Toward a Comprehensive and Systematic Methodology for Class Integration Testing

By

Yihong Wang, B. Eng.

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

Master of Applied Science

**Ottawa-Carleton Institute of
Electrical and Computer Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, K1S 5B6
Canada**

May 2003

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisisitons et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-83483-2

Our file Notre référence

ISBN: 0-612-83483-2

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

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

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

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

Canada

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

**Toward a Comprehensive and Systematic Methodology for
Class Integration Testing**

Submitted by

Yihong Wang, B. Eng.

in partial fulfillment of the requirements for
the degree of Master of Applied Science

Chair, Professor Rafik A. Goubran

Thesis Co-Supervisor, Professor L. Briand

Thesis Co-Supervisor, Professor Y. Labiche

Carleton University

May 2003

Abstract

Any class integration strategy must answer the following two questions: (i) In what order will classes be integrated and class interfaces be exercised? (ii) Which test design technique(s) should be used to exercise each interface?

With regard to (i), the issue of ordering class integration in the context of integration testing has been discussed by a number of researchers. More specifically, strategies have been proposed to generate a test order while minimizing stubbing. Recent papers have addressed the problem of deriving an integration order in the presence of dependency cycles in the class diagram. Such dependencies represent a practical problem, as they make any topological ordering of classes impossible. This thesis presents a strategy that integrates two existing methods, i.e. the methods proposed by Tai & Daniels and Le Traon et al, aimed at “breaking” cycles so as to allow a topological order of classes. The new strategy combines some of the principles of these two existing approaches and addresses some of their shortcomings. This thesis evaluates the new strategy by precisely investigating the above three approaches in terms of principles, advantages and drawbacks, and providing both analytical and empirical comparisons based on five case studies.

With respect to (ii), this thesis is a first attempt towards a comprehensive, systematic methodology for class interface testing in the context of client/server relationships. The proposed approach builds on and combines existing techniques. It first consists in selecting a subset of the method sequences defined for the class testing of the client class,

based on an analysis of the interactions between the client and the server classes. Coupling information is then used to determine the conditions, i.e., values for parameters and data members, under which the selected client method sequences are to be executed to exercise the interactions. The approach is illustrated by means of an abstract example and its cost-effectiveness is evaluated through two case studies.

Acknowledgements

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

I would also like to thank my family and my friends who urged me to go on when my patience ran out and confidence low. I thank them for their advice, support, tolerance and, most of all, for being my friends.

Table of Contents

Chapter 1	Introduction	1
Chapter 2	Related Works	3
2.1	Related works for class integration order	3
2.2	Related works for class integration testing	6
Chapter 3	Class Integration Order	11
3.1	Preliminary Issues in Assessing Integration Orders	11
3.1.1	Classes Stubbed versus Stubs	11
3.1.2	Stubbing associations versus other relationships	13
3.1.3	Object Relation Diagram	14
3.2	Existing graph-based approaches	15
3.2.1	Tai & Daniels	16
3.2.2	Le Traon et al.	19
3.3	A new approach	22
3.3.1	Weight computation	23
3.3.2	Application of the new approach to the example	25
3.4	Discussion of the above three approaches	27
3.5	Case Studies	29
Chapter 4	Class Integration Testing	34
4.1	The strategy for deriving the triggered server method sequence	35
4.1.1	Interprocedural Control Flow Graph for client methods	36
4.1.2	Server method sequences triggered by client methods	39
4.1.3	Server method sequences triggered by client method sequences	40
4.1.4	Removing redundancy	42
4.1.5	Test sequences	47
4.1.6	Client interactions with several server classes	48
4.2	Coupling-based testing of method interactions	49
4.2.1	Data-Flow analysis definitions	50
4.2.2	Coupling classification	52
4.2.3	Coupling-based integration criteria	56
4.2.4	Issues	56
4.3	Automation	57
4.3.1	The required information and the metamodels	58
4.3.2	The implementation of the tool	64
4.4	Case studies	69
4.4.1	Description of the case studies	70
4.4.2	Testing Sequences	72
4.4.3	Mutants	77
4.4.4	Results	80
Chapter 5	Conclusions	85

References	88
Appendix A Algorithms for deriving the class integration order	92
A.1 Procedure classLevel.....	92
A.2 Function sccArray	93
A.3 Procedure scc.....	94
A.4 Function collapse.....	95
A.5 Function TaiArray	96
A.6 Procedure level_sort	96
A.7 Function maxWeightEdge.....	97
A.8 Function deleteEdgeGraph.....	98
Appendix B More details on the running example.....	100
Appendix C Instantiation of the tuple metamodel	102
Appendix D Algorithms for the generation of tuples	104
D.1 Procedure combineMethodTuples.....	104
D.2 Procedure addTuples	105
D.3 Procedure removeRedundancyOne	106
D.4 Procedure removeRedundancyTwo	107
D.5 Procedure removeRedundancyThree	108
D.6 Procedure checkPendingTuples	109
D.7 Function checkIncludedTuple	110
Appendix E Instantiation of the coupling information metamodel	112
Appendix F Algorithms for coupling based integration test requirements	115
F.1 Function computeCallCouplingPath	115
F.2 Function computeParameterCouplingPaths	115
F.3 Function computeSharedAttributeCouplingPaths.....	117
F.4 Function computeExternalDeviceCouplingPath	118
Appendix G JAdvisor - Details	121
Appendix H Coupling paths for JAdvisor.....	123
Appendix I Use of Category-Partition for JAdvisor.....	124
Appendix J Linked List – Details	125
Appendix K Tuples for Linked List.....	127
Appendix L Coupling paths for Linked List	132
Appendix M Use of Category-Partition for Linked List	133

List of Figures

Figure 1	State Design Pattern.....	13
Figure 2	Example ORD.....	16
Figure 3	Intermediate steps of Tai & Daniels' approach for Figure 2	17
Figure 4	Major and Minor Level Numbers	18
Figure 5	Le Traon's approach for Figure 2, starting with node G (intermediate results).....	20
Figure 6	Le Traon's approach for Figure 2, starting with node G	21
Figure 7	Example SCC.....	24
Figure 8	Applying the new approach to Figure 2 (intermediate results)	25
Figure 9	Applying the new approach to Figure 2.....	27
Figure 10	Running example (source code)	36
Figure 11	Examples of annotated CFG and ICFG from Figure 10.....	39
Figure 12	Metamodel for tuples	61
Figure 13	Metamodel for coupling information.....	63
Figure 14	Overview of CIIA	65
Figure 15	Subsystem decomposition for CIIA.....	66
Figure 16	GUI for Class Interaction module.....	68
Figure 17	GUI for Method Interaction module.....	68
Figure 18	Jadvisor subsystems and class diagram in subsystem planner	71
Figure 19	Class diagram for Linked List	72
Figure 20	Distribution of mutants across operators and methods (Jadvisor).....	78
Figure 21	Distribution of mutants across operators and methods (Linked List).....	80
Figure 22	ICFGs for methods <code>mA1()</code> , <code>mA2()</code> and <code>mA5()</code> in Figure 10	100
Figure 23	Instance of the tuple metamodel for client methods <code>mA1</code> and <code>mA2</code> (Figure 10).....	102
Figure 24	Instance of the tuple metamodel showing tuples for client sequence <code>mA1.mA2</code>	103
Figure 25	Instance of the metamodel (excerpt) for a call coupling	112
Figure 26	Instance of the metamodel (excerpts) for parameter coupling	113
Figure 27	Instance of the metamodel (excerpts) for shared attribute and external device coupling.....	114

Figure 28	Java code chunk for class <code>StudentPlan</code>	121
Figure 29	Java code chunk for class <code>StudentSemesterPlan</code>	122
Figure 30	Source code for class <code>List</code> (excerpt).....	126
Figure 31	Source code for class <code>Node</code> (excerpt)	126

List of Tables

Table 1	Summary of the results obtained from the application of the three approaches.....	29
Table 2	Summary of the information for the five case studies	30
Table 3	Number of stubs produced by the three strategies (mean).....	30
Table 4	Attribute and method costs for orders produced by the three strategies (mean, median).....	32
Table 5	Server method sequences triggered by method <code>mA2()</code> in Figure 9.....	40
Table 6	Concatenations of Server Sequences on attribute <code>b2</code> for Client Sequence <code>mA1()</code> . <code>mA2()</code> . <code>mA5()</code>	42
Table 7	Tuples for client methods and client method sequences from Figure 9 .	45
Table 8	Applying the three redundancy criteria from Table 7.....	46
Table 9	Tuples for Jadvisor.....	74
Table 10	Applying redundancy criteria from Table 9 (Jadvisor)	75
Table 11	Client method sequences for Linked List	76
Table 12	Results for Jadvisor.....	81
Table 13	Results for Linked List	83
Table 14	Server method sequences triggered by method <code>mA5()</code> in Figure 9.....	101
Table 15	The coupling paths in Jadvisor	123
Table 16	The categories and choices for the parameters	124
Table 17	The test cases for category partition test.....	124
Table 18	Tuples for methods in class List	127
Table 19	Tuples for Linked List (from client method sequences in Table 11)....	131
Table 20	Selected tuples for Linked List according to redundancy criteria	131
Table 21	Coupling paths between List methods and Node methods	132
Table 22	The choices for the parameter and environment object.....	133
Table 23	The test cases for category-partition test	133

CHAPTER 1 INTRODUCTION

Integration testing explores how components (i.e., modules, sub-systems) interact with each other, assuming that the components have all passed their local tests [5]. This thesis focuses on integrating classes, though many of the ideas can be generalized to testing cooperating subsystems, which are simply an encapsulation mechanism for hiding class public interfaces in a subsystem from other subsystems. Class integration testing aims at detecting faults that cause inter-class failures [6]. There are two problems which lie within the class integration testing: the first one is deciding the order in which the classes should be integrated and class interfaces be exercised; the second is choosing test design technique(s) to test the class interactions, following the order derived previously.

Determining a class integration order is important because it affects the order in which classes are developed as well as the order in which classes are tested. In this thesis, we are interested in the latter. Since there exist dependencies among classes, testing classes may need stubs, i.e., pieces of software that need to be developed to simulate the response of components (i.e., classes in an object-oriented context) that have not been implemented yet but on which the classes under test depend on. Furthermore, a class integration order determines the order in which interclass faults are detected, and impacts the preparation of test cases. One of the main objectives of a class integration order is then to minimize the costs related to testing activities: e.g., building the test harness (drivers, stubs, ...). This can be done in different ways: reducing the number of stubs along with the complexity of the stubs, reducing the number of drivers, In this thesis,

we focus on minimizing the number of stubs. This thesis proposes a strategy and presents a systematic, analytical (by means of an abstract example), and empirical (by means of five different application systems) evaluation of the new strategy, along with two other existing techniques. It is shown that the new strategy outperforms the two existing strategies in terms of the number of stubs required, and the number of attributes or methods those stubs may need to emulate.

Regarding the second issue, this thesis investigates ways of performing class interface testing in the context of client/server relationships. In fact, testing client/server class interactions has received little attention in the literature, and existing works only partially address it. Since the unit testing of the client class is performed using a stub simulating the server class, which is an incomplete implementation of the server class, most interface faults remain undetected. Therefore it is necessary to perform integration testing of the interacting classes, in which the actual server class is used (instead of a stub). In this thesis, we proposed a precise methodology and automatable algorithms to test class interfaces.

The rest of the thesis is organized as follows: Chapter 2 reports on the related work regarding the two issues in class integration testing. Our two approaches, which build on and combine existing techniques, are then introduced in Chapter 3 and Chapter 4 respectively, where we focus in turn on the class integration order and class integration testing. The conclusion is drawn and the future work is outlined in Chapter 5.

CHAPTER 2 RELATED WORKS

Since the thesis addresses two problems with regard to class integration testing, the related works presented below consist of two parts, one for each problem accordingly.

2.1 Related works for class integration order

A number of papers have provided strategies and algorithms for deriving an integration and test order from dependencies among classes in the system class diagram [20, 21, 23, 38]. The objective of all these approaches is to minimize the number of test stubs to be produced, as this is perceived as a major cost factor of integration testing. Indeed, stubs are pieces of software that have to be built (cost) in order to simulate parts of the software that have either not been developed yet or are still to be unit tested, but are needed to test classes that depend on them.

Kung et al. are the first researchers to address the class test order problem and they show that, when no dependency cycles are present among classes, deriving an integration order is equivalent to performing a topological sorting of classes based on their dependency graph – a well known graph theory problem¹. In the presence of dependency cycles, the proposed strategy consists of identifying strongly connected components (SCC²s) and removing associations until there is no cycle in the SCCs. Note that the cycle in this

¹ The topological sorting of a graph G consists of numbering the vertices of G (e.g., labeling them with numbers $1, \dots, n$) such that for each edge, the label associated with the source vertex is strictly lower than the label associated with the target vertex. Furthermore, it can be shown that there is a topological sort for any directed acyclic graph [18].

² A SCC of a graph is a sub-graph such that for any pair of vertexes v, w , there is a path from v to w and vice versa.

thesis refers to the elementary circuit, i.e., each class appears once and only once in each circuit. However, Kung and his colleagues do not provide precise solutions when there is more than one candidate association for cycle breaking — in this case they simply perform a random selection. They mention that a possible solution would involve the use of the complexity of the associations involved in cycles. Unfortunately, such dependency cycles are commonplace in real world systems and hinder any topological sorting of classes. Cycles are usually rare in analysis class diagrams but then it is common, as the design progresses, to add classes and relationships in order, for example, to improve performance or maintainability. As a result, the class diagram usually contains cycles by the end of the low-level design.

Existing solutions to the foregoing problem are based on the principle of “breaking” some dependencies to obtain acyclic dependencies between classes. In our context, a broken dependency implies that the target class will have to be stubbed when integrating and testing the source class. Tai & Daniels [38] propose a 2-stage algorithm that deals with dependency cycles. However, in cases where class associations are not involved in cycles, their solution is sub-optimal in terms of the required number of test stubs. Le Traon et al. [23] propose an alternative strategy based on graph search algorithms that recognize strongly connected components, and that arguably yields better results. One issue, though, is that this algorithm is not fully deterministic in the sense that, depending on some arbitrary decision (e.g., the initial vertex (class) of the search, and the search itself), the algorithm may yield significantly different results. Furthermore, since the model used does not have any information regarding the kind of dependency (inheritance,

association or aggregation), this approach may lead to the removal of an inheritance or aggregation relationship.

Kung et al. [20], as well as others before them [37], point out that association relationships are usually the weakest links in a class diagram, i.e., the links involving the fewest dependencies and therefore the least stub complexity if broken. Kung et al. further argued that every cycle in a class diagram contains at least one association. Though no demonstration is provided in their paper, this is due to the fact that inheritance and aggregation relationships are defined as transitive, irreflexive, and asymmetrical [25]. It is then easy to show that a cycle involving only aggregation and inheritance relationships would lead, assuming transitivity, to symmetrical or reflexive dependencies between instances, thus transgressing one of the basic properties of these relationships. As a simple example, if classes A and B are related through compositions with B and C, respectively, and C inherits from A, then A instances are composed of C instances (transitivity), which are themselves A instances, thus transgressing the irreflexivity property.

It is worth noting that other approaches to the class integration order problem have recently been proposed [8, 22]. These approaches make use of Genetic Algorithms (GAs), a global optimization technique based on heuristics, which has been developed by the artificial intelligence community over the years. Though this is a promising avenue of research in order to facilitate the use of more complex stubbing complexity definitions [8], we do not investigate such an approach in this thesis, as we prefer to focus on graph-

based techniques which are simpler to apply and more efficient to use, and as GAs involve a different set of challenges.

Last, there exist other integration strategies that are not based on the class diagram, as derived from the software design or reverse engineering. They rather associate a functional description with a set of classes. For instance, in [17], Atomic System Functions (ASF), which involve system inputs and outputs and exercise Method/Message paths between objects, derive the integration test of classes. These ASFs correspond to a functional decomposition of the system, which is similar to use cases. The objective of this strategy is not to minimize test stubs but to execute complete, end-user functionalities, in an incremental manner during integration. Similar strategies using use cases can be found in [6, 27]. Since these strategies are not explicitly based on the class diagram, they will not be detailed and compared in this thesis. Though this is a topic for future research, it is likely that in practice the two sets of strategies would have to be combined.

2.2 Related works for class integration testing

As for the class integration testing, a first set of existing strategies can be described as functional and grey-box, since they both concentrate on the methods involved in interactions between classes/objects (without considering the details of the methods, however) and associate a functional description with the interacting classes. As introduced in the previous section, the ASFs in [17] and the UML-based strategies, such as [6, 27], can be classified into this set.

Other existing integration strategies are purely white-box. The first one to be discussed addresses the selection of client method sequences that exercise the interaction between a client class and one of its server classes [4], but does not address the generation of test data. Bashir & Goel's strategy for class integration testing consists of reusing method sequences defined for the class testing of the client class and re-executes a subset of these sequences, using the actual server classes instead of stubs, as is normally the case during class testing. Selecting a subset of client sequences to re-test is based on analyzing the server methods called by the client methods and devising the sequences of server methods that are triggered by the sequences of client methods. Client method sequences that trigger the same server method sequences are then considered redundant, as it is deemed that they exercise the client/server interface in the same way, and all but one are eliminated. Once this trimming process is complete, the remaining sequences have then to be executed to complete the integration testing of the client/server class pair.

Note that, though Bashir & Goel describe the derivation of the client method sequences in the context of their specific class testing technique, their integration testing strategy is independent from it and can be used in combination with other class testing techniques, such as using a state-based representation of the class [3, 30], or using constraints based on the methods' contracts [10].

Bashir & Goel's strategy is interesting since all its steps can be automated and it specifies how a subset of class testing sequences can be reused, thus drastically reducing the effort of class integration testing. However, it suffers major limitations, which we address in

this work as a first step toward specifying a comprehensive methodology for class integration testing.

First, the approach does not consider the control flow of both client and server methods. For example, a client method can call server methods several times, possibly not always in the same order (depending on the control flow in the client method). As a consequence, one sequence of client methods can trigger more than one sequence of server methods, whereas [4] assumes that there is a one-to-one mapping between them.

Second, when analyzing sequences of server methods triggered by sequences of client methods, the approach in [4] does not account for the instance(s) on which server methods are invoked. For instance, saying that sequence $m1.m2$ (methods of the server class) is triggered by a given sequence of client methods is not correct, when methods $m1$ and $m2$ are called on two different instances of the server class. In this case the sequence of client methods triggers two sequences of server methods: $m1$ and $m2$.

Another recent work concentrates on identifying data flow interactions between pairs of client methods through attributes that are instances of a server class [26]: the first method in the pair modifies the attribute, and the second uses the attribute. The data-flow analysis in [26] uses an adaptation of Harrold et al's notion of intra-class def-use pairs [14]. They only consider attributes (data members) that are object references: a definition/use of an attribute does not only apply to the attribute value, but is also a modification/access of the state of the object referenced by the attribute. Our approach furthers this work in two ways. First, since integration testing refers, in our context, to testing interfaces between classes to ensure that they have consistent assumptions and communicate correctly [5],

we are also interested in the details of the class methods that interact (e.g., where the uses and definitions in the corresponding control-flow graphs are located). Second, we do not restrict the data-flow analysis to attributes of the client class. We also want to consider interactions through method parameters and local variables (which may not be object references).

Note that the functional and white-box methodologies above are complementary. The former allows the selection of a subset of classes to be integrated in the next release, so as to be able to deliver end-user functional increments. The latter allows for the testing of class interactions in this subset in a stepwise manner, once an integration order has been determined. Next, once method sequences are determined, the conditions (values for parameters and data members) under which pairs of methods (client and server methods) are executed remains to be determined. This is the focus of [16] in a procedural programming context. The authors define four coupling criteria that are aimed at triggering the four different coupling types between modules they have identified (e.g., parameter coupling that refers to parameter passing). These criteria are based on traditional data-flow criteria, and have been extended to address polymorphic relationships [1, 2].

This thesis presents two approaches to deal with the above two issues in class integration testing. The first approach handles the problem of class integration order. The new approach is based on Le Traon et al. and Tai & Daniels, but addresses important weaknesses. It also identifies SCCs with a graph-based algorithm, but differs in the way in which cycles are broken (e.g., it does not remove inheritance or aggregation

relationships). The second approach, which is to test class interactions, adapts and extends Bashir & Goel's strategy (addressing its drawbacks), and combines it with an adaptation of the coupling criteria. Therefore, the two issues in the class integration testing mentioned above have been addressed, and an automatable, comprehensive class integration testing methodology has been devised.

CHAPTER 3 CLASS INTEGRATION ORDER

This chapter discusses the issue of class integration order. Section 3.1 first introduces some important issues in assessing integration ordering strategies. Since the strategy proposed in this thesis is based on Le Traon et al. and Tai & Daniels, their rationales are described in Section 3.2 through the use of an example. In Section 3.3, the new strategy is introduced and illustrated using the same example. The discussion on the three approaches in terms of their strengths and drawbacks is given in Section 3.4, and Section 3.5 presents an empirical evaluation of the three approaches using five different application systems.

3.1 Preliminary Issues in Assessing Integration Orders

Before discussing integration order strategies in detail, it is worthwhile clarifying a number of practical issues.

3.1.1 Classes Stubbed versus Stubs

One important question is to determine what the criterion should be used to evaluate strategies that break cycles. One possibility, which is implied by both Le Traon et al. and Tai & Daniels [23, 38], is to count the number of classes to be stubbed. However, when a client class uses a stub to be tested, this stub usually emulates the minimal subset of the server class functionality that is required for that specific client. Indeed, stubs need to remain as simple as possible (e.g., in terms of control and data flow) so as not to become error-prone: “If the stub is realistic in every way, it is no longer a stub but the actual

routine” [5]. Ideally, stubs should only contain sequential control flow so as to require little testing effort themselves. Therefore, when a server class is used by several client classes, we usually obtain at least as many stubs as client classes, if we want to minimize risks. It is in any case more likely that the number of stubs will be proportional to the number of client classes of classes to be stubbed, as opposed to just the latter ones. This leads us to the position that the number of stubbed classes multiplied by the number of their client classes is probably a more realistic evaluation criterion than the number of stubbed classes alone. Le Traon et al [23] refers to this as the number of *specific* stubs, and we reuse this term below. They, however, refer to the stubbed classes as *realistic* stubs. Since, as discussed above, the number of stubs is more likely proportional to the number of client classes of classes to be stubbed, we choose not to use this term any further.

The number of specific stubs still remains, nevertheless, an estimate of the cost of a class test order, as not all broken dependencies lead to stubs of similar complexity. This simplifying assumption is used by all the algorithms referenced above and may seem simplistic, but our observation has been that, in terms of the number of methods and attributes in the target class, frequency distributions are similar across test orders, so that orders entailing larger numbers of stubs are still likely to be more expensive than orders with lower numbers of stubs [9].

When assessing the stubbing cost involved in a test order, one may want to look further than the number of stubs so as to get more precise indicators of the cost of producing stubs. To increase the precision of our evaluations across our case studies, we not only

show the number of stubs but also what they correspond to in terms of methods and attributes (from the target class of the broken dependency) that may potentially be involved in the stubs. Though these two indicators are admittedly not perfect, they nevertheless improve the precision of the results reported.

3.1.2 Stubbing associations versus other relationships

As mentioned above (and explained in Chapter 2), Le Traon et al [23] may lead to “breaking” aggregation or inheritance relationships, thus leading to the stubbing of these relationships. If we take the state design pattern [12] as an example, the `Context` class whose state is being modeled is related, through an aggregation relationship, to an abstract class `State` (Figure 1). Every time the `Context` class receives a message, its state is likely to change, thus requiring a message to be sent to an instance of one of the subclasses of `State`. In turn, the subclasses instances may invoke `Handle` methods in the `Context` class. So we see that the dependencies are very tight and we claim that this is usually the case with aggregation relationships.

