Attributes	29	3	0	0	32
Methods	74	5	1	0	79
Relationships	12	0	0	0	12
Classes	8	0	4	0	8

Set of test cases

The initial system test cases with their respective path in the interaction overview diagram (T.IO) are shown in Table AII. The paths of system tests inside the sequence diagrams (T.SD) are

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

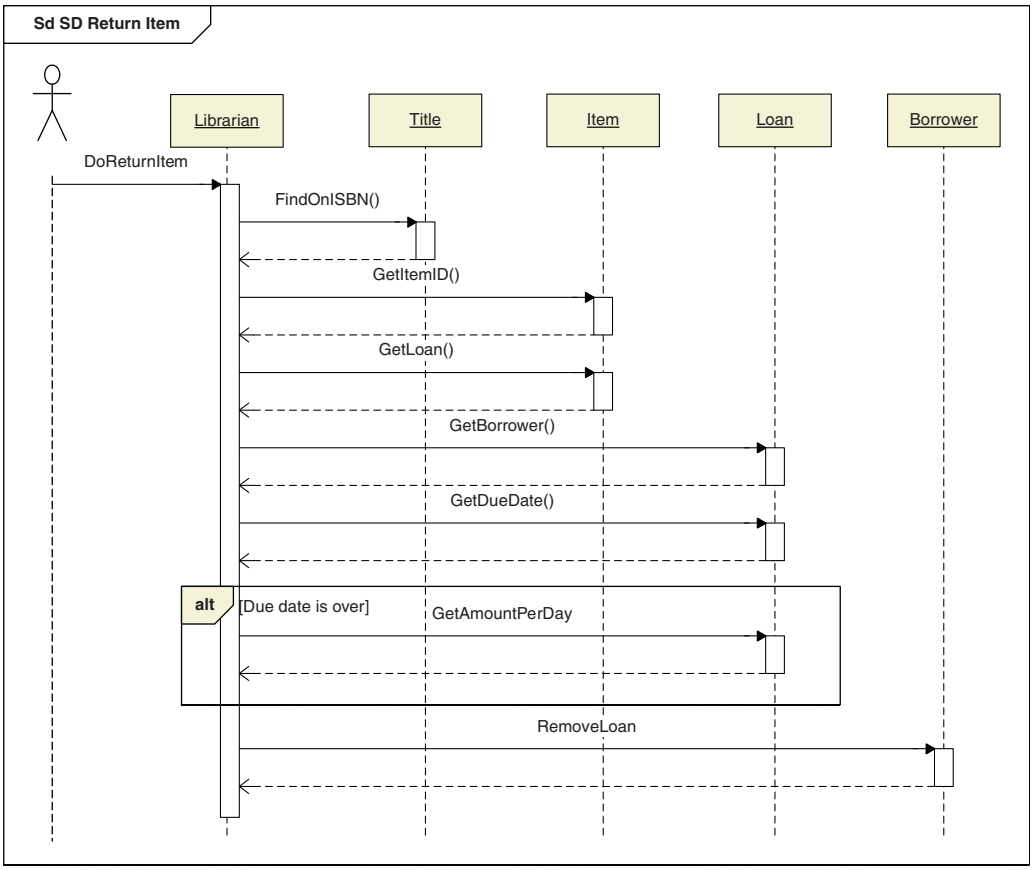

Figure A2. Modified Return Item sequence diagram.

presented in Table AIII. The initial unit test cases (UT) and the respective methods are shown in Table AIV.

Test case selection based on the class diagram

The changed method found by the *FindChangedMethods* algorithm is M={GetLendingTime()}. The sequence diagram referencing M, found by the *CDSystemTestSelection* algorithm is A8: SD Borrow Item (Figure A3). By inspecting the list of system test cases' paths in the sequence diagrams (T.SD), the system test case traversing A8 at the I.O level is only T5.

CDUnitTestSelection algorithm selects the unit test cases from both directly and indirectly changed methods. The unit test case executing M directly is UT32. Thus, the indirectly changed methods affected by the change of GetLendingTime() are: DoBorrowItem(), FindTitle(), GetTitle() and FindItem(). The unit test cases testing these methods (refer to Table AIV) are: UT44, UT16, UT22 and UT7. Hence, the selected test cases, based on CD changes are: T5, UT7, UT16, UT22, UT32 and UT44.