Figure 1 State Design Pattern

Regarding inheritance, stubbing a parent class would imply stubbing most inherited methods. This follows from the fact that inherited methods should, in many instances, be tested in the new context represented by a subclass, even though they may have been

fully tested in the parent class. In [35] the authors show, in the light of the adequacy criterion defined in [41], that inherited code needs to be retested. Furthermore, Harrold and McGregor [13] propose an incremental strategy for testing inheritance hierarchies that minimizes re-testing of inherited methods. This strategy requires that inherited methods already tested in the parent class A be retested in the child class B only if they interact with B's methods. In practice, when inheritance is properly used to extend the functionality of parent classes (Liskov principle [24]), most parent class methods inherited are usually interacting with at least one child class member. As a result, when breaking inheritance relationships, the resulting stubs would almost have to be the entire parent class, as all inherited, non-overridden methods would have to be tested in the subclass to consider it unit tested. So, we clearly see from the discussion above that any method breaking cycles should aim at only breaking association relationships, so as to require the development of stubs that are economically viable.

3.1.3 Object Relation Diagram

It ensues that the algorithms we are about to describe need a model of class relationships that allows the identification of breakable and unbreakable relationships. Such models have been defined and used in previous works [19, 38]. The Object Relation Diagram (ORD) identifies inheritance (*I*), aggregation (*Ag*), and association (*As*) relationships, the latter being the only breakable relationship [20]. The Test Dependency Graph (TDG) extends the ORD by including relationships at the method level [22, 23]. However, the mapping from UML to the TDG, proposed in [22, 23], does not distinguish UML aggregation and composition relationships, making the latter breakable. Since we do not

need information at the method level when devising a class test order and we do not want to break composition relationships, we will use the ORD in this thesis. The ORD can be derived from a UML class diagram or reverse engineered. The mapping between the relations in the ORD and those in a UML class diagram is as follows: *As* is mapped to Association, simple Aggregation or Usage; *Ag* is mapped to Composition and *I* is mapped to inheritance [8]. The rationale for the mapping of relations other than inheritance is the following: (i) as discussed above, we want to avoid breaking compositions in UML class diagrams; (ii) like in previous works, *Ag* relations are the relations in the ORD that are not supposed to be broken (as opposed to *As*).

3.2 Existing graph-based approaches

We follow below a chronological order to present existing work. The following running example (see the ORD in Figure 2) is used to illustrate the techniques and discuss their implications. It is a modified version of the example used in [38]. Classes are labeled with capital letters and dependencies are labeled according to their type: *As* for associations, *Ag* for aggregations, *I* for inheritance. Although this example may seem somewhat complex and dense, it is aimed at supporting our argument about using a minimum number of classes and dependencies. The case studies in Section 3.5 will present real examples.

Figure 2 Example ORD

3.2.1 Tai & Daniels

The strategy proposed by Tai & Daniels [38] assigns each class in the class diagram a *major* and a *minor* level number. Those numbers are then used to devise an integration order. Major level numbers are assigned based on inheritance and aggregation dependencies only. Since all association dependencies are ignored, there is no cycle and topological sorting can be applied. Then, minor level numbers are assigned, within each major level, based on association dependencies only. Here cycles may appear and must be broken in order to apply topological sorting. In that case, a $weight(d_i)$ function is defined for each association d_i in each major level as the number of the incoming dependencies of the origin node of d_i plus the number of outgoing dependencies of the target node of d_i . The rationale is that the higher the weight, the more likely it is that breaking a dependency will break a larger number of cycles, and therefore the dependency with the highest weight is selected to break cycles.

(a) Figure 2 without associations for major level numbers

(b) Associations inside major level 1 for minor level numbers

Figure 3 Intermediate steps of Tai & Daniels' approach for Figure 2

Using the example above, we derive the major level and minor level numbers for each class, applying the algorithms provided in [38]. Figure 3.a shows the ORD considered for the determination of major levels: the ORD is derived from Figure 2 and contains only aggregation and inheritance relationships. There is no cycle in the ORD and topological sorting assigns major levels: 1 to classes E, A, and C; 2 to classes F and H; 3 to classes D and B; 4 to class G. Figure 3.b then shows classes at major level 1 (the only major level containing a cycle when considering associations) and their association relationships so as to determine their minor level numbers. The weights of the different associations are: $W(C, A) = 2$; $W(E, A) = 2$; $W(A, C) = 4$, $W(C, E) = 2$. Association (A, C) is broken and classes A, E, and C have minor level numbers 1, 2, and 3, respectively.

Figure 4 shows the example classes sorted according to their major level, from top to bottom, and indicates the minor level numbers within the class boxes, denoted as decimal numbers. For example, class F has major level 2 and minor level 1, leading to level 2.1. Also visible in Figure 4 are the dependencies that are broken to obtain the final order,

thus leading to five classes to be stubbed (B, C, D, F, H) and five *specific* stubs; that is, one for each class. The table on the right in Figure 4 shows the final test order.

Figure 4 Major and Minor Level Numbers³

In Tai & Daniels, associations that cross major levels in an ascending direction⁴ (source and target classes are in major levels i and j respectively, such that $i < j$) are systematically broken, thus leading to stubs. Other associations that cross major levels (in a descending direction) do not lead to stubs, because the target class is tested before the source class (descending major levels). In Figure 4, four associations are broken as they cross major levels in an ascending direction (e.g., association (E, F)). However, since association (E, F) is deleted and D and F are no longer involved in any cycles, stubbing D is not necessary and is just an artifact of Tai & Daniels' algorithm. Through this example we see how their algorithm can lead to sub-optimal solutions when there are associations, which *cross major levels* that are *not* involved in dependency cycles⁵. There are two distinct cases: the association was never involved in any cycle or, because other

³ Stub(C, A) denotes any stub of C fulfilling the needs for testing A.

⁴ Though in Figure 4, from a purely graphical point of view, they appear in a descending direction.

⁵ Note that this problem could be solved by identifying associations that are not part of any cycle before the application of the approach, and by considering these associations as aggregations; i.e., they are not part of the set of associations that are removed during the first step.

associations are deleted, the cycles they were originally involved in are already broken -- the latter case being the situation we encounter here.

3.2.2 Le Traon et al.

Le Traon et al. propose a graph-based approach [23]. This approach identifies SCCs using an adaptation of Tarjan's algorithm [39]. The algorithm is adapted such that each dependency edge in the considered graph is labeled, as per the order of traversal, according to a classification scheme. One type of edge, which is used in the decision to break cycles, is denoted as a *frond* dependency. A frond dependency is defined as going from a vertex (class) to one of its "ancestors", i.e. a vertex that is traversed before it in a depth-first search or, in other words, a class that depends on it, directly or indirectly. Le Traon et al breaks the cycles by removing the incoming dependencies of the class with the highest *weight*, in the considered SCC. The weight here is defined differently from that of Tai & Daniels: it is the sum of incoming and outgoing *frond* dependencies for a given class, within the SCC under consideration. In short, the notion of weight is defined by class and it specifically focuses on the notion of frond dependencies (which capture some of the cycles in which the class is involved). For each non-trivial SCC (with more than one vertex), the procedure above is then called recursively.

We apply Le Traon et al to the above example. Since Le Traon et al compute the weight according to frond dependencies, and frond dependencies depend on the construction of the depth-first search (DFS) tree, therefore the weight depends on the vertex from which we start the DFS algorithm. In this section we choose vertex G.

While applying Tarjan's Algorithm to the example, using G as a starting vertex, we find one SCC made up of classes F, E, C, A, D, B, and H (Figure 5.a). The (partial) test order is:

- 1 SCC {E, A, C, F, D, B, H} is tested;
- 2 G is tested using F and B.

Figure 5 Le Traon's approach for Figure 2, starting with node G (intermediate results)

The weight values of each vertex in the SCC are: $W(E) = 1$, $W(A) = 1$, $W(C) = 2$, $W(B) = 2$, $W(H) = 2$, $W(D) = 1$, $W(F) = 1$. Three vertices maximize the weight, leading to three choices for the set of broken dependencies. We show just one of the three choices here. B is selected and its incoming dependencies in the SCC are broken, i.e., the association between H and B. Tarjan's Algorithm is applied and identifies a lower SCC, involving classes E, A, C, F, D, and H (Figure 5.b). The (partial) test order is:

- 1 SCC {E, A, C, F, D, B, H} is tested;
 - 1.1 SCC {E, A, C, F, D, H} is tested using stub(B, H) ;
 - 1.2 B is tested using D, C and H;
- 2 G is tested using F and B.

We do not give further details regarding the application of the algorithm in the example, as the above description is sufficient for our purpose. Continuing the process we will see that when C and F are selected in the subsequent steps we obtain Figure 6, which shows the identified SCCs and the corresponding broken dependencies, as well as the final test order. This solution leads to three stubbed classes (classes C, B, and F) and four specific stubs (two for class C, one for class B, and one for class F), and breaks the aggregation between H and C.

(a) First call to Tarjan's algorithm (b) Second call to Tarjan's algorithm

Figure 6 Le Traon's approach for Figure 2, starting with node G

From Figure 6 we see what the frond dependencies are (according to the traversal), what nested SCCs have been detected using Tarjan's algorithm recursively, and what dependencies are deleted. The number of classes stubbed and the number of specific stubs are three and four, respectively. The results are therefore better than those obtained in Tai & Daniels (five classes stubbed and specific stubs). However, we can see that an aggregation is deleted (between H and C), which goes against our basic principle of exclusively breaking association dependencies (Section 3.1.2).

The algorithm proposed by Le Traon et al is not deterministic in the sense that its output depends on a number of arbitrary decisions. There are two levels of non-determinism. First, the result depends on the initial class from which one begins the depth-first-search, since in Tarjan’s algorithm, whether dependencies are classified as fronds or not depends on how the graph is traversed, and the weight considers only frond dependencies. Second, the algorithm does not specify what should be done when classes show the same weight. In other words, depending on the graph traversal, one can obtain a different set and/or number of specific stubs and classes to be stubbed. For example, we could have selected H instead of G as the initial class for the depth-first search. Three choices would then have been possible for the next class to traverse: A, E, F have the same weight. The results would then range from six to seven specific stubs (instead of four above), with a constant of three classes to stub (see Table 1).

3.3 A new approach

The new approach proposed in this thesis resembles Le Traon et al’s in that it is based on a recursive call to Tarjan’s algorithm. However, it also exhibits one important difference from Le Traon et al. Similar to Tai & Daniels the new approach uses a weight definition that characterizes associations by computing an estimate of the number of cycles in which the association is involved in an SCC.

As for Le Traon et al’s approach, the SCCs are recursively identified using Tarjan’s algorithm. At each step, i.e., inside each non-trivial SCC, we calculate the weight of each association dependency using a *modified* version of Tai & Daniels’ definition, and then break the *association* dependency with the highest weight. Note that, as opposed to Le

Traon et al's approach, our strategy does not have the first level of non-determinism, since the algorithm for computing SCCs has no effect on the weight. We do not use any dependency classification, such as fronds. Moreover, regarding the second level of non-determinism (alternative choices when for equals weights), as opposed to Le Traon et al's definition of weight, all alternative choices will lead to the same number of specific stubs. This is due to the fact that our criterion leads to the removal of one and only one association, whereas the criterion used by Le Traon et al leads to the removal of every incoming dependency of the selected class. The algorithms that implement the new approach are provided in Appendix A.

3.3.1 Weight computation

Recall from Section 3.2.1 that the weight associated with each association, using Tai & Daniels' definition, is the number of incoming edges of the source class *plus* the number of outgoing edges of the target class. Though we expect this weight to be positively related to the number of cycles (i.e., this is a reasonable heuristic), it is not an approximation of the number of cycles in which an edge is involved (i.e., it is not directly related to the number of paths in which an edge is involved). In our case we choose to multiply the number of incoming and outgoing edges within the SCC under consideration. What we obtain is an estimate of the *minimum* number of cycle(s) in which an edge is involved, within an SCC⁶. As illustrated by the SCC in Figure 7, the weight for edge e is $W(e) = 1 * 1 = 1$, though e participates in 2 cycles. We obtain an estimate of the

⁶ Note that we could compute the actual number of cycles in which an edge is involved, but this increases significantly the time complexity of the algorithm. This is a trade-off that should be explored in any specific situation.

minimum as, when computing the weight for an edge, we only have a partial view of the situation (e.g., we do not account for the classes C, D, E and F in Figure 7).

Figure 7 Example SCC

Ideally, we would like to delete first the associations that are involved in the largest number of cycles in order to minimize the number of stubs. Since we do not have such a count, we use as a heuristic the minimum estimated number, as per our weight definition, to select the next association to be broken in the SCC. The implications of our weight definition are as follows:

- Because our selection unit is the *association*, as opposed to the *class* in Le Traon et al, we can select the smaller number of associations, one by one, that breaks the largest number of cycles and really minimizes the number of *specific* stubs. On the other hand, Le Traon et al will minimize the number of classes to be stubbed, though aggregation and inheritance dependencies may be broken as a result.
- Because we use the *minimum* number of cycles in which an association is involved, our heuristic is more precise than the definition of Tai & Daniels, which does not clearly relate to the actual number of cycles in which the association is involved.

Note that our approach and Le Traon et al. are both based on a recursive identification of SCCs using the same algorithm, which is linear in the number of classes in the ORD [23]. Since this identification of SCCs is by far the most expensive part in of both approaches, these two approaches have the same overall time complexity (which is not linear).

3.3.2 Application of the new approach to the example

We now apply our algorithm to the example ORD of Figure 2. First, we obtain the result of the first call to Tarjan's algorithm that identifies first level SCCs in the ORD. Figure 8.a shows one non-trivial SCC: {F, E, C, A, D, B, H}. This implies the following (partial) test order:

- 1 SCC{F, E, C, A, D, B, H} is tested;
- 2 G is then tested using F and B.

Figure 8 Applying the new approach to Figure 2 (intermediate results)

To continue the execution of our algorithm, we calculate each association's weight within the SCC {F, E, C, A, D, B, H}:

$$\begin{array}{lll}
W(H,B)=H_{in}*B_{out}=3*3=9 & W(B,D)=B_{in}*D_{out}=1*2=2 & W(B,C)=B_{in}*C_{out}=1*3=3 \\
W(A,C)=A_{in}*C_{out}=3*3=9 & W(C,A)=C_{in}*A_{out}=3*1=3 & W(C,E)=C_{in}*E_{out}=3*2=6 \\
W(E,A)=E_{in}*A_{out}=2*1=2 & W(E,F)=E_{in}*F_{out}=2*2=4 & W(F,D)=F_{in}*D_{out}=1*2=2 \\
W(C,H)=C_{in}*H_{out}=3*2=6 & &
\end{array}$$

Associations (A, C) and (H, B) have the same maximal weight value. Suppose we choose (A, C)⁷ to be deleted and apply Tarjan's algorithm on SCC{F, E, C, A, D, B, H} (see Figure 8.b). Now:

- 1 SCC{F, E, C, A, D, B, H} is tested;
 - 1.1 A is tested using stub(C, A);
 - 1.2 SCC{F, E, C, D, B, H} is tested using A (for classes E, C, and D);
- 2 G is tested using F and B in the previous SCC.

The weight of every association in SCC {F, E, C, D, B, H} is determined as follows:

$$\begin{array}{lll}
W(H, B) = 3*3 = 9 & W(B, D) = 1*1 = 1 & W(B, C) = 1*2 = 2 \\
W(C, H) = 2*2 = 4 & W(C, E) = 2*1 = 2 & W(E, F) = 2*2 = 4 \\
W(F, D) = 1*1 = 1 & &
\end{array}$$

Again, we do not further detail the application of the algorithm to the example. The remaining steps involve breaking associations (H, B), (E, F), and (C, H) in that order, thus leading to Figure 9, which shows all the SCCs that have been identified during the process and the corresponding broken associations, as well as the final test order. Four classes (B, C, F, H) need to be stubbed and four associations are broken, leading to four specific stubs. We therefore obtain a number of specific stubs that is comparable or better, depending on the traversal selected, to the results obtained by Le Traon et al. (see

⁷ The number of specific stubs generated by choosing E(H, B) to be deleted first is the same as choosing E(A, C).

Table 1). This was one of our two main objectives, along with not breaking aggregation or inheritance dependencies.

Figure 9 Applying the new approach to Figure 2

3.4 Discussion of the above three approaches

Table 1 summarizes the results we obtained (from the example in Figure 2) with all three strategies to test ordering with cycles. As expected because of the weight definition they use, the most salient results are that Le Traon et al show a lower or equal number of classes stubbed (three versus four and five, for our new strategy and Tai & Daniels, respectively). Because of the way we define weights, which specifically attempts to minimize the number of broken associations, we obtain a number of *specific* stubs (four) that is consistently lower than the number produced by le Traon et al (four, six, and seven) and Tai & Daniels (five). Because the latter technique produces more stubs, either in terms of classes to be stubbed or specific stubs, it should probably not be used. Problems are encountered when associations are cut when not participating in cycles, just because they cross major levels. This explains, in our example, why five associations are

deleted versus four with the new strategy. Its main advantage lies in its simplicity, but the other two algorithms, though more complex, can also be supported by tools.

As described in the previous sections, the two existing approaches have different objectives, and we have argued that the new approach considers a better measure of integration cost: the new approach minimizes the number of specific stubs, whereas Le Traon et al's strategy minimizes the number of classes stubbed. However, the two approaches are both based on a recursive identification of SCCs using the same algorithm, which is linear in the number of classes in the ORD [39]. Since this identification of SCCs is by far the most expensive part of both approaches, they have the same overall time complexity (which is not linear).

In addition, Le Traon et al may lead to unacceptable cases where aggregation or inheritance dependencies are broken, thus leading to an even higher integration cost. Lastly, a weight that characterizes associations is flexible and allows for numerous extensions, such as forbidding the deletion of associations leading to control or state-dependent classes (that are hard to stub). These last points are, however, the subject of further research.

Another issue is that both our and Le Traon et al's strategies do not specify what to do when two or more classes/dependencies have the same weight. We have seen in the example above (see also Table 1) that, when using Le Traon et al's strategy, choosing one successor class to traverse instead of the other would not only change the integration order but also the resulting number of specific stubs. For example, when the initial class is H in our example, we can then choose from three classes of equal weight (A, E, F) to

traverse. Depending on which one we choose, we obtain either six or seven specific stubs. With our strategy, we observe that the integration order changes but the number of specific stubs remains the same, as one association is broken in all cases, although this is not a guarantee. In the next section, the above observations are confirmed by experimental results on five application systems.

	Tai & Daniels	Le Traon et al				New Approach
		Traversal starting with H			Traversal starting with G	
		Choose vertex A	Choose vertex E	Choose vertex F		
Classes stubbed	C, F, H, D, B	H, A, F	H, E, A	H, F, A	B, C, F	C, F, H, B
# classes stubbed	5	3	3	3	3	4
Specific stub	C, F, H, D, B	3H, 3A, 1F	3H, 2E, 1A	3H, 1F, 2A	1B, 2C, 1F	1C, 1F, 1H, 1B
# specific stub	5	7	6	6	4	4

Table 1 Summary of the results obtained from the application of the three approaches

3.5 Case Studies

We present here a brief version of the case studies. The detailed information can be found in [9].

To evaluate the new ordering approach, we apply the above three graph-based techniques to five application systems. Table 2 summarizes the detailed information for these five systems. For each system, the number of classes and different dependencies, i.e., usage, association/aggregation, composition, inheritance, and the number of cycles⁸ in the system, as well as the line of code, are recorded in the table. These five application systems⁹ were chosen because they were deemed to be of sufficient size and of varying

⁸ We count here the number of elementary circuits.

⁹ We made a conscious effort not to use libraries but application systems, in order to use case studies representative of the type of systems on which the techniques would typically be used and so as to avoid the peculiar class diagram topologies encountered in libraries (e.g., in [15], 4 of the 6 systems used are libraries and, for example, the Java Library shows 8000 cycles that are broken using 7 stubs).

complexity in terms of the number of classes and the number of cycles, so as to allow us to assess the effectiveness of the three graph-based approaches.

System	Classes	Usages	As and Ag	Compositions	Inheritance	Cycles	# Lines of code
ATM	21	39	9	15	4	30	1390
Ant	25	54	16	2	11	654	4093
SPM	19	24	34	10	4	1178	1198
BCEL	45	18	226	4	46	416, 091	3033
DNS	61	211	23	12	30	16	6710

Table 2 Summary of the information for the five case studies

Table 3 and Table 4 present the comparison of the three techniques. Note that this comparison not only looks at the number of stubs required when applying each approach but also provides the stubbing complexity of the produced test orders in terms of attributes and methods that may potentially be involved in the stubs. It is also worth noting that since these strategies are not deterministic, we executed each algorithm 100 times on each system. For each of the five case studies, executing the three algorithms 100 times each required about one minute on a typical personal computer (500MHz, 128 MB). The comparison presented in Table 3 and Table 4 uses the averages of the distribution of the number of stubs, and the stubbing complexity, respectively (the distributions can be found in [9]).

Since the objective of all the three approaches is to break cycles while minimizing the number of the stubs, we present in Table 3 the comparison of the number of stubs produced by the different strategies.

	New Approach	Le Traon et al.	Tai & Daniels
ATM	7	9.31	8
Ant	11	19.68	28
SPM	17	25.51	20.36
BCEL	70	67.38	128
DNS	6	10.21	27

Table 3 Number of stubs produced by the three strategies (mean)

Highlighted cells indicate the minimum number of stubs produced. The new approach produces the fewest stubs except for BCEL. In this particular case, Le Traon et al produces fewer stubs than the new approach does, though it consistently, in 100 executions, breaks 22 inheritance relationships. These differences are confirmed in [9], where we provide the distributions of number of stubs produced by the 100 executions of the different algorithms and use statistical tests (in some cases, the ranges of the number of stubs generated by 100 executions of the three approaches do not even overlap).

The comparison of the three approaches in terms of stubbing complexity is presented in Table 4. The stubbing complexity measures we used here are *Attribute coupling* and *Method coupling*. More measures could have been used [7], but these two measures are enough to capture the two main types of coupling involved (method and attribute coupling), while avoiding complex reverse engineering or analyses of the UML diagrams. These two stubbing complexity measures are described below.

Attribute coupling is the number of attributes *locally* declared¹⁰ in the target class when references/pointers to instances of the target class appear in the argument list of some methods in the source class, as the type of their return value, in the list of attributes (data members) of the source class, or as local variables of methods. This complexity measure counts the (maximum) number of attributes that would have to be handled in the stub if the dependency were broken. In the case of inheritance, we count the number of attributes declared in the parent class.

¹⁰ We do not count inherited attributes (and methods) as this would lead to them being counted several times when the stubbing complexity of an order is measured. As a result, if the parent class of a stubbed class has been tested before in the order, we do not count inherited attributes and methods, and this is what we expect as they are already integrated and tested. On the other hand, if we need to stub both the target class and its parent class, we ensure that we account for inherited attributes and methods only once.

Method coupling is the number of methods (including constructors) *locally* declared¹⁰ in the target class and invoked by the source class methods (including constructors). This complexity measure counts the number of methods that would have to be emulated in the stub if the dependency were broken. In the case of inheritance, we count the number of methods declared in the parent class. Note that this is an approximation, as some of the methods can be overridden.

Table 4 records the mean and median values of both attribute and method costs for each system. Highlighted cells show the minima. For the same reason as explained previously for Table 3, the new approach produces the minimum complexity for all systems except BCEL.

	New Approach		Le Traon et al.		Tai & Daniels	
	Attribute	Method	Attribute	Method	Attribute	Method
ATM	54.16	16	94.87	25.57	66.8	21
	54	16	87	19	66	21
Ant	163.38	24.62	307	78.11	463.48	88.52
	157	26	314	88	463	89
SPM	148.36	26.87	265.94	64.18	216.74	39.46
	148	27	254	58	211	38
BCEL	117.12	72	92.38	325.9	219.48	182
	118	72	92	324	222	182
DNS	23.05	11	74.5	76.8	61	32
	22	11	84	74	61	32

Table 4 Attribute and method costs for orders produced by the three strategies (mean, median)

Overall, from Table 3 and Table 4, we can draw a number of conclusions regarding the results of our case study (see details in [9]):

- The new approach clearly outperforms, both in the statistical and practical sense, the two other graph-based techniques in minimizing stubbing effort. It does so with respect to the three criteria we have measured: stubs, attributes, and methods.

- Le Traon et al performs, in general, better than Tai & Daniels, but that is not consistently the case as it depends on which system and criterion we consider.
- Le Traon et al generates, in all cases, a distribution of stubs, thus showing the non-determinism of the algorithm in practice.
- Le Traon et al tends to break a significant number of inheritance and/or composition relationships.

The order of class integration is the first step in performing the integration testing. The next step, which consists in applying a strategy to the testing of the interactions between two classes following that order, is presented in the following chapter.

CHAPTER 4 CLASS INTEGRATION TESTING

The previous chapter discussed the derivation of the order in which pairs of classes undergo integration testing. Furthermore, we assume a class testing suite for the client class in each pair is available. Though the way this test suite was generated does not impact our class integration testing strategy, it is obvious that the more complete this test suite, the more effective integration testing as our integration test sequences will be a subset of the class testing sequences. The motivation at this stage is to reuse the client class test sequences to test the interaction(s) between this client and its server(s). This lowers the cost of integration testing and it is clear that at this stage, while testing class interactions, we usually cannot afford to perform as many tests as during class testing. Furthermore, fully testing the client class again would probably not be cost effective for testing its interaction with a given server class.

Having fulfilled the above pre-requisites, this chapter presents the strategies, for each pair of interacting classes, to derive the sequences of server methods that the client class method sequences trigger, and then fully exercise the interaction between each pair of interacting methods in an integration client test sequence. The strategy to derive the sequence of server methods triggered by the client method sequence is presented in Section 4.1, and Section 4.2 proposes the strategy to the testing of the interactions between each client and server method pair. Section 4.3 presents how these strategies have been automated by a prototype tool, the main feature of the tool, and the detailed

design and implementation aspects, are introduced as well. To illustrate the use of our strategies, two case studies are presented in Section 4.4.

4.1 The strategy for deriving the triggered server method sequence

Our motivation is to determine how the client class exercises, for each test sequence, the server class of interest. Recall from Chapter 2 that, to do so, we have to consider the control flow of client methods and must identify the objects on which calls to server methods are performed, as these two aspects derive the method server sequences actually triggered by client method sequences.

Our strategy first consists in building, for each client method, an annotated control flow graph that accounts for calls between client methods (Section 4.1.1). Paths in this control flow graph are used to derive the sequences of server methods that are triggered by the corresponding client method (Section 4.1.2). These sequences are then used to derive the server method sequences that are triggered by client method sequences (Section 4.1.3). The next steps are then to identify redundant server sequences triggered by different client sequences and reduce (or eliminate) this redundancy by removing some of the server sequences (Section 4.1.4). Last, we identify the client sequences that trigger the subset of remaining server sequences and retain them to test the interaction between classes (Section 4.1.5). Note that although a client class can interact with several server classes, we only consider one client/server pair in Sections 4.1.1 to 4.1.5. We then discuss the impact of client interactions with more than one server class in Section 4.1.6. The different steps of our strategy are illustrated using the examples in Figure 10, where class A is the client class and class B is the server class.

<pre> 1 public class A { 2 private B b1, b2; 3 private int dA1, dA2, dA3; 4 public A(){ 5 b1 = new B(); b2 = new B(); 6 dA1 = 0; dA2 = 0; dA3 = 0; 7 } 8 public int mA1(int i, B b){ 9 i++; 10 if (i > 0){ 11 dA1 = b1.mB1(i); 12 dA2 = b1.mB2(i); 13 } else { 14 dA1 = b2.mB2(dA1); 15 dA2 = b2.mB1(dA2); 16 } 17 dA3 = b.mB3(b1.mB3(dA1)); 18 return dA3; 19 } 20 public void mA2(B bb){ 21 dA2 = bb.mB1(dA1); 22 mA1(dA2, bb); 23 } 24 public int mA3(int i){ 25 B aB1 = new B(); 26 B aB2 = new B(); 27 int j = aB1.mB1(i); 28 int k = aB2.mB2(i); 29 if (j > k) 30 return aB1.mB3(j); 31 else 32 return aB2.mB2(j); 33 } 34 public int mA4(int i){ 35 B aB3 = new B(); 36 // Class C, not shown, has 37 // a static attribute VAR 38 // of type int. 39 C.VAR = 12 40 int j = b1.mB1(i); 41 int k = aB3.mB2(dA2); 42 if (j > k) 43 return aB3.mB2(j); 44 else 45 return b1.mB1(k); 46 } 47 public void mA5(){ 48 B aB = new B(); 49 dA2 = aB.mB2(2); 50 mA1(1, b2); 51 } </pre>	<pre> 51 public boolean mA6(){ 52 FileWriter out; 53 out = new FileWriter(new File("F.txt")); 54 if (dMA1 > 10) 55 out.write('T'); 56 else 57 out.write('F'); 58 out.close(); 59 b1.mB5(); 60 C.VAR = 0; 61 } 62 } </pre>
	<pre> 1 public class B { 2 private int dB1, dB2; 3 public B(){ dB1 = 1; } 4 public int mB1(int j){ 5 dB1 = dB1 - j; 6 j = (dB1 + dB2) * C.VAR; 7 return j; 8 } 9 public int mB2(int j){ 10 dB2 = dB2 - j; 11 if (dB2>0) { 12 dB1 -= j; 13 return dB1; 14 } else 15 dB1 += j; 16 for (...) 17 ... 18 return dB2; 19 } 20 public int mB3(int i){ 21 return (dB1 - dB2) * i; 22 } 23 public boolean mB4(){ 24 FileReader in; 25 in = new FileReader(new File("F.txt")); 26 char c; 27 c = in.read(); 28 in.close(); 29 if (c == 'T') 30 return true; 31 else 32 return false; 33 } 34 public void mB5(){ 35 ... 36 mB4(); 37 ... 38 } 39 } </pre>

Figure 10 Running example (source code)

4.1.1 Interprocedural Control Flow Graph for client methods

Each client method is associated with an annotated Control Flow Graph (CFG). The nodes in the CFG are blocks of consecutive statements (two specific nodes being the *entry* and *exit* nodes), and edges represent the flow of control between the blocks. Nodes are annotated with the sequence of calls to server methods that are performed in the corresponding blocks of consecutive statements. This is not sufficient in our context as we are interested in sequences of calls performed on individual server instances. We thus

have to uniquely identify the server instances, involved in a CFG, on which calls are performed. Symbolic names qualifying these calls are introduced here, and then calls can be described as (symbolic name, method name) pairs. The requirements in terms of source code reverse engineering, and in particular the way symbolic names are determined so as to uniquely identify the server instances involved, are described in Section 4.3, where we show that there exist techniques and tools that can be used for that purpose.

In addition, the entry and exit nodes indicate the objects—only those of the type of the server class we are interested in—that are passed to (and returned by) the method. Again, symbolic names are used and correspond to the symbolic names involved in the method annotated CFG. Figure 11.a shows the annotated CFG for method `mA1()` in class `A`: `mA1()` has one formal parameter of type `B`, the server class, named `b`. Note that we use attribute and parameter names instead of different symbolic names to improve the clarity of the examples by making the mapping between the code and ICFGs straightforward. This is, however, not possible on real code as object references are assigned to variables/attributes and changed during execution.

These symbolic names can be of different kinds. First, calls can be performed on parameters and local variables of the method for which we build the CFG. Calls can also be performed on attributes, provided that their class is the server class of interest: the attributes of the current object, but also any *accessible* attribute of objects other than the current one. This notion of *accessible* attribute is programming language dependent, and corresponds, in the case Java, to public or protected (when in the same package), either

static or not, attributes. Last, method calls can be performed on references returned by other calls.

Since client methods can call each other, thus resulting in more complicated sequences of server methods than those triggered by a single client method, we combine the control-flow of client methods. We then build an Interprocedural Control Flow Graph (ICFG) for each client method. This notion of ICFG has been introduced in [34] in the context of C programs, and was adapted in [14] to an object-oriented context (though for class testing instead of class integration testing). In an ICFG, specific nodes are created to connect the control flow graphs of methods that call each other. Call sites to other methods of the client class are split into *call* and *return* nodes. Call nodes are connected to the entry nodes of the methods they invoke, and exit nodes are connected to the return nodes of all call sites that invoke the method (i.e., if a method is called several times, its CFG appears only once in the ICFG). When their types are the server class, call and return nodes indicate the actual parameter (using symbolic names) used in the call, and the variable (using a symbolic name) to which the return value of the call is assigned. This is necessary to determine the sequences of server methods executed on the different objects of the server class involved in the interaction.

Figure 11.b shows the ICFG for method `mA2()` in class A. We can see that during the call to `mA1()` in `mA2()`, actual parameter `bb` is mapped to formal parameter `b` in method `mA1()`, implying that the last call to an instance of class B performed in `mA1()` is performed on parameter `bb`.

Figure 11 Examples of annotated CFG and ICFG from Figure 10

Note that client/server interactions can also consists of calls to static methods of the server class. Such calls, including calls to constructors, are reported in the ICFG, and are all qualified by a unique symbolic name: the server class name.

4.1.2 Server method sequences triggered by client methods

The annotated ICFG is used to derive the sequences of server methods that are triggered by the corresponding client method. For each client method, paths in the corresponding ICFG are then determined. Note that the number of paths in an ICFG can be infinite because of loops and recursive calls. In this situation, we assume that each loop (recursive call) is bypassed (if possible), taken only once, a representative or average number above 1, and a maximum number of times. Note that this may still produce unfeasible paths (the undecidable path sensitization problem [5, 6]). How unfeasible paths are identified is out of the scope of this thesis and we may resort, in practice, to getting the user to verify the feasibility of paths.

While determining paths in the ICFG, the mapping between actual parameters and formal parameters (symbolic names) is used to replace any occurrence of a formal parameter

name with the corresponding actual parameter name. For instance (Figure 11.b), any occurrence of `b` is replaced with `bb`. The result of these replacements is that the symbolic names in the server sequences no longer correspond to the formal parameter names of the methods involved in the ICFG other than the ones for which we have built the ICFG. Again, we have to ensure that different symbolic names really correspond to different server instances. These issues are discussed in Section 4.3.

From each path in a client method's ICFG we extract the sequence of calls to server methods, and sort out the sequences corresponding to the different symbolic names involved. These are the sequences of server methods triggered by a single client method. As an example, Table 5 shows the sequences of server methods triggered by method `mA2()` (see Figure 10 and Figure 11.b).

Server methods called by the two paths in <code>mA2()</code> 's ICFG	Corresponding server method sequences for each symbolic name (server instance)
(bb, mB1()), (b1, mB1()), (b1, mB2()), (b1, mB3()), (bb, mB3())	bb: mB1().mB3()
	b1: mB1().mB2().mB3()
(bb, mB1()), (b2, mB2()), (b2, mB1()), (b1, mB3()), (bb, mB3())	bb: mB1().mB3()
	b2: mB2().mB1()
	b1: mB3()

Table 5 Server method sequences triggered by method `mA2()` in Figure 10

4.1.3 Server method sequences triggered by client method sequences

Sequences of server methods triggered by sequences of client methods can now be determined by concatenating the server method sequences, identified in the previous step, based on client method sequences. We also have to ensure, as it has been done previously, that symbolic names uniquely identify server instances.

In particular, the user input is required to decide whether the parameters used in the sequence of client methods reference identical objects. Let us assume that client method sequence `mA1().mA2().mA5()` has been defined and run during the class testing of class A, and is considered for reuse during the integration testing of classes A and B. In this sequence, both methods `mA1()` and `mA5()` require a parameter of type B (Figure 10) and we need the tester's help to determine (e.g., based on the semantics of the two methods and purpose of the sequence) whether these two parameters are equal. This is an important practical issue as it determines which server method sequences are possible. However, since we expect class test sequences to be already defined and executed when starting class integration testing, this information is readily available in the test plans and drivers for the client class.

Using the example sequence above (the ICFG for method `mA5()` in class A as well as the server method sequences triggered by `mA5()` can be found in Appendix B), if we make the assumption that attributes `b1` and `b2`, and parameters `b` and `bb` are referencing distinct instances, then there is a maximum of two possible server method sequences, resulting from concatenating one possible server sequence in each of the methods `mA1()`, and `mA2()` and two possible server sequences in `mA5()` (Table 6: Hypothesis 1). If we make a different assumption, for example that `b2=b=bb`, in `mA1()` and `mA2()`, we now obtain eight longer server method sequences (Table 6: Hypothesis 2).

Hypothesis 1	Client method sequence	$mA1() . mA2() . mA5()$ $\swarrow \quad \downarrow \quad \searrow$		
	Server method sequences triggered by individual client methods on b2.	$mB2() . mB1()$	$mB2() . mB1()$	$mB2() . mB1() . mB3()$
				$mB3()$
Hypothesis 2	Client method sequence	$mA1() . mA2() . mA5()$ $\swarrow \quad \downarrow \quad \searrow$		
	Server method sequences triggered by individual client methods on b2.	$mB2() . mB1() . mB3()$	$mB1() . mB2() . mB1() . mB3()$	$mB2() . mB1() . mB3()$
		$mB3()$	$mB1() . mB3()$	$mB3()$

Table 6 Concatenations of Server Sequences on attribute b2 for Client Sequence $mA1() . mA2() . mA5()$

As a result, each sequence of client methods triggers zero or more sequences of server methods, these sequences being pairs of the form $(symbName, seqServ)$ where *symbName* is a symbolic name identifying the object on which the sequence of server methods *seqServ* is executed. It can also refer to the server class name when *seqServ* is a sequence of static methods (including constructors) of the server class.

4.1.4 Removing redundancy

There may exist some redundancy in the way different client method sequences trigger server method sequences. In other words, identical or similar server method sequences may be triggered by different client method sequences. Our objective here is to remove (or reduce) such a redundancy to avoid testing several times identical or similar interactions between a client and a server and make class integration testing more cost-effective.

A preliminary step is to ignore sequences of client methods that do not trigger any sequence of server methods, since we are only interested in the interaction between a client and a server class. Next, when the methods involved in client method sequence S_1

all appear in the same order (in the sequence) in client method sequence S_2 , then the server method sequences triggered by S_1 are included in the server method sequences triggered by S_2 . In such a situation we ignore the client method sequence (i.e., S_1), which is included in another client method sequence (i.e., S_2).

Other simplifications are possible. Two different client method sequences may trigger identical or overlapping server method sequences (i.e., there is a subsequence relationship between the triggered sequences), though on different symbolic names. Our goal is to have a systematic strategy to take advantage of those redundancies to decrease the number of client class test sequences to be executed. It would not be realistic to expect that all test cases executed during class testing would be re-run during class integration testing.

Recall that we are in a situation where we have a set of *client-server method sequences pairs* for each symbolic name, denoted as a tuple: $(cseq_i, sseq_j, name_k)$, where $cseq_i$ triggers $sseq_j$ on server instance $name_k$. Based on such information, we can define three criteria to remove redundant tuples from our test plan, which result in different testing effort (i.e., number of class test sequences that are re-run), and, as one can expect, in different fault detection capabilities. Those criteria are defined below and illustrated with our running example (Figure 10).

Criterion 1: Server method sequences (removal of all the redundant server method sequences)

For any two tuples $(cseq_i, sseq_j, name_k)$ and $(cseq_m, sseq_n, name_l)$, such that $sseq_j \subseteq sseq_n$, $(cseq_i, sseq_j, name_k)$ can be removed¹¹, unless one or more methods in $cseq_i$ do not appear in other tuples.

Criterion 2: Server method sequences and server instance origin (removal when redundant server method sequences and same instance kind)

For any two tuples $(cseq_i, sseq_j, name_k)$ and $(cseq_m, sseq_n, name_l)$, such that $sseq_j \subseteq sseq_n$ and both $name_k$ and $name_l$ have the same kind, e.g., an attribute (see Section 4.1.1), $(cseq_i, sseq_j, name_k)$ can be removed, unless one or more methods in $cseq_i$ do not appear in other tuples.

Criterion 3: Server method sequences and server instance (removal when redundant server method sequences and identical instance name)

For any two tuples $(cseq_i, sseq_j, name_k)$ and $(cseq_m, sseq_n, name_l)$, such that $sseq_j \subseteq sseq_n$ and $name_k = name_l$, $(cseq_i, sseq_j, name_k)$ can be removed, unless one or more methods in $cseq_i$ do not appear in other tuples.

Note that, for all three criteria, every client method that appears in the original client method sequences is present in at least one tuple, and is thus executed at least once. The rationale is that the interaction between the client and the server classes be exercised at least once by every client method.

¹¹ $s_1 \subseteq s_2$ means that s_1 is a subsequence of s_2 .

From these definitions, it is clear that criterion 3 subsumes criterion 2, as, according to the way symbolic names are derived (see the previous sections), when symbolic names involved in two different tuples are equal (criterion 3), they have the same kind (criterion 2). Also, since criterion 1 does not put any constraints on the symbolic names (as opposed to criteria 2 and 3), criteria 2 and 3 subsume criterion 1.

Table 7 lists the tuples (except those containing calls to static methods of the server class) for four of the methods of client class A in Figure 10, assuming parameters *b* and *bb* reference the same object. It also indicates the tuples for four client method sequences, assuming these sequences come from the class testing of class A.

Tuples for client methods	Client method sequences	Tuples for client method sequences for each server instance symbolic name
(mA1, mB1.mB2.mB3, b1) (mA1, mB2.mB1, b2) (mA1, mB3, b)	mA1.mA2	(mA1.mA2, mB1.mB2.mB3.mB1.mB2.mB3, b1) (mA1.mA2, mB2.mB1.mB2.mB1, b2) (mA1.mA2, mB3.mB1.mB3, b)
(mA2, mB1.mB2.mB3, b1) (mA2, mB2.mB1, b2) (mA2, mB1.mB3, b)	mA2.mA1	(mA2.mA1, mB1.mB2.mB3.mB1.mB2.mB3, b1) (mA2.mA1, mB2.mB1.mB2.mB1, b2) (mA2.mA1, mB1.mB3.mB3, b)
(mA3, mB1, aB1) (mA3, mB1.mB3, aB1) (mA3, mB2, aB2) (mA3, mB2.mB2, aB2)	mA3	(mA3, mB1, aB1) (mA3, mB1.mB3, aB1) (mA3, mB2, aB2) (mA3, mB2.mB2, aB2)
(mA4, mB1, b1) (mA4, mB1.mB1, b1) (mA4, mB2, aB3) (mA4, mB2.mB2, aB3)	mA4.mA1	(mA4.mA1, mB1.mB1.mB2.mB3, b1) (mA4.mA1, mB1.mB1.mB1.mB2.mB3, b1) (mA4.mA1, mB2, aB3) (mA4.mA1, mB2.mB2, aB3) (mA4.mA1, mB2.mB1, b2) (mA4.mA1, mB3, b)
	mA4.mA2	(mA4.mA2, mB1.mB1.mB2.mB3, b1) (mA4.mA2, mB1.mB1.mB1.mB2.mB3, b1) (mA4.mA2, mB2, aB3) (mA4.mA2, mB2.mB2, aB3) (mA4.mA2, mB1.mB3, b) (mA4.mA2, mB2.mB1, b2)

Table 7 Tuples for client methods and client method sequences from Figure 10

Table 8 indicates the sequences that are selected from Table 7 when using the three different redundancy criteria. Six sequences are selected according to redundancy

criterion 1. Criterion 2 would yield a larger subset of tuples, and criterion 3 would yield the largest tuple subset (the new tuples are underlined). With criterion 2, tuple $(mA4, mB1.mB1, b1)$ is added to the tuples produced by criterion 1.

Assuming that the call at line 30 in $mA3()$ is erroneously coded as $aB1.mB3(k)$, instead of $aB1.mB3(j)$, then this fault would be detected with criteria 2 and 3. This would not be the case with criterion 1 though, thus illustrating why criterion 2 strengthen criterion 1. Similarly, a fault in the call at line 42 in $mA4()$'s implementation would not be detected by the tuples produced according to criteria 1 and 2, but would be detected by the tuples produced according to criterion 3.

Redundancy criterion 1	Redundancy criterion 2
$(mA1.mA2, mB1.mB2.mB3.mB1.mB2.mB3, b1)$ $(mA1.mA2, mB2.mB1.mB2.mB1, b2)$ $(mA1.mA2, mB3.mB1.mB3, b)$ $(mA2.mA1, mB1.mB3.mB3, b)$ $(mA3, mB2.mB2, aB2)$ $(mA4.mA1, mB1.mB1.mB1.mB2.mB3, b1)$	$(mA1.mA2, mB1.mB2.mB3.mB1.mB2.mB3, b1)$ $(mA1.mA2, mB2.mB1.mB2.mB1, b2)$ $(mA1.mA2, mB3.mB1.mB3, b)$ $(mA2.mA1, mB1.mB3.mB3, b)$ <u>$(mA3, mB1.mB3, aB1)$</u> $(mA3, mB2.mB2, aB2)$ $(mA4.mA1, mB1.mB1.mB1.mB2.mB3, b1)$
Redundancy criterion 3	
$(mA1.mA2, mB1.mB2.mB3.mB1.mB2.mB3, b1)$ $(mA1.mA2, mB2.mB1.mB2.mB1, b2)$ $(mA1.mA2, mB3.mB1.mB3, b)$ $(mA2.mA1, mB1.mB3.mB3, b)$ $(mA3, mB1.mB3, aB1)$ $(mA3, mB2.mB2, aB2)$ $(mA4.mA1, mB1.mB1.mB1.mB2.mB3, b1)$ <u>$(mA4.mA1, mB2.mB2, aB3)$</u>	

Table 8 Applying the three redundancy criteria from Table 7

An additional simplification can be performed. Given two tuples generated from the same client method sequence $(cseq, sseq_j, name_1)$ and $(cseq, sseq_m, name_2)$, if $name_1$ and $name_2$ are accessed in the same control flow path in the $cseq$ methods' ICFG, then either of the two pairs can be removed. This simplification does not reduce the fault detection

effectiveness, and can be used independently from the selected redundancy criterion. It is required to examine each ICFG control flow path and decide which tuple is to be removed, a task that can be easily automated. Continuing with our working example, and assuming we were using criterion 1, tuple $(mA1.mA2, mB3.mB1.mB3, b)$ can be removed as it is executed in the same control flow path as $(mA1.mA2, mB1.mB2.mB3.mB1.mB2.mB3, b1)$, and only five tuples remain to be tested: $(mA1.mA2, mB1.mB2.mB3.mB1.mB2.mB3, b1)$, $(mA1.mA2, mB2.mB1.mB2.mB1, b2)$, $(mA2.mA1, mB1.mB3.mB3, b)$, $(mA3, mB2.mB2, aB2)$, and $(mA4, mB1.mB1.mB1.mB2.mB2, aB3)$. It is worth noting that although the server method sequence triggered by $mA3$ is included in the server method sequence triggered by $mA4$, since $mA3$ doesn't appear in any of other retained client method sequences, it is kept to ensure that every client method is present in at least one tuple.

4.1.5 Test sequences

At this stage, the test sequences to be re-run are the method sequences on the client class in the tuples we obtained in the previous simplification stages. Each one of these test sequences may then correspond to one or several test cases, depending on the strategy used to exercise the interactions between client and server methods (e.g., coupling information as discussed in Section 4.2). Going back to our example, assuming we used criterion 1 and removed a sequence based on the analysis of the ICFG's, we obtain the following client method sequences: $mA1.mA2$, $mA2.mA1$, $mA3$, and $mA4.mA1$, where one redundant client method sequence, i.e. $mA4.mA2$, has been removed. The reduction is also important when we consider the control flow paths, i.e., the tuples, involved in the

original set of sequences and in the selected sequences: from 22 (Table 7, right column) to five (tuples in Table 8 when using criterion 1, except $(mA1.mA2, mB3.mB1.mB3, b)$).

4.1.6 Client interactions with several server classes

In the previous sections we assumed client/server pairs of interacting classes. However, class interactions in object-oriented systems are not restricted to client/server pairs: A client class can use the services of different server classes at the same time. In such a situation the previous approach can be used for each client/server_i pair, where server_i is one of the server classes whose services are used by the client class. The union of all the tuples (cseq, sseq, name) thus derived for each pair provides the set of sequences that have to be executed in order to test the interactions between the client class and its server classes. There may exist some redundancy in the way different client method sequences trigger method sequences in the server classes.