Test case selection based on the interaction overview diagram

Comparing the interaction overview diagram before and after modification yields that the set of changed SDs is {A1: SD Return Item}. The *IOBasedClassification* algorithm will select test cases based on the IO diagram. The direct change in the IO diagram is the addition of F2, F3, F4, F5 and the deletion of F6. D1 was also added. The test cases selected from T.IO for retest are the ones traversing A1, i.e., the system test case T1. Then, since the flow traversed in the sequence

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

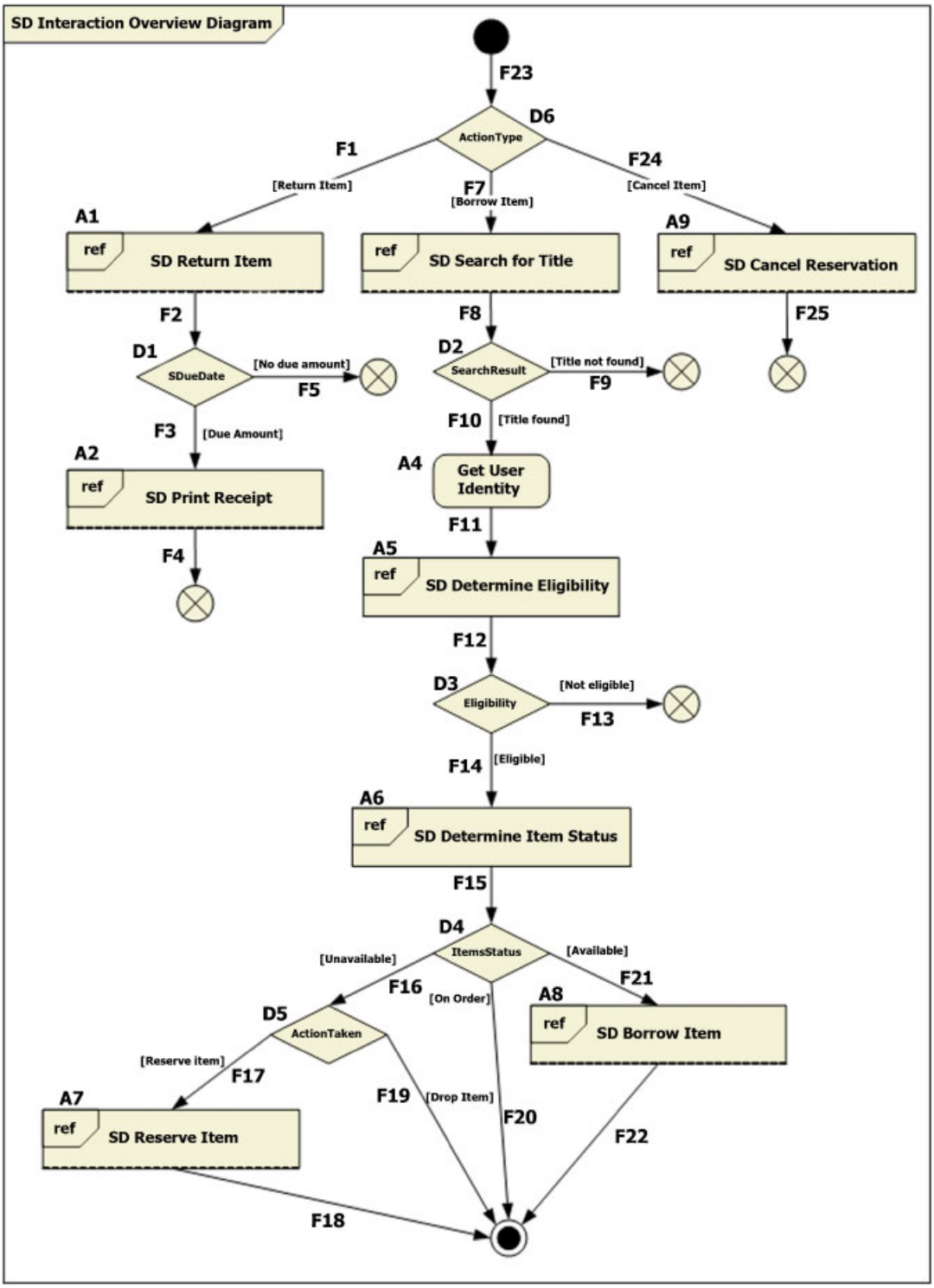

Figure A3. Modified interaction overview diagram.

diagram A1 is detected to have been changed by the *SDBasedReduction* algorithm, the test T1 will be marked for retest.

In conclusion, the final set of selected test cases consists of 5 unit tests and 2 system tests for a total of 7 out of 83 test cases.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table AII. System test cases in 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 AIII. System test cases in 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

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table AIII. *Continued*.