For instance, a given client method sequence can trigger, in one of its control flow paths, method sequences on objects of two (or more) server classes. This results in two tuples of the form (cseq, s_iseq_j, name_k) and (cseq, s_iseq_m, name_n) where cseq is the client method sequence that triggers s_iseq_j (respectively s_iseq_m), that is sequence seq_j (respectively seq_m) on server class s_i (respectively s_i), on symbolic name name_k (respectively name_n) that is an instance of server class s_i (respectively s_i). In such a situation, executing the client method sequence ICFG's control flow path only once is sufficient to trigger the two server method sequences. Removing (or reducing) such a redundancy thus requires that the control flows actually triggered by tuples in client method sequences be compared. This will be the subject of future work.

Another situation involving more than one server class is worth mentioning. Because of inheritance, even if one client/server pair is considered, several classes may be involved when the server class is a parent class. Indeed, the interaction between the client and its server is supposed to be consistent when an instance of the server class is replaced by an instance of any one of its child classes [24]. In such a situation, the interaction between the client and all its potential servers must be tested. However, the tuples defined between the client and the server parent class can be reused. Furthermore, if the multiplicity of the client/server relationship is one to many (more than one instance of the server class might be used by one client instance), the tester has to decide (1) how many server instances have to be instantiated (e.g., 0, 1, a reasonable number above one and a maximum number), and (2) what are the classes from which these instances are created (e.g., one instance from each potential server class). Solutions to these issues, likely involving the user's input, will be investigated in future work.

4.2 Coupling-based testing of method interactions

In the previous sections, our goal was to fully exercise the interactions between two classes, by carefully selecting client class test sequences. In this section, we go one level deeper and propose a procedure to fully exercise the interaction between each pair of interacting methods in an integration client test sequence. The method presented below makes use of the notion of coupling path between two subroutines [16] and adapts it to an object-oriented context to ensure that paths between definitions and uses of parameters, attributes, and files are covered during testing in order to exercise the interactions between a caller method and a callee method. In other words, we are combining two

types of techniques to fully exercise the interface between two interacting classes. We provide a short overview of basic data-flow analysis definitions (Section 4.2.1), propose a coupling type classification (Section 4.2.2) and coupling-based integration criteria (Section 4.2.3). All the definitions are illustrated using our running example in Figure 10. Important issues are then discussed in Section 4.2.4.

4.2.1 Data-Flow analysis definitions

Let us start with basic data flow analysis definitions in a non-OO context. A *definition* is a location in a subroutine where a variable's value is stored into memory. A *use* is a location where a variable's value is accessed. A *computation use (C-use)* is a node where a variable is used in a computation, as a functional parameter or in an output statement. A *predicate use (P-use)* is a node where a variable is used in a conditional expression. A new type of use, namely, *indirect-use (I-use)*, which is defined in [32], is also applied in [16]. An *indirect-use* occurs when a variable has a *C-use* first to define another variable, and then the defined variable is later used in a conditional expression. A *call-site* is a node in a caller's control flow graph from which a callee is called. A *return-site* is a node from which some value is returned to the caller.

From Figure 10, line 11 in class A (i.e., method `mA1()`) is a call-site to method `mB1()` in class B. Lines 13 and 18 in class B are return-sites from method `mB2()`. In method `mB2()` (class B), lines 10 and 11 are a C-use and P-use for parameter `j` and attribute `dB2`, respectively, thus resulting in an I-use of `j` at line 11. Line 10 is also a definition for attribute `dB2`.

In [16] the authors provide a number of basic definitions regarding coupling that we adapt and refine to our context based on ICFGs. If there is a node in a caller/callee subroutine's ICFG¹² that contains a definition and there is at least one execution path between this definition and one of its uses in a node in this ICFG, we say this definition is a *coupling-def* as far as that caller-callee pair is concerned. Similarly, a *coupling-use* is a node in a caller/callee subroutine's ICFG that contains a use so that there is at least one execution path between a definition in a node of the caller subroutine's ICFG and this use.

A coupling path is a control flow path between two subroutines from a coupling-def corresponding to an actual parameter in a caller/callee subroutine to a coupling-use of the corresponding formal parameter in the callee/caller unit. For the *in* parameters, a coupling path is from the last definition in the caller before calling the callee (*last-def-before-call*) to the first use of the parameter in the callee (*first-use-in-callee*). For the *out* parameters, the coupling path is from the last definition of the parameter before returning control to the caller (*last-def-before-return*) to its first use in the caller after the completion of the call (*first-use-after-call*). For instance (Figure 10), line 9 in class A (method `mA1()`) is a last-def-before-call for `i`, a call being located at line 11 (call to method `mB1()` in class B), and line 5 in class B (method `mB1()`) is the first-use-in-callee for formal parameter `j`, corresponding to actual parameter `i` in method `mA1()`.

¹² In this ICFG, the call site to the callee subroutine is split into call and return nodes in the caller's CFG (see Section 4.1.1).

Offutt et al present four coupling based integration test coverage criteria¹³ between subroutines in [16]:

- ♦ *Call-coupling* requires that the test cases cover all call-sites of the callee method in the caller method. This is the weakest criterion.
- ♦ *All-coupling-defs* requires that, for each coupling-def of a variable in the caller, the test cases cover at least one coupling path to at least one reachable coupling-use.
- ♦ *All-coupling-uses* requires that, for each coupling-def of a variable in the caller, the test cases contains at least one coupling path to each reachable coupling-use.
- ♦ *All-coupling-paths* requires that the test cases covers all coupling paths from each coupling-def of a variable to all reachable coupling-uses.

4.2.2 Coupling classification

Given the above definitions, we present a modified coupling type classification, based on [16] and tailored to Object – Oriented programming languages.

Call Coupling

There is a *Call coupling* between m_A and m_B when m_A invokes m_B .

¹³ Note when these criteria involve control flow paths and there is a loop along these paths, then typical techniques for testing loops can be applied to make the number of paths finite.

Parameter coupling

There is a *Parameter coupling* between m_A and m_B when m_A invokes m_B , and m_B has (at least) one parameter (or return value).

Shared Attribute coupling

There is a *Shared Attribute coupling* between m_A and m_B when m_A invokes m_B and when they both reference one or several attribute of any class such that m_A and m_B update and read that attribute, respectively. We define *shared-def* and *shared-use* nodes as ICFG nodes where a shared attribute is defined and read, respectively. Such attributes can be static, public (though this is considered poor practice), or accessible through specific mechanisms such as Friend classes in C++. This type is an adaptation of *Shared data coupling* defined in [16].

External device coupling

There is an *External device coupling* between m_A and m_B when m_A invokes m_B and when they both access an external medium (e.g., file) such that m_A and m_B update and read that medium, respectively. We define *external-def* and *external-use* nodes as ICFG nodes where an external device is updated and read, respectively.

Note that following these definitions, we do not consider definition-uses when the definition and the use are both either in the client method or in the server method, as this should be the focus of the unit testing of the client class in the former case (only a stub of the server class is then required), or the unit testing of the server class in the latter case. In other words, we do not consider the definition of a reference whose type is the server class (e.g., the assignment of a new reference to a local variable) and its use in a

following call on that reference¹⁴: former case. We do not consider interactions through shared attributes¹⁵ when a definition is performed in a call to the server object (i.e., the method modifies an attribute of the server object) and the use is also in a call to the server object¹⁶ (i.e., the method uses the attribute modified in the previous method): latter case.

The next step is to define the notion of coupling path, for each coupling type. A coupling path is a path in the ICFG of the caller and callee methods that exercise a particular pair of coupling definition and use. As in [16], those paths will constitute the basic coverage elements that will be used to elaborate our testing strategies. We then define four different types of coupling paths, corresponding to the four previous coupling types.

Call coupling paths

For a given method invocation on an instance of the server class, a call coupling path begins with the corresponding *call-site* and ends with a *return-site*.

Parameter coupling paths

For each actual *in* parameter involved in an invocation, a parameter coupling path starts with a *last-def-before-call* (of the actual parameter), continues through a *call-site*, and ends with a *first-use-in-callee* (of the formal parameter corresponding to the actual parameter). For each actual *out* parameter (and return parameter), a coupling

¹⁴ Definition of local variable `aB` at line 47 in class A and its use through a call to `mB2 ( )` at line 48 (Figure 10).

¹⁵ This corresponds to the definition-use pairs of references mentioned in [26], where the definition is a call to a client method modifier (i.e., a method that modifies the state of the object on which it is called) and the use is a call to a client method inspector (i.e., a method that uses the state of the object on which it is called).

¹⁶ Definition of attribute `dB1` at line 5 in class B as a result of call to `mB1 ( )` at line 11 in class A, and its use at line 12 in class B as a result of call to `mB2 ( )` at line 12 in class A (Figure 10).

path starts with a *last-def-before-return*, continues through a *return-site*, and ends with a *first-use-after-call* (of the returned parameter value).

Shared Attribute coupling paths

For each attribute shared by a caller and callee method, the shared attribute coupling path begins with a *shared-def* in the caller/callee, continues through a *call-site/return-site*, and ends with a *shared-use* in the callee/caller.

External device coupling path

For each external device shared by a caller and callee method, the shared external device coupling path begins with a *external-def* in the caller/callee, continues through a *call-site/return-site*, and ends with a *external-use* in the callee/caller.

Terms *coupling-def* and *coupling-use* can then be defined as any one of the “-def” (last-def-before-call, last-def-before-return, shared-def, external-def) and “-use” (first-use-in-callee, first-use-after-call, shared-use, external-use) above, respectively.

Figure 10 allows us to illustrate these different coupling paths. Considering, in method `mA1()` (class A), the call to method `mB2()` in class B (line 12 in class A), there is a call coupling path starting at line 12 in class A (call-site) and ending at line 13 in class B (return-site). Other coupling paths for the same call includes the loop in line 16 and end at line 18 (return-site). A parameter coupling path involving actual parameter `i` during the call to method `mB2()` in method `mA1()` starts at line 9 (last-def-before-call), continues through line 12 (call-site), and ends at line 10 in class B (first-use-in-callee). Attribute `VAR` in class C is shared by methods `mA4()` and `mB1()` in classes A (lines 38 and 39) and B (line 6) respectively. The corresponding shared attribute coupling path starts at line 38 (shared-

def) and continues through line 39 (call-site) in class A, and ends at line 6 (shared-use) in class B. Two external device coupling paths appear in the example: external-def at lines 55 and 57 and call-site at line 59 in class A, call to `mB4()` at line 36 (in `mB5()`) and external-use at line 27 in class B.

4.2.3 Coupling-based integration criteria

Offutt et al. presented four coupling-based integration test criteria in [16]. We adapted them here to define strategies, in increasing cost order, for covering all coupling paths defined above¹³:

- ♦ *All-Call-Sites* requires that the test cases cover all call-sites in the caller.
- ♦ *All-coupling-defs* requires for each coupling-def that the test cases contain at least one coupling path to at least one reachable coupling-use.
- ♦ *All-coupling-uses* requires that for each coupling-def that the test cases contain at least one coupling path to each reachable coupling-use.
- ♦ *All-coupling-paths* requires that the test cases cover all coupling paths from the coupling-def to all reachable *coupling-uses*.

4.2.4 Issues

The above definitions allow us to exercise client/server interactions, in a systematic way, by means of coupling paths: there is a coupling-def in `mA()` (respectively `mB()`) and a coupling-use in `mB()` (respectively `mA()`), and methods `mA()` and `mB()` are in the same tuple (`mA()` and `mB()` being in the client and server sequences, respectively). However,

when `mA()` and `mB()` are in two different tuples, or one of the two methods is not in any of the tuples, the coupling cannot be exercised, as it is not part of the original class testing sequences. Figure 10 illustrates the former situation, assuming method `mA6()` is not in the sequences, reused from the class testing of class A, where methods `mA1()` to `mA5()` appear. In other words, one of the sequences reused from the class testing of class A contains only method `mA6()`. In such a case, `mA6()` contains a definition of shared attribute `VAR` (class C) and method `mB1()` in class B contains a use of that shared attribute, thus resulting in a potential coupling between `mA6()` and `mB1()`. But methods `mA6()` and `mB1()` are not together in any of the tuples and that coupling cannot be exercised by integration test cases. Such situations can happen when two client classes A and A' use the services of a same server class B. Methods in A and A' can interact (and exhibit coupling) through the state of B or external devices (e.g., files) it accesses.

In order to exercise these couplings, a first solution is to ask the user to explicitly add sequence(s) so that the generated tuple(s) contain(s) both methods. Another solution is to wait for a subsequent integration step in the integration plan in which class C, a client class of both A and A' in the example above, is integrated right after A and A'. Such an integration step would allow us to exercise couplings between the methods of A and A' that were not exercised during their own integration.

4.3 Automation

This section introduces how the new approach presented in Sections 4.1 and 4.2 has been implemented in a prototype tool. Section 4.3.1 presents the metamodels used to model the

required information for sequence derivation and coupling coverage identification, the implementation of the tool is introduced in Section 4.3.2.

4.3.1 The required information and the metamodels

We first summarize the requirements in terms of source code reverse engineering of the approach (Section 4.3.1.1): Recall that we need to derive ICFGs and annotate them with symbolic names, and that we have to identify coupling paths. We show that there exist techniques and tools that can be used for that purpose, and we do not further discuss this issue in this thesis, though future work will address the integration of these techniques/tools into our prototype tool. We then identify the information relevant to the two different steps of the approach: deriving tuples for client method sequences and using redundancy criteria (Section 4.3.1.2), and applying the different coupling-based criteria (Section 4.3.1.3). This information, provided by reverse-engineering techniques is modeled by means of UML class diagrams (or metamodels). These metamodels first help us define the requirements in terms of information we need to retrieve from reverse-engineering. In turn, the algorithms that implement the approach (they are provided in Appendices) can be expressed in terms of classes and relationships in these metamodels. As a consequence, they make use of the Object Constraint Language (OCL) [40]. OCL in this context helps understand the algorithms as, no additional data structure is required (the metamodel describes what the structure is), and the mapping between the algorithms and the metamodel is straightforward. Last, we limit our description of the prototype tool to these two metamodels (and the corresponding algorithms) as they form the core subsystems of the tool.

4.3.1.1 Reverse-engineering Java source code

One important aspect of the approach is the amount of information that needs to be reverse-engineered. First, ICFGs must be built for each method in the client class. There exist approaches to gather control-flow, local data-flow, and symbol table information for Java programs [15]. In addition, we need information on the (possibly different) server instances that are used in these ICFGs, so as to assign symbolic names. Here again some techniques exist and can be reused. For instance, [28, 29, 36] propose techniques, referred to as points-to analysis, to determine the set of objects whose addresses may be stored in reference variables (e.g., method local variables) and attributes. These techniques can be applied in our context to ICFGs, thus identifying objects used in ICFGs' paths. Once objects have been identified, choosing a unique symbolic name for each one is straightforward.

However, we have to ensure that, when applying these techniques, the symbolic names produced are different if and only if they represent two different server instances. Let us take two example situations that illustrate why the choice of symbolic names may be difficult. First, two different symbolic names may correspond to the same attribute name, parameter name, local variable name, or return value name, as indicated in the source code. For instance, two calls on the same attribute name but in two different statements in the same method (same CFG) require two different symbolic names when the attribute is assigned a new value in a statement between the two calls, as two different objects are then involved. Also, when a specific CFG is used several times in an ICFG (i.e., different calls to the corresponding method are performed by the method for which we build the

ICFG), symbolic names corresponding to local variables in that CFG must be renamed when we derive sequences of server methods as they (may) correspond to different objects. Also, a given attribute name may correspond to different symbolic names in different CFG, as it is not possible when we build the CFGs to know whether the different names represent the same object.

As for coupling paths, tools have been built to measure coupling coverage of test cases: e.g., the coverage of the coupling criteria defined in [16] can be determined with the tool described in [31] in the case of Java programs. Again, such a tool can be reused but needs to be adapted since our definitions of coupling differ from the ones in [16].

The requirements of our class integration testing approach in terms of reverse-engineered information do seem realistic, as there exist techniques and tools that can be reused or adapted. As a consequence, we do not further discuss this issue in this thesis, though future work will address the integration of these techniques/tools into our prototype tool.

4.3.1.2 Automating the derivation of method sequences

Figure 12 shows the metamodel for the derivation of tuples. The first step of the approach consists in reusing client method sequences from the class testing of the client class: a `TestRequirement` instance is created along with `clientSeqNum` client method sequences (qualifier `seqIndex` is used to access these `clientSeqNum` sequences). The reverse engineering step provides a set of original tuples for each client method in these sequences (role name `originalTuples`). Each tuple consists of a client method sequence (role name `clientSeq`), a server method sequence (role name `serverSeq`), and a symbolic name. A symbolic name can have four different kinds, as defined in Section

4.1.1, and symbolic names are uniquely identified by their name. Then, applying any of the three redundancy criteria produces a new `TestRequirement` instance that is associated with a set of tuples (aggregation between `TestRequirement` and `Tuple`).

Figure 12 Metamodel for tuples

Note the user inputs, e.g., deciding whether actual parameters in a client method sequence are equal or not, are not part of our descriptions in this section, as we focus on the class diagram that models tuples, and its use by redundancy criteria.

Appendix C provides examples of instantiations of this metamodel for our running example (Figure 10, Table 7 and Table 8), and the algorithms that implement the three redundancy criteria from instances of this metamodel can be found in Appendix D.

4.3.1.3 Automating the identification of coupling coverage

Figure 13 is the metamodel (class diagram) for coupling information between the caller and callee methods. A `Method` may have several `CallSite` (0 or more), i.e., a node in its control flow graph where a method of the class is invoked, and several `ReturnSite` where the callee returns control to the caller (at least one return site corresponding to the last closing curly bracket). Since a `CallSite` corresponds to a method call, it is

associated with one `Method`, called the callee. The data flow information used for coupling-based integration testing criteria concerns definitions and/or uses before/after call-sites (e.g., last definition before call), return-sites (e.g., last definition before return) and method declaration (e.g., first use in callee). This is modeled by means of class `Site` (parent class for `CallSite`, `ReturnSite` and `Method`) and its associations with class `UseDef` (role names `lastDefs` and `firstUses`). Each definition or use is for an `InteractingElement` which can be either a `SharedAttribute`, an `ExternalDevice`, a `FormalParam` or an `ActualParam` (each `ActualParam` being associated with the corresponding `FormalParam`). Note that the fact that a callee method returns a value to a caller is modeled by specific instances of classes `FormalParam` and `ActualParam` in the callee and caller methods respectively. Last class `Path` is used by the algorithms that compute the test requirements according to the different coupling-based criteria we defined in Section 4.2.3: e.g., a parameter coupling path starts with a last-def-before-call, continues through a call-site and ends at a first-use-in-callee.

It is worth noting that abstracting associations to definitions and uses from classes `CallSite`, `ReturnSite` and `Method` to class `Site`, which greatly simplifies the metamodel, implies that constraints exist between classes `CallSite`, `ReturnSite`, `Method` and classes `UseDef` and `InteractingElement` (and its subclasses). For instance, role name `lastDefs` does not make sense for any instance of class `Method`, since it refers to “last definitions before” and, we are only interested in the first uses (i.e., first use in callee) in the case of a method declaration. Also, it does not make sense for a `CallSite` to have definitions and uses of formal parameters since in such a case we consider the

actual parameters used in the call, and the corresponding formal parameters in the called method only.

Figure 13 Metamodel for coupling information

These constraints are modeled below using the Object Constraint Language [40]:

- Instances of class Method cannot have definitions (i.e., last definition before):

```
Method.allInstances->forall (m:Method|m.lastDefs->Empty)
```

- Instances of class ReturnSite cannot have uses (i.e., first use after):

```
ReturnSite.allInstances->forall (rs:ReturnSite|rs.firstUses->Empty)
```

- Instances of CallSite cannot have definitions or uses of formal parameters:

```
CallSite.allInstances->forall  
(cs:CallSite|cs.firstUses.interactingElement  
->select(ie:InteractingElement|ie.oclType = FormalParam)->Empty)
```

```
CallSite.allInstances->forall  
(cs:CallSite|cs.lastDefs.interactingElement  
->select(ie:InteractingElement|ie.oclType = FormalParam)->Empty)
```

- Instances of `ReturnSite` cannot have definitions or uses of actual parameters:

```
ReturnSite.allInstances->forAll
(rs:ReturnSite|rs.firstUses.interactingElement
->select(ie:InteractingElement|ie.oclType = ActualParam)->Empty)

ReturnSite.allInstances->forAll
(rs:ReturnSite|rs.lastDefs.interactingElement
->select(ie:InteractingElement|ie.oclType = ActualParam)->Empty)
```

Appendix E provides examples of object diagrams that are instantiations of this metamodel for different coupling in our running example (Figure 10), and the algorithms that derive coupling paths from instances of this metamodel, according to the different criteria we defined in Section 4.2.3 can be found in Appendix F.

4.3.2 The implementation of the tool

Class Interaction Information Analyzer (CIIA) is developed to aid in the analysis of the interactions between the client and server classes.

This section is organized as follows: Section 4.3.2.1 introduces the system requirements and the overview of CIIA. Section 4.3.2.2 presents the system analysis and design in terms of the use case model and the subsystem decomposition. Section 4.3.2.3 summarizes the current implementation. Section 4.3.2.4 describes how to use CIIA to retrieve the information for the class interaction.

4.3.2.1 Requirements for CIIA

According to the previous discussion (Section 4.3.1) on the automation of our approach, CIIA has two primary functions: class interaction analysis and method interaction analysis. Therefore, the requirements of CIIA are classified into two aspects:

Figure 14 Overview of CIIA

- For class interaction analysis, CIIA shall be able to:
 - Load the original tuples for each client method, and the test requirement, i.e. the client method sequence, for the client obtained from the client's unit testing as well.
 - Generate new tuples that consist of client method sequence, server methods sequence and a symbolic name.
 - Remove the redundant tuples according to the corresponding redundancy criterion.
- For method interaction analysis, CIIA shall be able to:
 - Load the coupling information between a caller method and a callee method.
 - Identify the coupling paths between the caller and the callee.

Figure 14 describes CIIA in terms of its main functionality, inputs and outputs. The inputs to CIIA include the original tuples for each client method, the test requirement for the client method, and the coupling information between the methods in the client (caller), the selected redundancy criterion, and the method in the server (callee).

The output of the tool depends on the user's choice. In the case of class interaction analysis, the tool generates all the tuples according to the test requirement of the client; in the case of redundancy removal, the tool removes the redundant tuples based on the redundancy criterion the user chooses; in the case of method interaction analysis, the tool generates the coupling paths between the caller and the callee.

4.3.2.2 System Analysis and Design for CIIA

According to Section 4.1 and Section 4.2, our approach to class integration testing includes exercising the interactions between two classes and exercising the interactions between two interacting methods, therefore CIIA is decomposed into three subsystems, as shown in Figure 15. The public classes in each subsystem and the associations between these classes are shown to illustrate the organization of the subsystem.

Figure 15 Subsystem decomposition for CIIA

The CIIAGUI subsystem is responsible for providing a graphic user interface for the user to access the system. The class `UserMenu`, which is extended from `JFrame`, is a core class

in this subsystem that offers a set of menu items. Once a menu item is selected, the corresponding control object is instantiated and this control object in turn triggers the corresponding methods to perform certain operation.

The `ClassInteraction` subsystem is responsible for loading the original tuples for each client method and the client's test requirement, generating the client-server method sequence pair, removing the redundant server method sequences triggered by the client method sequence according to the user-selected redundancy criterion. `ClassInteraction` includes the classes shown in Figure 12.

The `MethodInteraction` subsystem is responsible for loading the coupling information between a client method and a server method, and identifying the coupling path between these two methods. `MethodInteraction` contains the classes shown in Figure 13.

4.3.2.3 Implementation

CIIA is implemented by Java™ 2 Platform, Standard Edition, v1.3.1. The CIIA GUI is created with the Swing components, which are part of the Java™ Foundation Classes (JFC) and can be used with either JDK™ 1.1 or the Java™ 2 Platform.

4.3.2.4 Usage

This section gives a brief introduction for the usage of CIIA.

- **Class Interaction**

Figure 16 shows the Class Interaction module of CIIA. To analyze the interactions between two classes, i.e. the client and the server, the user needs to provide: (1) **Client**

Original Tuples, which is the location and the name of the file name records of the client's original tuples with respect to the server. It can be input either manually, from the keyboard, or by clicking the corresponding button to browse the hard disk directory. (2) **Client Test Requirement**, which is the location and the name of the file that records the client method sequence. It can be input in the same way as the previous input. (3) **Redundancy Criterion** choice, which can be no tuple removal or one of the three redundancy criteria. The four choices are mutually exclusive. Given the above inputs, the server method sequences triggered by the client method sequences can be generated and saved in a file by clicking **Start**. Pressing **Report** shows the resulting tuples.

Figure 16 GUI for Class Interaction module

- Method Interaction

Figure 17 GUI for Method Interaction module

Figure 17 shows the Method Interaction module of CIIA. To analyze the interactions between two methods, the user needs to provide **Method Coupling Information**, i.e., the

location and the name of the file name recording of the coupling information between a caller method and a callee method. After `Start` is clicked, all the coupling paths could be identified and saved into a file, which can be viewed by clicking `Report`.

4.3.2.5 Summary

With the aid of CIIA the interaction information between two classes/methods can be derived correctly, completely and efficiently. However, this tool could still be improved in some places. For instance, for class interaction analysis function, the tester could be allowed to make run-time decision as to whether the actual parameters in a client method sequence are equal. For the method interaction analysis function, currently CIIA identifies all the coupling paths between two interacting methods. However, it should be able to identify different set of coupling paths according to different testing criterion.

4.4 Case studies

We report here the use of our approach on two case studies implemented in Java: (1) a class scheduler and course planner for students (Jadvisor), (2) an implementation of a linked list. The case studies, and the class interactions of interest are detailed in Section 4.4.1. We did not exercise all the client-server pairs of classes in each case study but rather selected the most complex ones for investigation in this thesis. Section 4.4.2 reports on the application of our strategy to derive sequences to be tested. The mutation operators, defined specifically to focus on the interactions between modules, that we used in the case studies are then described in Section 4.4.3. Section 4.4.4 reports on the results of our approach and compares its cost (in terms of test cases) and effectiveness (in terms

of killed mutants) with a black-box technique, i.e., Category-Partition [33]: Recall that a subset of client method sequences defined during the class testing of the client class are reused for integration testing. We compare our approach (removing redundant sequences and using coupling path information to set input parameter values) with the use of Category-Partition to select input parameter values. In other words, we assess the benefit of using white-box information (i.e., coupling paths) to reduce the number of test cases and select appropriate input parameter values over simply considering the specification of the client methods. Note that no other integration testing methodology really compares to what we propose here, and we therefore cannot really compare our approach with any alternative that has been designed for the same purpose. Section 4.4.4.3 summarizes the conclusions that can be drawn from these case studies.

4.4.1 Description of the case studies

4.4.1.1 Jadvisor

Jadvisor is a class scheduler and course planner for students, written in Java (<http://jadvisor.sourceforge.net>). Figure 18 shows the four subsystems of Jadvisor, as well as the class diagram in subsystem planner. The client and server classes we selected for the case study, are `StudentPlan` and `StudentSemesterPlan` respectively. The corresponding Java source code can be found in Appendix G. The aggregation between `StudentPlan` and `StudentSemesterPlan` is implemented with a two-dimensional array, called `_plan`, with four lines and three columns, corresponding to years (students graduate in 4 years) and semesters (3 semesters per year) respectively.

Figure 18 Jadvisor subsystems and class diagram in subsystem planner

4.4.1.2 Linked List

The second case study is an implementation of a bi-directional linked list, involving client class `List` and server class `Node` (Figure 19): each `Node` instance has references to the previous and next `Node` instances in the list. An excerpt of the corresponding Java source code can be found in Appendix J. In this implementation, `Node` instances hold a data (an `int` value) and are uniquely identified by a key (`int` value). Note that a `List` instance is not responsible for ordering the `Node` instances it contains (e.g., in increasing order of the keys). Rather, this is the responsibility of clients of `List` instances, as methods `insertAfter` and `insertBefore` have a parameter of type `Node` that determines where that new `Node` instance (whose data value is also provided as a parameter) is to be inserted (e.g., after the node passed as a parameter in the case of `insertAfter`). The key attribute is uniquely set in the constructor of `Node` using static attribute `public_key`.

Figure 19 Class diagram for Linked List

4.4.2 Testing Sequences

Bashir & Goel’s class unit testing strategy [4] has been used to derive client method sequences for the two case studies. The technique first identifies slices where a *slice* is defined as a quantum of a class including only a single attribute and the set of methods that manipulate this attribute. Methods in slices are classified as *reporters* (return the value of an attribute), *transformers* (modify one or more attributes) and *others* (do not fall into the two previous categories). Then the rationale is to devise sequences for each slice: (1) for each other method, one sequence containing only the method is produced; (2) all possible permutations of the transformers in the slice are considered, such that each transformer appears only once in the sequence.

4.4.2.1 Advisor

Following Bashir & Goel’s strategy [4], we derived the following simple sequences: `add.remove`, `remove.add`, `contains`, and `satisfiesPrerequisites`. Note that we have omitted sequences derived for the test of the constructor and “get” methods (also called reporters in [4]) of class `StudentPlan`, as they do not interact with server class

StudentSemesterPlan. ICFGs for methods `add` and `remove` both contain two paths that exercise the interaction with server class `StudentSemesterPlan`: server method sequences triggered are `contains` and `contains.add`, and `contains` and `contains.remove` for client methods `add` and `remove`, respectively. ICFGs for methods `contains` and `satisfiesPrerequisites` have more than one path as those methods have nested loops¹⁷, and exercise the interaction in the same way: call(s) to method `contains` of class `StudentSemesterPlan` (Appendix G).

The actual server instances on which these calls are performed depend on the number of times each loop is executed: for instance, if the first loop in `contains` is executed once, and the second loop is executed twice, two instances of the server class are used, i.e., the server instance in the first line (index 0) and first column (index 0), and the server instance in the first line (index 0) and second column (index 1) of array `_plan`. We thus decided, in order to ease the presentation of the case study, to associate symbolic names of the form `_plan(i, j)` where `i` and `j` are the line and column of the server instance in array `_plan` of the objects involved in client method sequences.

Table 9 uses this convention for representing symbolic names, and shows the tuples generated according to our approach from the four client sequences above. Note that it has been decided that client methods `add` and `remove` will be executed with the same parameters: server instance `_plan(1, 2)`. Similarly, one of the symbolic names used in `satisfiesPrerequisites` is directly driven by its parameters, and has been chosen to correspond to server instance `_plan(1, 0)`. We also decided that each loop be taken once

¹⁷ For instance, verifying that a `StudentPlan` contains a particular course (method `contains`) amounts to looking at all the cells of array `_plan`: four lines and three columns.

and twice (we selected these numbers so as to limit the number of tuples). Last, when a given tuple is exercised in different paths in an ICFG, only one is shown in Table 9, so as to simplify the table. Table 9 shows that 14 tuples were produced.

Tuples	
1	add.remove, contains.add.contains.remove, _plan(1,2)
2	add.remove, contains.contains.remove, _plan(1,2)
3	remove.add, contains.remove.contains.add, _plan(1,2)
4	remove.add, contains.contains.add, _plan(1,2)
5	Contains, contains, _plan(0,0)
6	Contains, contains, _plan(0,1)
7	SatisfiesPrerequisites, contains, _plan(0,0)
8	SatisfiesPrerequisites, contains, _plan(0,1)
9	SatisfiesPrerequisites, contains, _plan(1,0)
10	SatisfiesPrerequisites, contains, _plan(1,1)
11	SatisfiesPrerequisites, contains.contains, _plan(0,0)
12	SatisfiesPrerequisites, contains.contains, _plan(0,1)
13	SatisfiesPrerequisites, contains.contains, _plan(1,0)
14	SatisfiesPrerequisites, contains.contains, _plan(1,1)

Table 9 Tuples for Jadvisor

Table 10 shows the result of applying the three redundancy criteria we defined previously on tuples in Table 9. Server sequences for tuples 5 to 10 are included in tuples 11 to 14 with the same symbolic names, and are thus removed according to criterion 3, except tuple 5 that has been selected to ensure there is at least one execution of client method contains. In addition to these removals, server sequences in tuples 11 to 14 are included in tuples 2 and 4 with symbolic names of the same kind, and are thus removed according to criterion 2, except tuple 11 that has been selected so has to ensure there is at least one execution of client method satisfiesPrerequisites. Criterion 1 provides the same results as criterion 2 since we do not have different kinds of symbolic names.

Tuples			Criterion 2	Criterion 1
Criterion 3	1	add.remove, contains.add.contains.remove, _plan(1,2)		
	2	add.remove, contains.contains.remove, _plan(1,2)		
	3	remove.add, contains.remove.contains.add, _plan(1,2)		
	4	remove.add, contains.contains.add, _plan(1,2)		
	5	contains, contains, _plan(0,0)		
	11	satisfiesPrerequisites, contains.contains, _plan(0,0)		
	12	satisfiesPrerequisites, contains.contains, _plan(0,1)		
	13	satisfiesPrerequisites, contains.contains, _plan(1,0)		
	14	satisfiesPrerequisites, contains.contains, _plan(1,1)		

Table 10 Applying redundancy criteria from Table 9 (Jadvisor)

We then identified the coupling paths for every client-method/server-method pair and used the All-Coupling-Path criterion to cover method interactions. The paths can be found in Appendix H.

4.4.2.2 Linked List

Following Bashir & Goel's class unit testing strategy [4], we derived the seven client method sequences in Table 11. As for the previous case study, we have omitted sequences derived for the test of the constructor and "get" methods of class List, as they do not interact with server class Node. CFGs for methods insertAfter, insertBefore and deleteNode are simple. The complexity of the corresponding ICFGs is due to the fact that these methods call contains, which mainly consists of a loop (contains traverses the list to find the Node instance which is passed as a parameter). We decided that the loop in each ICFG be bypassed (if possible), taken once and twice (we selected these numbers so as to limit the number of tuples). As a result methods insertAfter, insertBefore, deleteNode and contains produce 9, 10, 11 and 8 tuples respectively

(see Table 18 in Appendix K), and these tuples correspond to different settings: e.g., a tuples can be exercised only when the list contains at least two elements and the second element is the parameter passed to the method. Every possible combination of those tuples has to be considered when determining tuples for client method sequences. In order to reduce the number of possible combinations, we decided that methods in a client sequence be executed with the same `Node` parameter. The total number of tuples generated from client method sequence is then 89 (the second column in Table 11 indicates the number of possible combinations for each client method sequence). Those tuples for client method sequences can be found in Appendix K (Table 19).

	Client method sequences	Number of tuples produced
1	<code>insertAfter.insertBefore.deleteNode</code>	11
2	<code>insertAfter.deleteNode.insertBefore</code>	11
3	<code>insertBefore.insertAfter.deleteNode</code>	12
4	<code>insertBefore.deleteNode.insertAfter</code>	12
5	<code>deleteNode.insertAfter.insertBefore</code>	17
6	<code>deleteNode.insertBefore.insertAfter</code>	17
7	Contains	9

Table 11 Client method sequences for Linked List

When applying the three different redundancy criteria, we go from 89 tuples to 21, 21 and 17 for criteria 3, 2 and 1 respectively (see Table 20 in Appendix K for details). Then we identified the coupling paths for every client-method/server-method pair and used the All-Coupling-Path criterion to cover method interactions. The paths can be found in Appendix L.

4.4.3 Mutants

Our intent was to select mutation operators that focus on interface faults between modules so as to emulate, as realistically as possible, the situation where class testing has been performed and most remaining faults are interface faults to be found during integration. In [11] the authors propose a set of such operators, to be applied to both the caller and callee modules, that are specifically defined for the C language. However, the same concepts can be applied to other similar languages (e.g., C++ and Java), e.g., a mutation operator that modifies a global variable in a C program can be used in a Java program to modify a shared attribute. As usual, for a given system under study, all *applicable* mutation operators must be covered, and the distribution of mutants across operators is driven by the program characteristics (e.g., use of parameters in calls).

4.4.3.1 Jadvisor

Seven mutation operators were used. The first two are specific to the caller method: *FunCalDel* deletes the call to the callee method in the callsite¹⁸; *ArgRepReq* replaces each actual parameter by a compatible constant. The following five operators are specific to the callee method: *DirVarRepReq* replaces each use or definition of an interface variable (i.e., formal parameter or the shared attribute accessed by the callee method) by another variable or constant; *IndVarRepReq* is similar to *DirVarRepReq*, but applies to the local variable and constant used in the callee method; *IndVarIncDec* inserts a pre-decrement operator (--) or a pre-increment operator (++) at each reference to a local variable or constant used in the callee method; *IndVarAriNeg* inserts an arithmetic

¹⁸ If the callee method returns `void`, then nothing is required to implement the operator; otherwise, a constant is to be set and used to replace the returned value.

negation operator (-) before local variables and constants used in the callee method; *RetStaRep* replaces the expression used in a return statement by one of the expressions used in other return statements in the callee method.

Figure 20 Distribution of mutants across operators and methods (Jadvisor)

We generated 29 mutants according to these mutation operators (see the distribution of mutants across operators in Figure 20). These mutants were seeded in both client class *StudentPlan* (mutants numbered 11 to 29) and server class *StudentSemesterPlan* (mutants numbered 1 to 10). More precisely, methods *remove*, *add*, *contains*, and *satisfiesPrerequisites* in client class *StudentPlan*, and in methods *add*, *remove* and *contains* in server class *StudentSemesterPlan* (see the distribution of mutants across methods in Figure 20). These numbers, and the distribution of mutants among classes/methods were deemed sufficient considering the complexity of the different methods (see the source code in Appendix G). For example, class *StudentSemesterPlan* has an attribute of type `java.util.List` for storing *Course* instances, and methods *add* and *remove* on *StudentSemesterPlan* instances simply call methods *add* and *remove* on this attribute. There is then a small number of opportunities for the seeding of faults. Note that one of these mutants was found to be equivalent (mutant number 21 seeded in method *remove* in class *StudentPlan*), thus resulting in 28 non-equivalent mutants.

4.4.3.2 Linked List

Seven mutation operators were used for the Linked list. Three of those operators were also used for Jadvisor: *FunCallDel*, *ArgRepNeg*, *DirVarRepReq*. Four new operators were used (the difference between the two case studies is due to differences in the source code). The first one is specific to the caller method: *ArgAriNeg* inserts an arithmetic negation for an actual parameter. The following three operators are specific to the callee method: *DirVarReqGlo* replaces an interface variable (i.e., formal parameter or the shared attribute accessed by the callee method) with a global variable accessed by the callee; *DirVarIncDec* inserts/removes an increment/decrement operation for a use of an interface variable; *DirVarAriNeg* inserts an arithmetic negation for a use of an interface variable.

We generated 63 mutants according to these mutation operators (see the distribution of mutants across operators in Figure 21). These mutants were seeded in both client class `List` (38 mutants) and server class `Node` (25 mutants). More precisely, methods `insertBefore`, `insertAfter`, `deleteNode`, and `contains` in client class `List`, and in methods `addNodeAfter`, `removeNodeAfter`, `getKey`, `getNext`, `getPrevious` and `setPrevious` in server class `Node` (see the distribution of mutants across methods in Figure 21). These numbers, and the distribution of mutants among classes/methods were deemed sufficient considering the complexity of the different methods (see the source code in Appendix J).

Figure 21 Distribution of mutants across operators and methods (Linked List)

4.4.4 Results

4.4.4.1 Jadvisor

As stated above, we generated functional test cases following the Category-Partition approach [33], and produced 38 test cases (see Appendix I for details). Following our strategy, that is, combining tuples and coupling paths (test cases satisfy the all-coupling-paths criterion), we produced 18, 12 and 28 test cases for the redundancy criteria 3, 1 and 2, and when using no redundancy criterion (i.e., keeping all sequences), respectively (see Table 12). The number of test cases is twice the number of tuples because methods `contains` and `satisfyPrerequisites` return a Boolean value and both `true` and `false` values are tested when considering coupling paths.

The five test sets, i.e., Category-Partition, redundancy criterion 3, redundancy criteria 1 and 2, and no criterion, killed 23, 28, 25, and 28 non-equivalent mutants (out of 28) respectively, thus resulting in 82%, 100%, 89% and 100% mutation scores (percentage of non-equivalent mutants killed), respectively. Table 12 thus shows that the redundancy

criterion 3 has a better cost-effectiveness than criteria 1, 2, no criterion and Category-Partition (half the number of test cases for a slightly higher effectiveness).

	Testing technique (and criterion)			
	No criterion	Criteria 1 & 2	Criterion 3	Category Partition
Tuples	14	6	9	NA
Test cases	28	12	18	38
Non-equivalent mutants killed	28	25	28	23
Mutation score	100%	89%	100%	82%
Number of Live mutants	0	3	0	5
Live mutants		17, 28, 29		16, 26, 27, 28, 6

Table 12 Results for Jadvisor

The three mutants that are not killed by criteria 1 and 2 were seeded in client method `satisfiesPrerequisites`. After investigation, we found that two different paths in `satisfiesPrerequisites`'s ICFG correspond to tuple `(satisfiesPrerequisites, contains.contains, _plan(0,0))`, i.e., the tuple selected by criteria 1 and 2. In other words, executing server sequence `contains.contains` on symbolic name `_plan(0,0)` from client method `satisfiesPrerequisites` can be achieved in two different paths of `satisfiesPrerequisites`'s ICFG. Only one of these two paths was used when producing test cases from that tuple, thus resulting in not exercising the faulty statements in the other path.

Mutants remain alive when using Category-Partition because of the constraints that we defined among the choices in order to limit the number of resulting test cases [33]. For instance, as a result of our constraints among the choices, we verified prerequisites between courses in the same year, but did not verify them between courses in different years, thus resulting in not killing mutant 16 (seeded in method

satisfiesPrerequisites). However, though removing those constraints would allow us to kill more mutants (and thus reaching a similar effectiveness as redundancy criterion 3), the cost difference (e.g., measured in terms of number of test cases) between the criterion and category-partition would be much larger (i.e., we go from 38 to 138 test cases). Note that after investigation, it appears that the live mutants shown in Table 12 would have been killed by any class testing technique applied to client class `StudentPlan` and server class `StudentSemesterPlan`.

4.4.4.2 Linked List

As stated above, we generated functional test cases following the Category-Partition approach [33], and produced 38 test cases (see Appendix M for details). Following our strategy, that is, combining tuples and coupling paths (test cases satisfy the all-coupling-paths criterion), we produced 13, 13, 12 and 41 test cases for the redundancy criteria 3 and 2, 1 and when using no redundancy criterion (i.e., keeping all sequences), respectively (see Table 13). As opposed to the previous case study, the number of test cases is less than the number of tuples. This is due to the fact that several tuples and coupling paths are executed in the same control flow paths, and thus are exercised by the same test cases.

The five test sets, i.e., Category-Partition, redundancy criteria 3, 2 and 1 and no criterion, killed 63, 57 and 63 non-equivalent mutants (out of 63) respectively, thus resulting in 100%, 90% and 100% mutation scores (percentage of non-equivalent mutants killed), respectively (Table 13).

	Testing technique (and criterion)			
	No Criterion	Criterion 1	Criteria 2 & 3	Category Partition
Tuples	89	17	21	N/A
Test Cases	41	12	13	38
Non-equivalent mutants killed	63	57	57	63
Mutation Score	100%	90%	90%	100%
Number of Live Mutants	0	5	5	0
Live Mutants		11,12,13,22,36	11,12,13,22,36	

Table 13 Results for Linked List

As in the previous case study, the three redundancy criteria are less costly than Category-Partition (one third of the number of test cases). Note that, as opposed to Jadvisor, we did not use constraints between choices to reduce the number of test cases. However, the three redundancy criteria miss the same five mutants, resulting in a smaller effectiveness. These mutants were all seeded in one statement, i.e., a call to the constructor of class `Node` in method `insertBefore` of class `List`: The first actual parameter, which represents the data stored in the node, does not have the correct value (line 18: Figure 30 in Appendix J). The corresponding tuple, (`insertBefore`, `Node`, `Node`), was considered redundant with other similar tuples by the three redundancy criteria and thus removed: call to `Node`'s constructor is tested in another method of class `List` (i.e., method `insertAfter` as shown in Appendix K by Table 18 and Table 19). Another selection of redundant tuples would have resulted in test sets that achieve 100% mutation score. Such a situation was expected since, as mentioned in Section 4.1.4, reducing redundancy between tuples may result in not exercising control flow paths, and thus not detecting faults along those paths.

After investigation, it appears that these five live mutants would have been killed by any class testing technique applied to client class `List`, as long as the stub used to simulate the behaviour of class `Node` reports on the value of the parameters that are passed to calls. The question is then whether these five mutants should be counted as our strategy assumes that interacting classes have passed their local tests.

4.4.4.3 Results from the case studies

The two case studies suggest that the three redundancy criteria along with the most expensive coupling based criterion (i.e., the all-coupling-path criterion) produce less expensive test sets than Category-Partition: from half to one third of the number of test cases. Recall that we used Category-Partition as no other integration testing methodology really compares to what we propose here, and we therefore cannot really compare our approach with any alternative that has been designed for the same purpose.

The results of the three redundancy criteria in terms of effectiveness are mixed and vary across case studies: mutation scores in the first and second case studies are 100% and 90% (or 100% if we do not count the mutants that would have been killed by class testing techniques), respectively. However, the mutation score is high in both cases, especially since the five live mutants in the `LinkedList` case study were seeded in the same statement. The main reason for such a difference in mutation score is that when applying our strategy, some random choices have to be made during the selection of redundant tuples. Future work may include performing other case studies to better generalize these results.

CHAPTER 5 CONCLUSIONS

Recall that two problems regarding the class integration testing have been addressed: (1) Deciding the order in which classes should be integrated and class interfaces should be exercised; (2) Choosing test design technique(s) to test the interactions of these integrating classes, following the order derived previously.

For the issue of class integration order, we have proposed an improved graph-based strategy that integrates the fundamental principles of Le Traon et al. and the dependency weighting principles of Tai & Daniels. Moreover, we have performed a systematic analytical and empirical evaluation [9] of the three graph-based techniques used to generate class integration order. We showed that in cases where associations cross major levels and are not involved in cycles, the strategy by Tai & Daniels [38] leads to the unnecessary use of stubs. Regarding the strategy of Le Traon et al. [23], though it clearly optimizes the number of classes to be stubbed, it can lead in practice to the deletion of aggregation or inheritance dependencies, and the output of their algorithm depends on some arbitrary search choices that may produce very different results. The new strategy proposed in this thesis does not have the problems mentioned above, and performs better in terms of the number of stubs required, and the number of attributes or methods those stubs may need to emulate. These statements are demonstrated in analytical terms but also clearly supported by our experimentation on five application systems of non-trivial and varying complexity [9].

Future work will include the development of more precise measures to better estimate the cost of stubbing and the graph-based search for orders in the context of constraints (e.g., one may want to big-bang test classes in a strongly connected component instead of breaking the cycles it contains; one may want to consider that some parts of the class diagram are not available because not already implemented or unit tested). This will probably require the use of optimization techniques, as it has already been explored recently with genetic algorithms [8], and we believe this is a new direction of research that is worth further investigating and involves entirely new challenges.

With respect to the problem of class interaction testing, this thesis has defined a precise and automatable procedure to test class interactions in the context of class integration testing. This is complementary to the previous work on integration test orders and is based on existing works regarding class and integration testing [4, 16]. After deriving the class integration order, we test each pair of interacting client/server classes by analyzing the patterns of method executions from clients to servers and their inter-method control and data flows (referred to as coupling paths) when a method call takes place from the client to the server. Our rationale is to define coverage criteria that ensure that interactions between classes are exercised (i.e., all possible server method sequences are executed, coupling paths are covered) while minimizing the number of test cases to be re-executed on the client class during integration. We thus define four criteria and compare them with a black-box testing technique (category-partition) in setting the method parameter values, so as to have a baseline of comparison and assess the benefit of using control and data flow information to optimize integration testing. The case studies we present show that our strategy and criteria yield smaller test suites and higher fault

detection rates than those of category partition. This is particularly true for one case study and one of the criteria we define that produces half the test cases of category-partition while detecting all seeded class interface faults. Category-partition is a general-purpose functional testing technique and it should be expected that it does not perform as well as a technique that is specifically aimed at class integration testing. Our results are consistent with this expectation and thus show that the class integration test technique we propose is particularly efficient and far better than performing standard black-box testing in this context.