Test Case	SD	Path
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 AIV. 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

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table AIV. *Continued.*

Test Case	Method
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

ACKNOWLEDGEMENTS

We wish to thank the anonymous referees whose comments helped in improving the paper.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

REFERENCES

1. Bennett KH. The software maintenance of large software systems: Management, methods and tools. *Reliability Engineering and Systems Safety* 1991; **32**:135–154.
2. White L, Jaber K, Robinson B. Utilization of extended firewall for object oriented regression testing. *Proceedings IEEE International Conference on Software Maintenance*, 2005; 695–698.
3. Gupta R, Harrold MJ, Soffa ML. Program slicing-based regression testing techniques. *Software Testing*, *Verification and Reliability* 1996; **6**(2):83–111.
4. Rothermel G, Harrold MJ. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 1997; **6**(2):173–210.
5. Mansour N, Bahsoon R. Reduction-based methods and metrics for selective regression testing. *Information and Software Technology* 2002; **44**(7):431–443.
6. Mansour N, El-Fakih K. Simulated annealing and genetic algorithms for optimal regression testing. *Journal of the Software Maintenance*: *Research and Practice* 1999; **11**:19–34.
7. Zheng J, Robinson B, Williams L, Smiley K. Applying regression test selection for COTS-based applications. *Proceedings IEEE International Conference on Software Engineering*, May 2006.
8. Tarhini A, Fouchal H, Mansour N. Simple approach to testing web services based applications. *Proceedings of the Innovative Internet Community Systems Conference*. Springer: Berlin, June 2005.
9. Rothermel G, Harrold MJ, Dedhia J. Regression test selection for C++ software. *Journal of Software Testing Verification and Reliability* 2000; **10**:77–109.
10. Harrold MJ, Jones JA, Li T, Liang D, Orso A, Pennings M, Sinha S, Spoon SA, Gujarathi A. Regression test selection for Java software. *Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming*, *Systems*, *Languages*, *and Applications*, 2001; 312–326.
11. Taneja K, Xie T. DiffGen: Automated regression unit-test generation. *Proceedings 23rd IEEE/ACM International Conference on Automated Software Engineering*, September 2008; 407–410.
12. Wu Y, Chen M, Kao H. Regression testing on object-oriented programs. *Proceedings of the 10th International Symposium on Software Reliability*, 1999.
13. Le Traon Y, Jéron T, Jézéquel JM, Morel P. Efficient OO integration and regression testing. *IEEE Transactions on Reliability* 2000; **49**(1):12–25.
14. Chen Y, Probert RL, Sims DP. Specification-based regression test selection with risk analysis. *Proceedings of the Conference of IBM Center for Advanced Studies*, 2002.
15. Beydeda S, Gruhn V. Intergrating white- and black-box techniques for class-level regression testing. *Proceedings of the 25th International Computer Software and Applications Conference*, 2001; 357–362.
16. Korel B, Tahat LH, Vaysburg B. Model based regression test reduction using dependence analysis. *Proceedings IEEE International Conference on Software Maintenance*, 2002.
17. Wu Y, Offutt J. Maintaining evolving component based software with UML. *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, 2003.
18. Briand LC, Labiche Y, Buist K, Soccar G. Automating impact analysis and regression test selection based on UML designs. *Proceedings IEEE International Conference on Software Maintenance*, Montreal, 2002.
19. Farooq Q, Iqbal MZ, Malik Z, Nadeem A. An approach for selective state machine based regression testing. *Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, London, U.K., 2007; 44–52.
20. Pilskalns O, Uyan G, Andrews A. Regression testing UML designs. *Proceedings 22nd IEEE International Conference on Software Maintenance*, 2006; 254–264.
21. Batra G. Model-based software regression testing for software components. *Proceedings of the 3rd International Conference on Information Systems*, *Technology and Management*, Prasad SK (ed.), vol. 31. Springer: Berlin, March 2009.
22. Arlow J, Neustadt I. *UML 2 and the Unified Process* (2nd edn). Addison-Wesley: U.S.A., 2005.

AUTHORS' BIOGRAPHIES

Nashat Mansour is a Professor of Computer Science at the Lebanese American University. He received BE and MEngSc in Electrical Engineering from the University of New South Wales, and MS in Computer Engineering and PhD in Computer Science from Syracuse University. His research interests include: software testing, application of metaheuristics and data mining to real-world problems, and protein structure prediction.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Husam Takkoush is currently a computing professional at MDS systems. He received an MS in Computer Science from the Lebanese American University and a BS in Computer Science from the American University of Beirut. His research interests include: software design and testing.

Ali Nehme is currently an IT consultant. He received his BS and MS in Computer Science from the Lebanese American University. His research interests are in software engineering.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/smr