There are several limitations of our class integration testing approach that should be considered. An implementation limitation is that, as described in Section 4.3.1.1, the techniques/tools for reverse-engineering source code should be integrated into the prototype tool presented in the thesis. As discussed in Section 4.1.6, a methodological limitation is that our approach considers one client/server class pair each time. Since a client class may interact with several server classes at the same time, the approach should be adjusted to compare the control flows when removing the redundant tuples. Furthermore, if the server class is a super class, then the interactions between the client and all the subclasses of the server class should be tested. In addition, we may take the multiplicity of the client/server relationship into consideration to instantiate more than one object of the server classes during testing. The issue of handling the exceptions in CFG/ICFG is also worth further investigating.

Future work will address these limitations and includes running additional case studies to explore the above issues and the refinement of our integration test support environment.

References

- [1] R. T. Alexander and A. J. Offutt, "Criteria for testing polymorphic relationships," *Proc. 11th International Symposium on Software Reliability Engineering (ISSRE 2000)*, San Jose, CA, USA, pp. 15-23, 2000.
- [2] R. T. Alexander, A. J. Offutt and J. M. Bieman, "Fault Detection Capabilities of Coupling-based OO Testing," *Proc. 13th International Symposium on Software Reliability Engineering (ISSRE)*, Annapolis, Maryland, pp. 207-218, November, 2002.
- [3] G. Antoniol, L. C. Briand, M. Di Penta and Y. Labiche, "A Case Study Using the Round-Trip Strategy for State-Based Class Testing," *Proc. 13th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Annapolis (MD, USA), IEEE Computer Society, pp. 269-279, November 12-15, 2002.
- [4] I. Bashir and A. L. Goel, *Testing Object-Oriented Software - Life Cycle Solutions*, Springer-Verlag, 2000.
- [5] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, 2nd Ed., 1990.
- [6] R. V. Binder, *Testing Object-Oriented Systems - Models, Patterns, and Tools*, Addison-Wesley, 1999.
- [7] L. Briand, J. Daly and J. Wuest, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 25 (1), pp. 91-121, 1999.
- [8] L. Briand, J. Feng and Y. Labiche, "Experimenting with Genetic Algorithms and Coupling Measures to Devise Optimal Test Orders," in T. M. Khoshgoftaar, Ed., *Software Engineering with Computational Intelligence*, Kluwer, 2003.
- [9] L. C. Briand, Y. Labiche and Y. Wang, "An Investigation of Graph-Based Class Integration Test Order Strategies," *IEEE Transactions on Software Engineering*, vol. 29 (6), 2003.
- [10] F. J. Daniels and K.-C. Tai, "Measuring the Effectiveness of Method Test Sequences Derived from Sequencing Constraints," *Proc. Technology of Object-Oriented Languages and Systems (TOOLS)*, Santa Barbara, California, pp. 74-83, August, 1999.
- [11] M. E. Delamaro, J. C. Maldonado and A. P. Mathur, "Interface Mutation: An Approach for Integration Testing," *IEEE Transactions on Software Engineering*, vol. 27 (3), pp. 228-247, 2001.

- [12] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [13] M. J. Harrold, J. D. McGregor and K. J. Fitzpatrick, "Incremental Testing of Object-Oriented Class Structures," *Proc. 14th IEEE International Conference on Software Engineering (ICSE)*, Melbourne, Australia, pp. 68-80, May, 1992.
- [14] M. J. Harrold and G. Rothermel, "Performing Data Flow Testing on Classes," *Proc. 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 154-163, December, 1994.
- [15] M. J. Harrold and G. Rothermel, "Aristotle: A System for Research On and Development of Program-Analysis-Based Tools," The Ohio State University, Technical Report OSU-CISRC-3/97-TR17, March, 1997.
- [16] Z. Jin and A. J. Offutt, "Coupling-based Criteria for Integration Testing," *Software Testing, Verification and Reliability*, vol. 8 (3), pp. 133-154, 1998.
- [17] P. C. Jorgensen and C. Erickson, "Object-Oriented Integration Testing," *Communications of the ACM*, vol. 37 (9), pp. 30-38, 1994.
- [18] D. Jungnickel, *Graphs, Networks and Algorithms*, Springer, 1999.
- [19] D. Kung, J. Gao, P. Hsia, J. Lin and Y. Toyoshima, "Class Firewall, test order, and regression testing of object-oriented programs," *Journal of Object-Oriented Programming*, vol. 8 (2), pp. 51-65, 1995.
- [20] D. Kung, J. Gao, P. Hsia, Y. Toyoshima and C. Chen, "On Regression Testing of Object-Oriented Programs," *Journal of Systems Software*, vol. 32 (1), pp. 21-40, 1996.
- [21] Y. Labiche, P. Thévenod-Fosse, H. Waeselynck and M.-H. Durand, "Testing Levels for Object-Oriented Software," *Proc. 22nd IEEE International Conference on Software Engineering (ICSE)*, Limerick (Ireland), IEEE Computer Society, pp. 136-145, June, 2000.
- [22] V. Le Hanh, K. Akif, Y. Le Traon and J. M. Jezequel, "Selecting an efficient OO integration testing strategy: an experimental comparison of actual strategies," *Proc. 15th European Conference for Object-Oriented Programming (ECOOP)*, Budapest (Hungary), pp. 381-401, June, 2001.
- [23] Y. Le Traon, T. Jérón, J.-M. Jézéquel and P. Morel, "Efficient Object-Oriented Integration and Regression Testing," *IEEE Trans. on Reliability*, vol. 49 (1), pp. 12-25, 2000.
- [24] B. H. Liskov and J. M. Wing, "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16 (6), pp. 1811-1841, 1994.

- [25] L. A. Maciaszek, *Requirements Analysis And System Design - Developing Information Systems with UML*, Addison-Wesley, 2001.
- [26] V. Martena, A. Orso and M. Pezzè, "Interclass Testing of Object Oriented Software," *Proc. IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, Greenbelt, MD, 2002.
- [27] J. D. Mc Gregor and D. A. Sykes, *A Practical Guide to Testing Object-Oriented Software*, Addison-Wesley, 2001.
- [28] A. Milanova, A. Rountev and B. Ryder, "Parameterized object-sensitivity for points-to and side-effect analysis for Java," *Proc. ACM International Symposium on Software Testing and Analysis*, Roma, Italy, pp. 1-11, July, 2002.
- [29] A. Milanova, A. Rountev and B. G. Ryder, "Constructing Precise Object Relation Diagrams," *Proc. IEEE International Conference on Software Maintenance*, Montreal, Canada, pp. 586-595, October, 2002.
- [30] A. J. Offutt and A. Abdurazik, "Generating Tests from UML specifications," *Proc. 2nd International Conference on the Unified Modeling Language (UML'99)*, Fort Collins, CO, pp. 416-429, October, 1999.
- [31] A. J. Offutt, A. Abdurazik and R. T. Alexander, "An Analysis Tool for Coupling-based Integration Testing," *Proc. IEEE International Conference on Engineering of Complex Computer Systems*, Tokyo, Japan, pp. 172-178, September, 2000.
- [32] A. J. Offutt, M. J. Harrold and P. Kolte, "A software metric system for module coupling," *Journal of Systems and Software*, vol. 20 (3), pp. 295-308, Elsevier, 1993.
- [33] T. J. Ostrand and M. J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Test," *Communications of the ACM*, vol. 31 (6), pp. 676-686, 1988.
- [34] H. D. Pande, W. A. Landi and B. Ryder, "Interprocedural Def-Use Associations for C Systems with Single Level Pointers," *IEEE Transactions on Software Engineering*, vol. 20 (5), pp. 385-403, 1994.
- [35] D. E. Perry and G. E. Kaiser, "Adequate Testing and Object-Oriented Programming," *JOOP*, vol. 2 (5), pp. 13-19, 1990.
- [36] A. Rountev, A. Milanova and B. Ryder, "Points-to analysis for Java based on annotated constraints," *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 43-55, October, 2001.
- [37] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

- [38] K.-C. Tai and F. J. Daniels, "Interclass Test Order for Object-Oriented Software," *Journal of Object-Oriented Programming*, vol. 12 (4), pp. 18-25,35, 1999.
- [39] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, vol. 1 (2), pp. 146-160, 1972.
- [40] J. Warmer and A. Kleppe, *The Object Constraint Language*, Addison-Wesley, 1999.
- [41] E. J. Weyuker, "The evaluation of program-based software test data adequacy criteria," *Communications of the ACM*, vol. 31 (6), pp. 668-675, 1988.

Appendix A Algorithms for deriving the class integration order

The rationale of the algorithms is as follows: Given a cyclic Object Relation Diagram (ORD), first, the Strongly Connected Components (SCCs) are identified, then the original ORD is transformed to an acyclic digraph ORG'. A topological sorting is applied to the ORG', therefore, a first level order is generated for each SCC. For each SCC, we select the edge with the maximal weight and remove it to break the cycle in this SCC. The above procedures are applied recursively until all the cycles are broken and all the classes in the ORD are assigned a test order.¹⁹

Eight algorithms are provided in this appendix. The following are the primary algorithms: Procedure `classLevel` takes a cyclic ORD and assigns each class in the ORD a test order; Function `sccArray` applies Tarjan Algorithm to identify the SCCs in the ORD; Function `TaiArray` is used to assign a test level to each SCC; and the others are auxiliary algorithms.

A.1 Procedure `classLevel`

This algorithm is used to derive the test level of the class in an Object Relation Graph.

Input:

```
original_Graph: string[1..V,1..V]
//represents the original Object Relation Diagram. V is the
number of //classes. Each class is labeled by an integer from 1
to V. The element //of the array is maintained to record the
relation between the //classes, which is 'I', 'AS' or 'AG'.
operating_Graph: string[1..V, 1..V] //part of Original_Graph.
```

Input/Output: `level: string [1..V]` //records each class's test level.

¹⁹ This approach is used by Kung et al in [19].

Algorithm

```
procedure classLevel (original_Graph,operating_Graph,Level)

var:   SCCGraph: boolean[1..V,1..V]
        SCCLevel: int[1..V]
        SCCs: int[1..V,1..V]
        edge: int[2]
        newGraph: string[1..V,1..V]
        j,k,w,z,scc_number: int;

function:   getNumberofSCCs
              SCCArray
              Collapse
              TaiArray
              MaxWeightEdge
              deleteEdgeGraph

begin
  SCCs:=sccArray (operating_Graph);
  //Apply Tarjan's Algorithm to identify SCCs and record SCCs into an array.
  SCCGraph:=collapse (SCCs, Operating_Graph);
  //Converts each SCC to a corresponding node and collapses the edges between
  two //SCCs.
  SCCLevel:=TaiArray (SCCGraph);
  // Apply Tai's Major Level Algorithm to assign each SCC a level number.
  scc_number:=getNumberofSCCs(SCCs)
  for z:=1 to scc_number do
    begin
      w:=1;
      k:=SCCs[z,w];
      while k>0 do
        begin
          Level[k]:=Level[k] append ( '.' append ( SCCLevel[z]));
          w++;
          k:=SCCs[z,w];
        end;
      end;
    end;
    for j:=1 to scc_number do
      begin
        if SCCs[j, 2]!=0 then // SCCs[j,1..V] has more than one class.
          edge:=maxWeightEdge (operating_Graph, SCCs[j,1..V]);
          newGraph:=deleteEdgeGraph (Operating_Graph,SCCs[j,1..V],Edge);
          classLevel (original_Graph,newGraph,Level); //Recursive call
        endif;
      end;
    end.
```

A.2 Function sccArray

This function implements Tarjan's Algorithm to identify the SCCs in a cyclic graph and records the SCCs in the corresponding sccArray by the class label. An SCC may contain only one class.

Input: graph: int[1..V, 1..V]

Output: SCCs: int[1..V, 1..V]

//array records the labels of the classes in a SCC

Algorithm

```
function sccArray(graph: array[1..V,1..V]):
var: counter, i: int
    w1, w2: int //class label
    lowlink: int[1..V] //lowlink for each class
    number: int[1..V] of int
function: scc //Tarjan's Algorithm
begin
    i:=0;
    counter:=0;
    init_stack();
    for w1:=1 to V do
    begin
        lowlink[w1]:=0;           // Initializes the lowlink number of each class
        number[w1]:=0;           // Marks all classes in the graph as "unvisited"
    end;
    for w2:=1 to V do           // Starts to search SCC
    begin
        while number[w2]=0 and !(Graph [w2,1..V]=null and Graph [1..V,w2]=null) do
            // For each class which is unvisited and connected with other classes apply
            //Tarjan's algorithm.
            SCC(w2);
        end;
    end.
end.
```

A.3 Procedure scc

This is Tarjan's Algorithm.

Input: k: int // class label

Algorithm

```
procedure scc(k)
var: j, m: int
function: min //return the minimum number
    init_stack
    in_stack
    peek
    pop
    push
begin
    lowlink(k):=number(k):=++counter;
    push(k); //push class k into the stack
    for m:=1 to V do
    begin
```

```

if graph[k,m]!=null then // m is k's successor
  if number[m]=0 then // m is not visited, k->m is a tree arc
    scc(m);
    lowlink(k):=min( lowlink(k), lowlink(m));
  elseif number(m) < number(k) then // k->m is cross link or fronds
    if in_stack(m) then // m is in the stack
      lowlink(k) := min( lowlink(k), number(m))
    endif;
  endif;
endif;
end; // end for
end.

```

A.4 Function collapse

This function converts a SCC to a node and collapses the edges into one between the SCCs.

Input: SCCs: int[1..V, 1..V] // record of each SCC and the classes in each SCC
 Operating_Graph: string[1..V, 1..V] // string array representing a graph

Output: array[1..V, 1..V] of boolean
 //indicating whether the edges between two SCCs have been collapsed.

Algorithm

```

function collapse( SCCs, Operating_Graph):
  array[1..numberOfSCCs, 1..numberOfSCCs] of boolean
var: i,j,k1,k2,m,n,scc_number: int;
function:   getNumberOfSCCs
             getNumberOfClasses
begin
  scc_number:=getNumberOfSCCs(SCCs);
  // The number of edges in SCCs which values are not 0
  for i:=1 to scc_number do
    for j:=1 to scc_number do
      collapse[i,j]:=false;
    for i:=1 to scc_number do
begin
  for j:=1 to scc_number and i!=j do
begin
    if collapse[i,j] == false then
      class_num1:=getNumberOfClasses(SCCs[i,1..V]);
      class_num2:=getNumberOfClasses(SCCs[j,1..V]);
      for m:=1 to class_num1 do
begin
        k1:=SCCs[i,m];
        for n:=1 to class_num2 do
begin
          k2:=SCCs[j,n];
          if operating_Graph[k1,k2]!=null then

```

```

        collapse[i,j]:=true;
        m:=class_num1+1;
        n:=class_num2+1; // jumps out of the loops
    endif;
end;
end;
endif;
end;
end;
end.

```

A.5 Function TaiArray

This function applies topological sorting using Tai's Major Level Algorithm to assign each SCC a level number. Note that the weight is calculated by the product, instead of the sum proposed in [Tai+99], of the number of the incoming edges and the outgoing edges.

Input: SCCGraph: int[1..V, 1..V] // acyclic graph made up of SCCs

Output: int[1..numberOfSCCs] // array recording each SCC's level

Algorithm

```

function TaiArray(SCCGraph): int[1..numberOfSCCs]
procedure: level_sort
begin
    scc_number:=getNumberOfSCCs(SCCGraph);
    for i:=1 to scc_number do
        mark[i]:=false;
        for i:=1 to scc_number do
            if mark[i]==false then
                level_sort(i);
            end.

```

A.6 Procedure level_sort

This is a topological sorting.

Input: n: int // scc_number

Algorithm

```

procedure level_sort(n)
var: sucs: int[1..V]; //array records the successors
begin
    mark[n]:=true;

```

```

for i:=1 to V do
    sucs[i]:=0;          // Initializes the successors
sucs:=suc(n);          // node n's successors
if sucs[1]==0 then
    TaiArray[n]:=1;
else
    i:=1;
    while sucs[i]>0 do
        begin
            k:=sucs[i];
            if mark[k]=false then
                level_sort(k);
            i++;
        end;
    TaiArray[n]:=max(suc_level[n])+1;
endif;
end.

```

A.7 Function maxWeightEdge

This function calculates the weight of an edge, which is the product of the number of incoming edges of the start node and the number of out-going edges of the end node.

Input: original_Graph: int[1..V, 1..V]
 SCCs[j, 1..V]: int[1..V]//the classes in SCCs[j]

Output: maxWeightEdge[2] // the selected edge

Algorithm

```

function      maxWeightEdge(original_Graph, SCCs[j, 1..v])
var:    k1,k2: int //class label
           m,n:int //class index in a SCC
           x,y: int //array indices
           edgePool: Set(edge) //set of edges with max weight
           maxWeightValue: int//records the maximum weight
           maxWeightEdge: array[2] //records the edge with the max weight
function:    record //Records the edge with the max weight
               selectEdge //selects an edge
               fan_in //returns the number of in-coming edges of m in a SCC
               fan_out //returns the number of out-going edges of m in a SCC
begin
    original_GraphWeight[1..V,1..V]:=0;
    maxWeightValue:=0;
    maxWeightEdge:=null;
    m:=1;
    k1:=SCCs[j,m];
    while k1>0 do
        begin
            n:=1;
            k2:=SCCs[j,n];
            while k2>0 do

```

```

begin
  if Original_Graph[k1,k2]=='AS' then //select the association edge only
    Original_GraphWeight[k1,k2]:=Fan_in(k1)*Fan_out(k2);
    if maxWeightValue < Original_GraphWeight[k1,k2] then
      maxWeightValue:=Original_GraphWeight[k1,k2];
      maxWeightEdge[1]:=k1;
      maxWeightEdge[2]:=k2;
    endif;
  endif;
  n++;
  k2:=SCCs[j,n];
end;
m++;
k1:=SCCs[j,m];
end;
for x:=1 to V do
  for y:=1 to V do
    begin
      if (Original_GraphWeight[x,y]==maxWeightValue then
        edgePool:=record(x,y,edgePool);
      end;
    end;
  maxWeightEdge:=select(edgePool);
end.

```

A.8 Function deleteEdgeGraph

This function deletes the selected maxWeightEdge from a SCC in a graph. It involves two steps: First, if a class doesn't belong to the SCC, the class's incoming and outgoing edges are deleted; Second, the maxWeightEdge is deleted.

Input: operating_Graph: int[1..V, 1..V]
 SCCs[j, 1..V]: int[1..V]//the classes in SCCs[j]
 maxWeightEdge: int[2] //the edge with the max weight ;

Output: deleteEdgeGraph: string[1..V, 1..V]
 //the graph after the deletion of certain edge from the perating_Graph

Algorithm

```

function deletedEdgeGraph (Operating_Graph, SCCs[j, 1..V], maxWeightEdge[2]):
  string[1..V, 1..V] of string
var:  class_number,k1,k2,n: int;
begin
  class_number:=getNumberOfClasses(SCCs[j,1..V]);
  deletedEdgeGraph:=Operating_Graph;
  for k1:=1 to V do // p is the class label in Operating_Graph
    begin
      classexist:=false;
      for n:=1 to class_number do
        begin
          if (SCCs[j, n]==k1) then

```

```

        classexist:=true;// p exists in SCCs[j,1..V]
    if (!classexist) then
        for k2:=1 to V do
            begin
                deletedEdgeGraph [k1, k2]:=null; // deletes the outgoing edges of k1
                deletedEdgeGraph [k2, k1]:=null; // deletes the incoming edges of k1
            end;
        endif;
    end;
end;
deleteEdgeGraph [maxWeightEdge[1], maxWeightEdge[2]]:=null;
end.

```

Appendix B More details on the running example

The figure below (Figure 22) provides the ICFGs for methods `mA1()`, `mA2()`, and `mA5()` in our running example (Figure 10).

Figure 22 ICFGs for methods `mA1()`, `mA2()` and `mA5()` in Figure 10

Table 14 shows the sequences of server methods triggered by methods `mA1()` and `mA5()` (see Figure 10). As we can see, though `mA5()` calls `mA1()` in a similar way as `mA2()` (the ICFGs for `mA2()` and `mA5()` are similar), the sequences triggered are different because `mA5()` calls `mA1()` using attribute `b2` (instead of parameter `bb` in the case of `mA2()`).

Server methods triggered by the two paths in $mA5()$'s ICFG	Corresponding server method sequences for each symbolic name (server instance)
(aB, mB3()), (b1, mB1()), (b1, mB2()), (b1, mB3()), (b2, mB3())	aB: mB3()
	b1: mB1().mB2().mB3()
	b2: mB3()
(aB, mB3()), (b2, mB2()), (b2, mB1()), (b1, mB3()), (b2, mB3())	aB: mB2()
	b1: mB3()
	b2: mB2().mB1().mB3()
Server methods triggered by the two paths in $mA1()$'s ICFG	Corresponding server method sequences for each symbolic name (server instance)
(b1, mB1()), (b1, mB2()), (b1, mB3()), (b, mB3())	b1: mB1().mB2().mB3()
	b: mB3()
(b2, mB2()), (b2, mB1()), (b1, mB3()), (b, mB3())	b2: mB2().mB1().mB3()
	b: mB3()

Table 14 Server method sequences triggered by method $mA5()$ in Figure 10

Appendix C Instantiation of the tuple metamodel

In this appendix, we use our running example (Figure 10) and client and server sequences in Table 7 and Table 8 to illustrate how instances of the tuple metamodel are created. Given the four client method sequences in Table 7, Figure 23 shows an excerpt of the corresponding tuple metamodel instance. Two `Method` instances appear (names `cM1` and `cM2`). They are both associated with three tuples (called `t1`, `t2`, `t3` and `t4`, `t5`, `t6`, respectively). Those tuples share the same server methods (called `sM1`, `sM2` and `sM3`) and symbolic names.

Figure 23 Instance of the tuple metamodel for client methods `mA1` and `mA2` (Figure 10)

Figure 24 then shows the result of the first algorithm described in Appendix D, i.e., the algorithm that derives server method sequences triggered by client method sequences, given server method sequence triggered by client methods. In this particular case, the server method sequence triggered by methods `mA1` and `mA2` from Figure 23 are used.

Three tuples are created (called tuple1, tuple2 and tuple3). The corresponding client and server method sequences contain methods mA1 and mA2, and mB1, mB2 and mB3 respectively (as described in Table 7). Note that the order is not shown in the figure.

Figure 24 Instance of the tuple metamodel showing tuples for client sequence mA1 . mA2

Appendix D Algorithms for the generation of tuples

Seven algorithms are provided in this appendix. The first two are used to derive tuples for complete client method sequence from tuples associated to individual client methods. Then three algorithms provide implementations to the three redundancy criteria. The last two algorithms are used by the redundancy criteria algorithms to check whether there is an inclusion relationship between two server method sequences, and whether additional tuples are needed to ensure that every client method appears in at least one tuple when applying the redundancy criteria.

D.1 Procedure combineMethodTuples

This procedure uses tuples for individual client methods to produce tuples for complete client method sequences. The procedure calls addTuples, that combines tuples from individual methods for a given symbolic name, for every symbolic names used in tuples triggered by client methods.

Input/Output: origTestReq: TestRequirement

Algorithm

```
procedure combineMethodTuples(origTestReq: TestRequirement)
var:   symbNameUsed: Sequence(SymbolicName)
        clientSeq: Sequence(Method)
        serverSeq: Sequence(Method)
        i, j: int
begin
  serverSeq := new Sequence;
  for i:=1 to origTestReq.clientSeqNumber do
    begin
      clientSeq := origTestReq.method[i]; // one sequence of client method
      symbNameUsed := clientSeq.originalTuples.symbolicName->asSet->asSequence;
      // all the symbolic names used in client methods
      for j:=1 to symbNameUsed->size do
        begin
```

```

        empty serverSeq;
        addTuples(clientSeq, 1, serverSeq, null, symbNameUsed->at(j), origTestReq)
    end
end
end

```

D.2 Procedure addTuples

This procedure produces tuples for a given client method sequence (first parameter, i.e., `classTestSeq`) and for a given symbolic name (fifth parameter, i.e., `sN`) in a recursive manner. Each time a new method in the sequence must be considered, its position in the sequence is provided (second parameter, i.e., `methodNum`). The third and fourth parameters (i.e., `serverSeq1` and `serverSeq2`) are the server sequence triggered by the previous client methods in the sequence (except the last one) and the server sequence triggered by the last previous client method in the sequence. The last parameter (i.e., `testReq`) is the `TestRequirement` instance that holds the client method sequences.

Input:

<code>classTestSeq: Sequence(Method)</code>	// the client method sequence to be used
<code>serverSeq1: Sequence (Method)</code>	// the server method sequence triggered by
	// client methods number 1 to methodNum-2 in
	// <code>classTestSeq</code> .
<code>serverSeq2: Sequence(Method)</code>	// the server method sequence triggered by
	// client method methodNum-1 in
	// <code>classTestSeq</code>
<code>sN: SymbolicName</code>	// the symbolic name to be used
<code>methodNum: int</code>	// the index in <code>classTestSeq</code> of the method to
	// be considered in the current execution of
	// the procedure
<code>testReq: TestRequirement</code>	// the <code>TestRequirement</code> instance to be used

Input/Output: `testReq: TestRequirement`

Algorithm

```

procedure addTuples( classTestSeq: Sequence(Method),
                      methodNum:int,
                      serverSeq1: Sequence (Method),
                      serverSeq2: Sequence (Method),

```

```

                                sN: SymbolicName,
                                testReq: TestRequirement)
var:  clientMethod: Method
        serverSeq: Sequence (Method)
        tuples: Sequence(Tuple)
        t: Tuple
        i: int
begin
  if (serverSeq2 != null) then
    serverSeq := serverSeq1->union(serverSeq2);
  else
    serverSeq := serverSeq1;
  end
  if methodNum > classTestSeq->size then
    t := new Tuple;
    t.symbolicName := sN;
    t.serverSeq := serverSeq;
    t.clientSeq := classTestSeq;
    testReq.tuple->including(t);
  else
    clientMethod := classTestSeq->at(methodNum);
    tuples := clientMethod.originalTuples->select(symbolicName = sN)-
    >asSequence;
    if tuples->isEmpty then
      addTuples(classTestSeq, methodNum+1, serverSeq, null, sN, testReq);
    else
      for i:=1 to tuples->size do
        addTuples(classTestSeq, methodNum+1, serverSeq, tuples->at(i).serverSeq,
          sN, testReq);
      end
    end
  end
end

```

D.3 Procedure removeRedundancyOne

This function implements redundancy criterion 1. It consists in selecting all the tuples that are not contained in other tuples (the server method sequence is not included in a server method sequence from another tuple). Then additional tuples may be added by the call to `checkPendingTuples` so as to make sure that every client method appears in at least one tuple.

Input: testReq: TestRequirement

Output: Set(Tuple)

Algorithm

```

function removeRedundancyOne(testReq: TestRequirement): Set(Tuple)
var: tuple1, tuple2: Tuple
    pendingTuples: Bag(Tuple)
    resultTuples: Bag(Tuple)
    i: int
function:    checkIncludedTuple
procedure:  checkPendingTuples
begin
    resultTuples := testReq.tuple;
    resultTuples->asSequence;
    for i:= 1 to testReq.tuple->size do
        begin
            tuple1 := resultTuples->at(i);
            if (resultTuples->exists(tuple2:Tuple|    checkIncludedTuple(tuple2,tuple1))
            then
                pendingTuples->including(tuple1); //tuple1 is included in another tuple
                resultTuples->excluding(tuple1);
            endif
        end
        // in the end, resultTuples contains the tuples that are not included in other
        tuples
        checkPendingTuples(pendingTuples, resultTuples);
        removeRedundancyOne = resultTuples->asSet;
    end

```

D.4 Procedure removeRedundancyTwo

This function implements redundancy criterion 2. It consists in selecting all the tuples that are not contained in other tuples, while accounting for the kind of the symbolic name (the server method sequence is not included in a server method sequence from another tuple). Then additional tuples may be added by the call to `checkPendingTuples` so as to make sure that every client method appears in at least one tuple.

Input: testReq: TestRequirement

Output: Set(Tuple)

Algorithm

```

function removeRedundancyTwo(testReq: TestRequirement): Set(Tuple)
var: tuple1, tuple2: Tuple
    pendingTuples: Bag (Tuple)
    resultTuples: Bag(Tuple)
    i: int
function:    checkIncludedTuple
procedure:  checkPendingTuples
begin

```

```

resultTuples := testReq.tuple;
resultTuples->asSequence;
for i:= 1 to testReq.tuple->size do
begin
  tuple1 := resultTuples->at(i);
  if (resultTuples->exists(tuple2:Tuple|
    tuple2.symbolicName.kind = tuple1.symbolicName.kind
    and checkIncludedTuple(tuple2,tuple1)) then
    result->including(tuple1); // tuple1 is included in another tuple
    resultTuples->excluding(tuple1);
  endif
end
// in the end, resultTuples contains the tuples that are not included in other
// tuples, accounting for the kind of the associated symbolic names
checkPendingTuples(pendingTuples, resultTuples);
removeRedundancyTwo = resultTuples->asSet;
end

```

D.5 Procedure removeRedundancyThree

This function implements redundancy criterion 3. It consists in selecting all the tuples that are not contained in other tuples, while accounting for the name of the symbolic name (the server method sequence is not included in a server method sequence from another tuple). Then additional tuples may be added by the call to `checkPendingTuples` so as to make sure that every client method appears in at least one tuple.

Input: testReq: TestRequirement

Output: Set(Tuple)

Algorithm

```

function removeRedundancyThree(testReq: TestRequirement): Set(Tuple)
var: tuple1, tuple2: Tuple
      pendingTuples: Bag (Tuple)
      resultTuples: Bag(Tuple)
      resultTuples: Bag(Tuple)
      i: int
function: checkIncludedTuple
procedure: checkPendingTuples
begin
  resultTuples := testReq.tuple
  resultTuples->asSequence;
  for i:= 1 to testReq.tuple->size do
  begin
    tuple1 := resultTuples->at(i);

```

```

if (resultTuples->exists(tuple2:Tuple|
    tuple2.symbolicName.name = tuple1.symbolicName.name
    and checkIncludedTuple(tuple2,tuple1)) then
    result->including(tuple1); //tuple1 is included in another tuple
    resultTuples->excluding(tuple1);
endif
end
// in the end, resultTuples contains the tuples that are not included in other
// tuples, accounting for the name of the associated symbolic names
checkPendingTuples(pendingTuples, resultTuples);
removeRedundancyThree = resultTuples->asset;
end

```

D.6 Procedure checkPendingTuples

This procedure makes sure that every client method appears in at least one of the tuples selected when applying a particular redundancy criterion. Parameters `pendingTuples` and `resultTuples` are the tuples that were not selected by the redundancy criterion and the tuples that were selected. If a client method appears in a tuple from `pendingTuples` does not appear in any tuple in `resultTuples`, then a tuple from `pendingTuples` must be added to `resultTuples`. The procedure identifies the tuple in `pendingTuples` with the maximum number of client methods that do not appear in the `resultTuples`, and adds it to that `resultTuples`. In other words, we make sure that every client method appears in tuples in `resultTuples` with a minimum number of tuples. This is repeated until every client method appears in at least one tuple from `resultTuples`.

Input: `pendingTuples: Bag (Tuple)`

Input/Output: `resultTuples: Set(Tuple)`

Algorithm

```

procedure    checkPendingTuples(pendingTuples:    Bag(Tuple),    resultTuples:
Set(Tuple))
var:    maxDiffTuple, tmpTuple: Tuple
    tmpPendingTuples: Bag (Tuple)
    maxDiffNumber, diffNumber: int
begin
    Tuple maxDiffTuple := new Tuple;
    Tuple tmpTuple := new Tuple;

```



```

while pendingTuples->size > 0 do
begin
  maxDiffNumber := 0;
  maxDiffTuple := null;
  tmpPendingTuples := pendingTuples;
  for i:= tmpPendingTuples->size downto 1 do
  begin
    //looking for the tuple with the maximum number of different client methods
    tmpTuple := tmpPendingTuples->at(i);
    if resultTuples.clientSeq->includes(tmpTuple.clientSeq) then
      pendingTuples->excluding(tmpTuple);
    else
      diffNumber := resultTuples.clientSeq->asset
        ->reject(tmpTuple.clientSeq->asSet)->size;
      if diffNumber > maxDiffNumber then
        maxDiffNumber := diffNumber;
        maxDiffTuple := tmpTuple;
      endif
    endif
  end
  resultTuples->including(maxDiffTuple);
  pendingTuples->excluding(maxDiffTuple);
end //end while loop
end

```

D.7 Function checkIncludedTuple

This function is used to check whether a tuple's server method sequence is included in another tuple's server method sequence. If the second parameter (tuple2) is included in the first parameter (tuple1) then the function return true. Server method sequence inclusion means that the same methods appear in the two sequences in the same order.

Input: tuple1, tuple2: Tuple

Output: boolean

Algorithm

```

function checkIncludedTuple(tuple1, tuple2: Tuple): boolean
var:  subSeq: Sequence
      i: int
begin
  if tuple1.server->size >= tuple2.server->size then
    for i:=1 to tuple1.server->size do
    begin
      if tuple1.server->size > (i + tuple2.server->size) then
        return false;
      endif
      if tuple1.server->at(i) = tuple2.server->first then

```

```
        subSeq := tuple1.server->subsequence(i, i+tuple2.server->size-1);
        if subSeq = tuple2.server then
            return true;
        endif
    endif
end
return false;
else
    return false;
endif
end
```

Appendix E Instantiation of the coupling information metamodel

In this appendix we illustrate the instantiation of the coupling information metamodel in Figure 13 (Section 4.3.1.3) using our running example in Figure 10. Note that we split the object diagram, the instance of the metamodel, in several parts so as to illustrate with simple object diagrams the different coupling between client and server classes (the object diagrams below are not meant to be complete).

First, Figure 25 illustrates the call coupling that exists between client class A (i.e., client method `mA1`) and server class B (i.e., server method `mB1`). The call site at line 11 in class A is in client method `mA1` (the caller method) and is a call to method `mB1` in class B. `mB1` has one return site at line 7 in class B.

Figure 25 Instance of the metamodel (excerpt) for a call coupling

Figure 26 illustrates parameter coupling due to call to method `mB1` (class B) at line 11 in method `mA1` (class A): parameter coupling path involving in parameter `i` (Figure 26.a), and parameter coupling path involving the return value (Figure 26.b). Line 9 in class A is the last definition of actual parameter `i` (type `int`) before the call to method `mB1` in line 11, thus objects named `lastDefBefCall` (class `UseDef`), `actParam` (class `ActualParam`),

call (class CallSite) and callee (class Method). The first use in callee (i.e., method mB1 in class B) is a use of formal parameter j (type int) at line 5, thus objects named callee (class Method), firstUseInCallee (class UseDef) and formParam (class FormalParam). Similarly (Figure 26.b), call to mB1 returns a value: last definition before return at line 6 in class B (object named lastDefBefRet of class UseDef) and first use after call at line 17 in class A (object named firstUseAftCall of class UseDef).

Figure 26 Instance of the metamodel (excerpts) for parameter coupling

Figure 27 illustrates shared attribute coupling (attribute VAR in class C shared by methods mA4 and mB1 in Figure 27.a) and external device coupling (file named File.txt shared by methods mA6 and mA4 in Figure 27.b). Shared attribute VAR (object named sharedAtt of class SharedAttribute in Figure 27.a) is defined (last definition) at line 38 in class A (object named sharedDef of class UseDef) before call to mB1 at line 39 in class A, and used (first use) at line 6 in class B (object named sharedUse of class UseDef). Similarly, external device File.txt (object named extDev of class ExternalDevice in Figure 27.b) is defined (last definition) at line 55 in class A (object named externalDef of class

UseDef) before call to mB5 at line 39 in class A, and used (first use) at line 6 in class B (object named externalUse of class UseDef).

Figure 27 Instance of the metamodel (excerpts) for shared attribute and external device coupling

Appendix F Algorithms for coupling based integration test requirements

Four functions are defined in this appendix for the derivation of call, parameter, shared attribute and external device coupling paths respectively.

F.1 Function computeCallCouplingPath

This function computes the set of call coupling paths for a client method.

Input: caller: Method

Output: callCouplingPaths: Set(Path)

Algorithm:

```
function computeCallCouplingPaths(caller: Method): Set(Path)
var: p: Path;
begin
  for i:=1 to caller.call->size do
    p := new Path;
    p->append(caller.call->at(i))->append(caller.call.callee.return->first);
    callCouplingPaths->including(p);
  end
end
```

F.2 Function computeParameterCouplingPaths

This function computes the parameter coupling paths for a client method. It first consists in deriving coupling paths for in parameters, and then for out parameters.

Input: caller: Method

Output: parameterCouplingPaths: Set(Path)

Algorithm:

```
function computeParameterCouplingPaths(caller: Method): Set(Path)
var: call: CallSite;
```

```

    returnSites: Sequence(ReturnSite);
    aReturn: ResturnSite;
    actPs: Set(ActualParam);
    ap: ActualParam;
    fp: FormalParam;
    actDefs, formUses, actUses, formDefs: Set(UseDef);
    p: Path;
    i, j, k, l: int;
begin
  for i:= 1 to caller.call->size do // for each call in client method
  begin
    call := caller.call->at(i);
    // Devise coupling paths for in parameters
    // get the set of the actual parameters the caller defines before call
    actPs := call.lastDefs.interactingElement
      ->select(ap:InteractingElement|ap.oclType = ActualParam)-
      >asSequence;
    for j:=1 to actPs->size do // for each actual parameter defined before call
    begin
      ap := actPs->at(j);
      fp := ap.formalParam; // formal parameter corresponding to ap
      // get the sequence of last defs before call of ap
      actDefs := call.lastDefs->select(ld:UseDef| ld.interactingElement = ap)
        ->asSequence;
      // get the set of first uses in callee of fp
      formUses := call.callee.firstUses->select(fu:UseDef|fu.interactingElement =
        fp)
        ->asSequence;
      for k:=1 to actDefs->size do // for each last def before call of ap
      begin
        for l:=1 to formUses->size do // for each first use in callee of fp
        begin
          p := new Path;
          p->append(actDefs->at(k))->append(call)->append(formUses->at(l));
          parameterCouplingPaths->including(p);
        end
      end
    end // for each actual parameter

    // Devise coupling paths for out parameters
    // get the set of the actual parameters the caller uses after call
    actPs := call.firstUses.interactingElement
      ->select(ap:InteractingElement|ap.oclType = ActualParameter)-
      >asSequence;
    // get the sequence of return sites of callee
    returnSites = call.callee.return
    for j:=1 to actPs->size do // for each actual parameter used after call
    begin
      ap := callerUseAtts->at(j);
      fp := ap.formalParam;
      // get the set of first uses (after call) of ap
      actUses := call.firstUses->select(fu:UseDef|fu.interactingElement = ap)
        ->asSequence;
      for k:=1 to actUses->size do //for each first use after call of ap
      begin
        for l:=1 to returnSites->size do // for each return site
        begin
          aReturn := returnSites->at(l);
          // get the set of aReturn's last defs of fp

```

```

formDefs := aReturn.lastDefs->select(ld:UseDef|ld.interactingElement =
fp)
->asSequence
for m:=1 to formDefs->size do // for each aReturn's last def of fp
begin
p := new Path;
path->append(formDefs->at(m))->append(aReturn)->append(actUses-
>at(k));
parameterCouplingPaths->including(path);
end
end // for each return site
end // for each first use after call of ap
end // for each actual parameter used after call
end // for each call in client method
end // end function

```

F.3 Function computeSharedAttributeCouplingPaths

This function computes the shared attribute coupling paths for a client method.

Input: caller: Method

Output: sharedAttributeCouplingPaths: Set(Path)

Algorithm:

```

function computeSharedAttributeCouplingPaths(caller: Method): Set(Path)
var: call: CallSite
aReturn: ReturnSite
returnSites: Sequence(ReturnSite)
att: SharedAttribute
callerDefAtts, callerUseAtts: Set(SharedAttribute)
sharedDefs, sharedUses: Set(UseDef)
path: Path
i, j, k, l: int
begin
for i:= 1 to caller.call->size do // for each call in client method
begin
// Compute coupling paths involved in the call
call := caller.call->at(i);
// get the set of the shared attribute the caller defines
callerDefAtts := call.lastDefs.interactingElement
->select(sa:InteractingElement|sa.oclType = SharedAttribute)
->asSequence;
for j:=1 to callerDefAtts->size do
begin // for each shared attribute defined before the call
att := callerDefAtts->at(j)
// get the set of last defs of att in caller
sharedDefs := call.lastDefs->select(ld:UseDef|ld.interactingElement = att)
->asSequence;
// get the set of first uses of att in callee
sharedUses := call.callee.firstUses->select( fu:UseDef|
fu.interactingElement = att)->asSequence;

```



```

for k:=1 to sharedDefs->size do // for each last def before call of att
begin
  for l:=1 to sharedUses->size do // for each first use in callee of att
  begin
    p := new Path;
    path->append(sharedDefs->at(k))->append(call)->append(sharedUses-
      >at(l));
    sharedAttributeCouplingPaths->including(path);
  end
end
end // for each shared attribute defined before the call

// Compute coupling paths involved in the return
// get the set of the shared attribute the caller uses
callerUseAtts := call.firstUses.interactingElement
  ->select(sa:InteractingElement|sa.oclType = SharedAttribute)
  ->asSequence;
// get the sequence of return sites of callee
returnSites = call.callee.return

for j:=1 to callerUseAtts->size do
begin // for each shared attribute (first)used after call in caller
  att := callerUseAtts->at(j);
  // get the set of first uses (after call) of att in caller
  sharedUses := call.firstUses->select(fu:UseDef|fu.interactingElement =
    att)
    ->asSequence
  for k:=1 to sharedUses->size do // for each first use (after call) of att
  begin
    for l:=1 to returnSites->size do // for each return site
    begin
      aReturn := returnSites->at(l);
      // get the set of last def of att before aReturn
      sharedDefs := aReturn.lastDefs->select(ld:UseDef|
        ld.interactingElement = att)->asSequence
      for m:=1 to sharedDefs->size do //for each last def of att
      begin
        p := new Path;
        path->append(sharedDefs->at(m))->append(aReturn)->append(sharedUses-
          >at(k));
        sharedAttributeCouplingPaths->including(path);
      end
    end
  end // for each first use (after call) of att
end // for each shared attribute (first)used after call in caller
end // for each call in client method
end // end function

```

F.4 Function computeExternalDeviceCouplingPath

This function computes the external device coupling paths for a client method.

Input: caller: Method

Output: externalDeviceCouplingPaths: Set(Path)

Algorithm:

```
function computeExternalDeviceCouplingPaths(caller: Method): Set(Path)
var:   call: CallSite
        aReturn: ReturnSite
        returnSites: Sequence(ReturnSite)
        device: ExternalDevice
        callerDefDevs, callerUseDevs: Set(ExternalDevice)
        extDevDevs, extDevUses: Set(UseDef)
        path: Path
        i, j, k, l: int
begin
  for i:= 1 to caller.call->size do // for each call in client method
  begin
    // Compute coupling paths involved in the call
    call := caller.call->at(i);
    // get the set of the external device the caller defines
    callerDefDevs := call.lastDefs.interactingElement
                    ->select(ed:InteractingElement|ed.oclType = ExternalDevice)
                    ->asSequence;
    for j:=1 to callerDefDevs->size do
    begin // for each external device defined before the call
      device := callerDefDevs->at(j)
      // get the set of last defs of device in caller
      extDevDevs := call.lastDefs->select(ld:UseDef|ld.interactingElement =
                                         device)
                     ->asSequence;
      // get the set of first uses of device in callee
      extDevUses := call.callee.firstUses->select(fu:UseDef|
                                                    fu.interactingElement = device)->asSequence;
      for k:=1 to extDevDevs->size do
      begin
        // for each last def before call of device
        for l:=1 to extDevUses->size do
        begin
          // for each first use in callee of device
          p := new Path;
          path->append(extDevDevs->at(k))->append(call)->append(extDevUses-
                                                                >at(l));
          externalDeviceCouplingPaths->including(path);
        end
      end
    end // for each external device defined before the call

    // Compute coupling paths involved in the return
    // get the set of the external device the caller uses
    callerUseDevs := call.firstUses.interactingElement
                    ->select(ed:InteractingElement|ed.oclType = ExternalDevice)
                    ->asSequence;
    // get the sequence of return sites of callee
    returnSites = call.callee.return

    for j:=1 to callerUseDevs->size do
    begin // for each external device (first)used after call in caller
      device := callerUseDevs->at(j);
      // get the set of first uses (after call) of device in caller
```

```

extDevUses := call.firstUses -> select (fu:UseDef|
    fu.interactingElement = device)->asSequence
for k:=1 to extDevUses->size do
begin
    // for each first use (after call) of device
    for l:=1 to returnSites->size do // for each return site
    begin
        aReturn := returnSites->at(l);
        // get the set of last def of device before aReturn
        extDevDefs := aReturn.lastDefs->select( ld:UseDef|
            ld.interactingElement = device)->asSequence
        for m:=1 to extDevDefs->size do //for each last def of device
        begin
            p := new Path;
            path->append(extDevDefs->at(m))->append(aReturn)->append(extDevUses-
                >at(k));
            externalDeviceCouplingPaths->including(path);
        end
    end
end // for each first use (after call) of device
end // for each external device (first)used after call in caller
end // for each call in client method
end // end function

```

Appendix G Jadvisor - Details

```
1  package jadvisor.planner;
2
3  import jadvisor.scheduler.Course;
4
5  .....
6
7  public class StudentPlan implements Serializable {
8      private StudentSemesterPlan[][] _plan;
9
10     public StudentPlan (int numYears, int numSemesters) {
11         ...
12     }
13
14     public void add (Course course, int year, int semester) {
15         if (!_plan[year][semester].contains(course)) {
16             _plan[year][semester].add(course);
17             fireCourseAdded(course, year, semester); // GUI
18         }
19     }
20
21     public void remove (Course course, int year, int semester) {
22         if (_plan[year][semester].contains(course)) {
23             _plan[year][semester].remove(course);
24             fireCourseRemoved(course, year, semester); // GUI
25         }
26     }
27
28     public boolean contains (Course course) {
29         for (int i = 0; i < _plan.length; i++)
30             for (int j = 0; j < _plan[i].length; j++)
31                 if (_plan[i][j].contains(course))
32                     return true;
33         return false;
34     }
35
36     public boolean satisfiesPrerequisites (Course course, int year, int
37     semester) {
38         int counter;
39         List prerequisites = course.getPrerequisites();
40         for (int c = 0; c < prerequisites.size(); c++) {
41             counter = 0;
42             for (int i = 0; i < year; i++)
43                 for (int j = 0; j < semesters(i); j++)
44                     if (_plan[i][j].contains((Course)prerequisites.get(c)))
45                         counter++;
46             for (int j = 0; j < semester; j++)
47                 if (_plan[year][j].contains((Course)prerequisites.get(c)))
48                     counter++;
49             if (counter == 0)
50                 return false;
51         }
52         return true;
53     }
54     .....
55 }
```

Figure 28 Java code chunk for class StudentPlan

```

1  package jadvisor.planner;
3  import jadvisor.scheduler.Course;
   .....
11  public class StudentSemesterPlan implements ListModel, Serializable {
12      private final List _semesterPlan;

15      public StudentSemesterPlan() {_semesterPlan = new LinkedList();}

17      public void add (Course course) {
18          _semesterPlan.add(course);
19      }

21      public void remove (Course course) {
22          _semesterPlan.remove(course);
23      }

25      public boolean contains (Course course) {
26          for (int i = 0; i < _semesterPlan.size(); i++)
27              if (course.equals(_semesterPlan.get(i)))
28                  return true;
29          return false;
30      }
   .....
}

```

Figure 29 Java code chunk for class StudentSemesterPlan

Appendix H Coupling paths for JAdvisor

The table below shows the coupling paths between the methods in StudentPlan and the methods in StudentSemesterPlan.

Abbr: SP = StudentPlan; SSP = StudentSemesterPlan

Method Pair: SP::add(Course course, int year, int semester) => SSP::Contains(Course course)

	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			SP: ln 21			SSP: ln38 SSP: ln39	
Parameter Coupling	course: Course (in)	SP: ln 20	SP: ln 21	SSP: ln 27			
Parameter Coupling	Return variable: Boolean (out)				SSP: ln 28 SSP: ln 29	SSP: ln 28 SSP: ln 29	SP: ln21

Method Pair: SP::add(Course course, int year, int semester) => SSP::add(Course course)

	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			SP: ln 22			SSP: ln19	
Parameter Coupling	course: Course (in)	SP: ln 20	SP: ln 22	SSP: ln 18			

Method Pair: SP::remove(Course course, int year, int semester) => SSP::Contains(Course course)

	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			SP: ln 28			SSP: ln38 SSP: ln39	
Parameter Coupling	course: Course (in)	SP: ln 27	SP: ln 28	SSP: ln 27			
Parameter Coupling	Return variable: Boolean (out)				SSP: ln 28 SSP: ln 29	SSP: ln 28 SSP: ln 29	SP: ln28

Method Pair: SP::remove(Course course, int year, int semester) => SSP::remove(Course course)

	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			SP: ln 29			SSP: ln23	
Parameter Coupling	course: Course (in)	SP: ln 27	SP: ln 29	SSP: ln 22			

Method Pair: SP::satisfiesPrerequisites (Course course, int year, int semester)
=> SSP::contains(Course course)

	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			SP: ln 49			SSP: ln 28 SSP: ln 29	
Call Coupling			SP: ln 52			SSP: ln 28 SSP: ln 29	
Parameter Coupling	prerequisite s: Course (in)	SP: ln 44	SP: ln 49	SSP: ln 27			
Parameter Coupling	prerequisite s: Course (in)	SP: ln 44	SP: ln 52	SSP: ln 27			
Parameter Coupling	Return variable: Boolean (out)				SSP: ln 28 SSP: ln 29	SSP: ln 28 SSP: ln 29	SP: ln49
Parameter Coupling	Return variable: Boolean (out)				SSP: ln 28 SSP: ln 29	SSP: ln 28 SSP: ln 29	SP: ln52

Table 15 The coupling paths in JAdvisor

Appendix I Use of Category-Partition for JAdvisor

The tables below show that how we derive the test cases for JAdvisor using Category-Partition.

Parameters	Categories/Choices	Constraint
course (for add and remove)	course in the plan [if valid year and valid semester]	
	course not in the plan [if valid year and valid semester]	
course (for satisfiesPrerequisites)	Pre-requisites fulfilled [if valid year and valid semester]	
	Pre-requisites unfulfilled [if valid year and valid semester]	
Year	year in the range	[property valid year]
	year out of the range	[error]
semester	semester in the range	[property valid semester]
	semester out of the range	[error]

Table 16 The categories and choices for the parameters

Public void add (Course course, int year, int semester)

	Choices of course	Choices of year	Choices of semester
1	course not in the plan	in range	in range
2	course in the plan	in range	in range
3	Don't care	out of range	in range
4	Don't care	in range	out of range

public void remove(Course course, int year, int semester)

	Choices of course	Choices of year	Choice of semester
1	course in the plan	in range	in range
2	course not in the plan	in range	in range
3	Don't care	out of range	in range
4	Don't care	in range	out of range

public boolean satisfiesPrerequisites(Course course, int year, int semester)

	Choices of course	Choices of year	Choices of semester
1	Satisfies	in range	in range
2	Unsatisfies	in range	in range
3	Don't care	out of range	in range
4	Don't care	in range	out of range

Table 17 The test cases for category partition test

Appendix J Linked List – Details

```
1  public class List{
2      private Node head;
3      private int size = 0;
4
5      ...
6
7      public List() {}
8
9      ...
10
11     // Inserts a node before the Node n in this list.
12     public void insertBefore(int i, Node n) {
13         if (size == 0) {
14             if (n != null)
15                 return ;
16             else
17                 head = new Node(i, null, null);
18         }
19         else {
20             if ((n == null) || (!contains(n)))
21                 return;
22             else {
23                 if ( n.getPrevious() == null ) {
24                     Node newNode = new Node(i, null, n);
25                     n.setPrevious(newNode);
26                     head = newNode;
27                 } else
28                     n.getPrevious().addNodeAfter(i);
29             }
30         }
31         size++;
32     }
33
34     ...
35
36     // Inserts a node after the Node n in this list.
37     public void insertAfter(int i, Node n) {
38         if (size == 0 ) {
39             if (n != null)
40                 return ;
41             else {
42                 head = new Node(i, null, null);
43             }
44         }
45         else {
46             if ((n == null) || (!contains(n)))
47                 return;
48             else
49                 n.addNodeAfter(i);
50         }
51         size++;
52     }
53
54     ...
55
56     // Deletes Node n in this list.
57     public void deleteNode(Node n) {
58         if ((n == null) || (!contains(n)))
59             return;
60         Node e = n.getPrevious();
61         if (e == null) { // n is head
62             head = n.getNext();
63             if (n.getNext() != null)
64                 n.getNext().setPrevious(head);
65         } else
66             e.removeNodeAfter();
67         size--;
68     }
69
70     ...
71
72     // Returns true if Node n is in this list.
73     public boolean contains(Node n) {
74         if (n == null)
75             return false;
76         Node e = head;
77         int key = n.getKey();
78     }
```



```

79     int data = n.getData();
80     while (e != null) {
81         if ((e.getKey() == key) && (e.getData() == data))
82             return true;
83         e = e.getNext();
84     }
85     return false;
86 }
87...
88 }

```

Figure 30 Source code for class List (excerpt)

```

1  public class Node {
2      static int public_key = 0;
3      private int key;
4      private int idata;
5      private Node previous;
6      private Node next;
7
8      public Node(int element, Node previousNode, Node nextNode) {
9          key = public_key++;
10         idata = element;
11         next = nextNode;
12         previous = previousNode;
13     }
14
15     public void addNodeAfter(int element) {
16         Node n = new Node(element, this, next);
17         next = n;
18         if (n.next != null)
19             n.next.previous = n;
20     }
21
22     public void removeNodeAfter() {
23         if (next != null) {
24             next = next.next;
25             if (next != null)
26                 next.previous = this;
27         }
28     }
29
30     ...
50 }

```

Figure 31 Source code for class Node (excerpt)

Appendix K Tuples for Linked List

The table below (Table 18) shows the server sequences, the corresponding symbolic names and the conditions (setting) under which these sequences are triggered (conditions are detailed after the table) for each method in client class `List`.

Method	Server sequence	Symbolic name	Setting (see below for details)
insertAfter	Node	ClassNode	A
	getKey.getData	n	D
	getKey.getData.getNext	head	
	getKey.getData.getKey.getData.addNodeAfter	n	E
	getKey.getData	n	F
	getKey.getData.getNext	head	
	getKey.getData.getNext	e	
	getKey.getData.getKey.getData.addNodeAfter	n	H
	getKey.getData.getNext	head	
insertBefore	Node	ClassNode	A
	getKey.getData	n	D
	getKey.getData.getNext	head	
	getKey.getData.getKey.getData.getPrevious.setPrevious	n	E
	Node	ClassNode	
	getKey.getData	n	F
	getKey.getData.getNext	head	
	getKey.getData.getNext	e	
	getKey.getData.getKey.getData.getPrevious.getPrevious	n	H
deleteNode	getKey.getData.getNext.addNodeAfter	head	
	getKey.getData	n	B
	getKey.getData	n	D
	getKey.getData.getNext	head	
	getKey.getData.getKey.getData.getPrevious.getNext.getNext.getNext	n	G
	setPrevious	e	
	getKey.getData	n	F
	getKey.getData.getNext	head	
	getKey.getData.getNext	e	
	getKey.getData.getKey.getData.getPrevious	n	H
	getKey.getData.getNext.removeNodeAfter	head	
	getKey.getData.getKey.getData.getPrevious.getNext.getNext	n	E
contains	getKey.getData	n	B
	getKey.getData	n	D
	getKey.getData.getNext	head	
	getKey.getData.getKey.getData	n	E
	getKey.getData	n	F
	getKey.getData.getNext	head	
	getKey.getData.getNext	e	
	getKey.getData.getKey.getData	n	H
	getKey.getData.getNext	head	

Table 18 Tuples for methods in class List

Conditions on the setting are the following:

- A Empty list and parameter n equals n (list.size=0 and n=null);
- B Empty list and parameter n doesn't equal n (list.size=0 and n!=null);
- C Non-empty list and parameter n equals n (list.size!=0 and n=null);
- D List with one element different from parameter n (list.size=1 and n!=null, n=head);
- E List with one element, which is parameter n (list.size=1 and n=head);
- F List with two elements and parameter n is not in the list (list.size=2 and n!=null, n is not in the list). In this case, symbolic name e is the second element in the list;
- G List with two elements and the first one is parameter n (list.size=2 and n=head). In this case, symbolic name is the second element in the list;
- H List with at least two elements and the second one is parameter n (list.size>=2 and n is the second element);
- I List with three elements and the second one is parameter n (list.size=3 and n=head);
- J List with at least three elements and the second one is parameter n (list.size>=3 and n is the third element).

Given the client method sequences in Table 11 (Section 4.4.2.2) and the server method sequences they trigger (Table 18), Table 19 below indicates all the tuples that are produced according to our strategy (combination of tuples from each client method). Each cell in the second column shows a client method sequence (first line of the cell) along with one possible triggered server method sequence. The following column provides the corresponding symbolic name. The last column then indicates the conditions under which each client method in the sequence (second column) is executed: these conditions are those determined above. When a setting is not mentioned (we use character '-'), no tuple from the corresponding client method was selected: i.e., the previous method in the client method sequence does not leave the list in a state that allows the execution of any tuple from the following method in the sequence. Note that highlighted tuples are those removed according to redundancy criterion 3.

	Tuples for client method sequences	Symbolic name	Setting
1	insertAfter.insertBefore.deleteNode, Node	Node	A, C, C
2	insertAfter.insertBefore.deleteNode, getKey.getData.getKey.getData.getKey.getData	n	D, D, D
3	insertAfter.insertBefore.deleteNode, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	head	
4	insertAfter.insertBefore.deleteNode, getKey.getData.getKey.getData.addNodeAfter.getKey.getData.getKey.getData.getPrevious.setPrevious.getKey.getData.getKey.getData.getPrevious	n	E, G, H
5	insertAfter.insertBefore.deleteNode, Node	Node	
6	insertAfter.insertBefore.deleteNode, getKey.getData.getKey.getData.getKey.getData	n	F, F, F
7	insertAfter.insertBefore.deleteNode, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	head	
8	insertAfter.insertBefore.deleteNode, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	e	
9	insertAfter.insertBefore.deleteNode, getKey.getData.getKey.getData.addNodeAfter.getKey.getData.getKey.getData.getPrevious.getPrevious	n	H, H, J
10	insertAfter.insertBefore.deleteNode, getKey.getData.getNext.getKey.getData.getNext.addNodeAfter	head	
11	insertAfter.insertBefore.deleteNode, getKey.getData.getNext.removeNodeAfter	e	
12	insertAfter.deleteNode.insertBefore, Node	Node	A, C, C
13	insertAfter.deleteNode.insertBefore, getKey.getData.getKey.getData.getKey.getData	n	D, D, D
14	insertAfter.deleteNode.insertBefore, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	head	
15	insertAfter.deleteNode.insertBefore, getKey.getData.getKey.getData.addNodeAfter.getKey.getData.getKey.getData.getPrevious.getNext.getNext.getKey.getData	n	E, G, D
16	insertAfter.deleteNode.insertBefore, setPrevious	e	
17	insertAfter.deleteNode.insertBefore, getKey.getData.getKey.getData.getKey.getData	n	F, F, F
18	insertAfter.deleteNode.insertBefore, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	head	
19	insertAfter.deleteNode.insertBefore, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	e	
20	insertAfter.deleteNode.insertBefore, getKey.getData.getKey.getData.addNodeAfter.getKey.getData.getKey.getData.getPrevious.getKey.getData	n	H, H, F
21	insertAfter.deleteNode.insertBefore, getKey.getData.getNext.getKey.getData.getNext.removeNodeAfter.getKey.getData.getNext	head	
22	insertAfter.deleteNode.insertBefore, getKey.getData.getNext	e	
23	insertBefore.insertAfter.deleteNode, Node	Node	A, C, C
24	insertBefore.insertAfter.deleteNode, getKey.getData.getKey.getData.getKey.getData	n	D, D, D
25	insertBefore.insertAfter.deleteNode, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	head	
26	insertBefore.insertAfter.deleteNode, getKey.getData.getKey.getData.getPrevious.setPrevious.getKey.getData.getKey.getData.addNodeAfter.getKey.getData.getKey.getData.getPrevious	n	E, H, H
27	insertBefore.insertAfter.deleteNode, Node	Node	
28	insertBefore.insertAfter.deleteNode, getKey.getData.getNext.removeNodeAfter	head	
29	insertBefore.insertAfter.deleteNode, getKey.getData.getKey.getData.getKey.getData	n	F, F, F
30	insertBefore.insertAfter.deleteNode, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	head	
31	insertBefore.insertAfter.deleteNode, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	e	

32	insertBefore.insertAfter.deleteNode, getKey.getData.getKey.getData.getPrevious.getPrevious.getKey. getData.getKey.getData.addNodeAfter.getKey.getData.getKey.getData. getPrevious	n	H, J, J
33	insertBefore.insertAfter.deleteNode, getKey.getData.getNext.addNodeAfter.getKey.getData.getNext.getKey. getData.getNext	head	
34	insertBefore.insertAfter.deleteNode, getKey.getData.getNext.getKey.getData.getNext.removeNodeAfter	e	
35	insertBefore.deleteNode.insertAfter, Node	Node	A, C, C
36	insertBefore.deleteNode.insertAfter, getKey.getData.getKey.getData.getKey.getData	n	D, D, D
37	insertBefore.deleteNode.insertAfter, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	head	
38	insertBefore.deleteNode.insertAfter, getKey.getData.getKey.getData.getPrevious.setPrevious.getKey. getData.getKey.getData.getPrevious.getKey.getData	n	E, H, D
39	insertBefore.deleteNode.insertAfter, Node	Node	
40	insertBefore.deleteNode.insertAfter, getKey.getData.getNext.removeNodeAfter.getKey.getData.getNext	head	
41	insertBefore.deleteNode.insertAfter, getKey.getData.getKey.getData.getKey.getData	n	F, F, F
42	insertBefore.deleteNode.insertAfter, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	head	
43	insertBefore.deleteNode.insertAfter, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	e	
44	insertBefore.deleteNode.insertAfter, getKey.getData.getKey.getData.getPrevious.getPrevious.getKey. getData.getKey.getData.getPrevious.getKey.getData	n	H, J, F
45	insertBefore.deleteNode.insertAfter, getKey.getData.getNext.addNodeAfter.getKey.getData.getNext.getKey. getData.getNext	head	
46	insertBefore.deleteNode.insertAfter, getKey.getData.getNext.removeNodeAfter.getKey.getData.getNext	e	
47	deleteNode.insertAfter.insertBefore, getKey.getData	n	B, -, -
48	deleteNode.insertAfter.insertBefore, getKey.getData.getKey.getData.getKey.getData	n	D, D, D
49	deleteNode.insertAfter.insertBefore, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	head	
50	deleteNode.insertAfter.insertBefore, getKey.getData.getKey.getData.getPrevious.getNext.getNext	n	E, B, B
51	deleteNode.insertAfter.insertBefore, getKey.getData.getKey.getData.getPrevious.getNext.getNext.getNext. getKey.getData.getKey.getData	n	G, D, D
52	deleteNode.insertAfter.insertBefore, getKey.getData.getNext.getKey.getData.getNext	head	
53	deleteNode.insertAfter.insertBefore, getKey.getData.getKey.getData.getPrevious.getNext.getNext.getNext. getKey.getData.getKey.getData	n	I, F, F
54	deleteNode.insertAfter.insertBefore, getKey.getData.getNext.getKey.getData.getNext	head	
55	deleteNode.insertAfter.insertBefore, getKey.getData.getNext.getKey.getData.getNext	e	
56	deleteNode.insertAfter.insertBefore, getKey.getData.getKey.getData.getKey.getData	n	F, F, F
57	deleteNode.insertAfter.insertBefore, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	head	
58	deleteNode.insertAfter.insertBefore, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	e	
59	deleteNode.insertAfter.insertBefore, getKey.getData.getKey.getData.getPrevious.getKey.getData.getKey. getData	n	H, D, D
60	deleteNode.insertAfter.insertBefore, getKey.getData.getNext.getKey.getData.getNext	head	
61	deleteNode.insertAfter.insertBefore, getKey.getData.getKey.getData.getPrevious.getKey.getData.getKey. getData	n	H, F, F
62	deleteNode.insertAfter.insertBefore, getKey.getData.getNext.getKey.getData.getNext	head	
63	deleteNode.insertAfter.insertBefore, getKey.getData.getNext.getKey.getData.getNext	e	
64	deleteNode.insertBefore.insertAfter, getKey.getData	n	B, -, -
65	deleteNode.insertBefore.insertAfter, getKey.getData.getKey.getData.getKey.getData	n	D, D, D
66	deleteNode.insertBefore.insertAfter, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	head	
67	deleteNode.insertBefore.insertAfter, getKey.getData.getKey.getData.getPrevious.getNext.getNext	n	E, B, B

68	deleteNode.insertBefore.insertAfter, getKey.getData.getKey.getData.getPrevious.getNext.getNext.getNext. getKey.getData.getKey.getData	n	G,D,D
69	deleteNode.insertBefore.insertAfter, getKey.getData.getNext.getKey.getData.getNext	head	
70	deleteNode.insertBefore.insertAfter, getKey.getData.getKey.getData.getPrevious.getNext.getNext.getNext. getKey.getData.getKey.getData	n	I,F,F
71	deleteNode.insertBefore.insertAfter, getKey.getData.getNext.getKey.getData.getNext	head	
72	deleteNode.insertBefore.insertAfter, getKey.getData.getNext.getKey.getData.getNext	e	
73	deleteNode.insertBefore.insertAfter, getKey.getData.getKey.getData.getKey.getData	n	F,F,F
74	deleteNode.insertBefore.insertAfter, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	head	
75	deleteNode.insertBefore.insertAfter, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext	e	
76	deleteNode.insertBefore.insertAfter, getKey.getData.getKey.getData.getPrevious.getKey.getData.getKey. getData	n	H,D,D
77	deleteNode.insertBefore.insertAfter, getKey.getData.getNext.getKey.getData.getNext	head	
78	deleteNode.insertBefore.insertAfter, getKey.getData.getKey.getData.getPrevious.getKey.getData.getKey. getData	n	H,F,F
79	deleteNode.insertBefore.insertAfter, getKey.getData.getNext.getKey.getData.getNext	head	
80	deleteNode.insertBefore.insertAfter, getKey.getData.getNext.getKey.getData.getNext	e	
81	contains, getKey.getData	n	B
82	contains, getKey.getData	n	D
83	contains, getKey.getData.getNext	head	
84	contains, getKey.getData.getKey.getData	n	E
85	contains, getKey.getData	n	F
86	contains, getKey.getData.getNext	head	
87	contains, getKey.getData.Next	e	
88	contains, getKey.getData.getKey.getData	n	H
89	contains, getKey.getData.getNext	head	

Table 19 Tuples for Linked List (from client method sequences in Table 11)

Table 20 indicates the tuples selected according to the three redundancy criteria we defined in Section 4.1.4.

Redundancy criterion	Selected tuples from Table 19	
	# of tuples	Actual tuples
Criterion 3	21	1, 2, 3, 4, 8, 9, 10, 15, 16, 20, 21, 26, 32, 33, 34, 38, 44, 46, 51, 59, 84
Criterion 2	21	1, 2, 3, 4, 8, 9, 10, 15, 16, 20, 21, 26, 32, 33, 34, 38, 44, 46, 51, 59, 84
Criterion 1	17	1, 2, 3, 4, 9, 10, 15, 20, 21, 26, 32, 33, 38, 44, 51, 59, 84

Table 20 Selected tuples for Linked List according to redundancy criteria

Appendix L Coupling paths for Linked List

The table below shows the coupling paths between the methods in `List` and the methods in `Node`.

Method Pair: <code>List::insertAfter(int i, Node n) => Node::getKey()</code>							
	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			List: ln86			Node: ln47	
Parameter Coupling	Node::key				Node: ln47	Node: ln47	List: ln86
Method Pair: <code>List::insertAfter(int i, Node n) => Node::getData()</code>							
	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			List: ln87			Node: ln43	
Parameter Coupling	Node::idata				Node: ln43	Node: ln43	List: ln87
Method Pair: <code>List::insertAfter(int i, Node n) => Node::addNodeAfter(i)</code>							
	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call coupling			List: ln56			Node: ln20	
Parameter Coupling	i: int (in)	List: ln56	List: ln56	Node: ln16			
Method Pair: <code>List::insertBefore(int i, Node n) => Node::getKey()</code>							
	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			List: ln86			Node: ln47	
Parameter Coupling	Node::key (out)				Node: ln47	Node: ln47	List: ln86
Method Pair: <code>List::insertBefore(int i, Node n) => Node::getData()</code>							
	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			List: ln87			Node: ln43	
Parameter Coupling	Node::idata (out)				Node: ln43	Node: ln43	List: ln87
Method Pair: <code>List::insertBefore(int i, Node n) => Node::getPrevious()</code>							
	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			List: ln28			Node: ln55	
Parameter Coupling	Node::pre-vious (out)				Node: ln55	Node: ln55	List: ln28
Method Pair: <code>List::insertBefore(int i, Node n) => Node::setPrevious(newNode)</code>							
	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			List: ln30			Node: ln40	
Parameter Coupling	Node::newNode (in)	List: ln29	List: ln30	Node: ln39			
Method Pair: <code>List::deleteNode (Node n) => Node::getKey()</code>							
	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			List: ln86			Node: ln47	
Parameter Coupling	Node::key (out)				Node: ln47	Node: ln47	List: ln86
Method Pair: <code>List::deleteNode (Node n) => Node::getData()</code>							
	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			List: ln87			Node: ln43	
Parameter Coupling	Node::idata (out)				Node: ln43	Node: ln43	List: ln87
Method Pair: <code>List::deleteNode (Node n) => Node::getPrevious()</code>							
	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			List: ln67			Node: ln55	
Parameter Coupling	Node::pre-vious (out)				Node: ln55	Node: ln55	List: ln67
Method Pair: <code>List::deleteNode (Node n) => Node::getNext()</code>							
	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			List: ln69			Node: ln59	
Parameter Coupling	Node::next (out)				Node: ln59	Node: ln59	List: ln69
Method Pair: <code>List::deleteNode (Node n) => Node::removeNodeAfter()</code>							
	Coup-Var	LDBC	Call-site	FUIC	LDBR	Return-Site	FUAC
Call Coupling			List: ln73			Node: ln28	

Table 21 Coupling paths between List methods and Node methods

Appendix M Use of Category-Partition for Linked List

The following tables show how the test cases for Linked List are generated using Category-Partition.

Parameter/ Environment object	Categories	Choices
Node n	n's status	Null [property Node Null]
		Not null [property Node not null]
List	The length of the List object	Empty [property empty]
		Not Empty [property non-empty]
	Node's position in the list	Node is the head of the list [if non-Empty and Node not null] or [if empty and Node null]
		Node is in the middle of the list [if non-Empty and Node not null]
		Node not in the list

Table 22 The choices for the parameter and environment object

1	Node null, empty list
2	Node null, non-empty list
3	Node not null, empty list
4	Node not null, non-empty list, node is the head
5	Node not null, non-empty list, node is in the middle of the list
6	Node not null, non-empty list, node not in the list

Table 23 The test cases for category-partition test