THESIS

RANDOM GENERATION OF VALID OBJECT CONFIGURATIONS FOR TESTING OBJECT-ORIENTED PROGRAMS

Submitted by

Devadatta Sadhu

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2012

Master's Committee:

Advisor: Sudipto Ghosh

Robert France
Daniel Turk

ABSTRACT

RANDOM GENERATION OF VALID OBJECT CONFIGURATIONS FOR TESTING OBJECT-ORIENTED PROGRAMS

A unit test case for an object-oriented program typically requires the creation of an object configuration on which the method under test is invoked. Certain approaches, such as RANDOOP, perform feedback-directed random test generation. RANDOOP incrementally generates test cases by only extending the valid ones. Invalid test cases are not explored, and thus, RANDOOP can miss the creation of some valid object configurations. In our approach, we generate a new random object configuration for each test case. This configuration may or may not satisfy the multiplicity constraints in the UML class model of the program. Instead of discarding an invalid configuration, we attempt to fix it and generate a valid test case. Since we do not reject any test case, and do not depend on the feedback from previous test cases, our object configurations are likely to obtain a higher coverage of the domain of valid configurations than RANDOOP.

We implemented our approach in a prototype tool called RanTGen, which produces JUnit-style test cases. We also created an Eclipse plugin for RanTGen.

Our preliminary results show that RanTGen takes less time than RANDOOP to generate the same number of test cases. RanTGen test cases kill more mutants and achieve higher coverage in terms of statements, branches, and association-end multiplicity (AEM) than RANDOOP test cases. The AEM coverage criterion defines the set of representative multiplicity tuples that must be created during a test, and

is used to measure coverage of the domain of valid configurations.

ACKNOWLEDGMENTS

I would like to thank Dr. Sudipto Ghosh for his help and support as my research advisor and mentor. I highly appreciate his help in influencing, cultivating, and fine tuning my research ideas, and his detailed and prompt feedback on my reports and findings.

I also thank my committee members, Dr. Robert B. France and Dr. Daniel E. Turk, for their careful study of this thesis and constructive feedback.

I am grateful to my professors, Dr. James Bieman, Dr. Indrajit Ray, Dr. Indrakshi Ray, Dr. Ann Hess, and Russ Wakefield for teaching me courses that greatly helped me in my research.

Thanks a bunch to Sharon Van Gorder and all the other staff members of our department for helping me with various official matters, and for always being so caring towards me.

I thank Dr. Carlos Pacheco, who developed RANDOOP, for helping me with the customization of RANDOOP's test generation engine.

And especially I thank God, who made everything possible.

I dedicate this thesis to my father, Dr. Manoj Kumar Sadhu, my mother, Mrs. Purabi Sadhu, and my brother, Mr. Dwaipayan Sadhu. Thanks to my family for everything – for cradling me, smiling at me, being proud of me, laughing with me, loving me, and most importantly having faith in me no matter what I did.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Algorithms

Chapter 1

Introduction

A test case intended to test a given method in an object-oriented program contains three parts: (1) code to create an initial object configuration, (2) a call to the method under test, and (3) an assertion that determines test success or failure. For a test case to be valid, the initial object configuration must be consistent with the constraints, such as multiplicity constraints, specified in the UML class model for the program.

1.1 The Problem

There exist several test generation techniques to generate test cases. Constraint-based test input generation techniques [6, 12] solve constraints to generate valid test cases, while other systematic test generation techniques, such as Korat [14], exhaustively explore a bounded input space to generate valid test inputs. Systematic techniques may not scale; they typically require considerable time and computational resources when applied to large programs. Random test generation techniques, such as JCrasher [11] and ARTOO [10] do not attempt to satisfy constraints. They generate random test inputs. Such techniques are cheaper but are likely to produce many invalid test inputs.

Some approaches, such as RANDOOP [17] and the guided object selection approach proposed by Wei et al. [23], generate random test inputs that satisfy certain types of constraints. Their test input generation processes are random, and hence

computationally less expensive. The test inputs are checked against the constraints and only valid test inputs are produced. RANDOOP uses feedback from previously generated valid inputs to generate new inputs that are more likely to be valid. However, by only extending the valid test inputs, RANDOOP may miss the opportunity for creating certain valid configurations.

1.2 Motivation for our Approach

The above problem motivated us to propose a new test generation technique. Our approach for generating random object configurations is guided by two principles: (a) instead of extending previously generated test inputs, we generate new random configurations each time in order to produce more varied configurations, and (b) instead of rejecting an invalid configuration, we attempt to fix it and create a valid one.

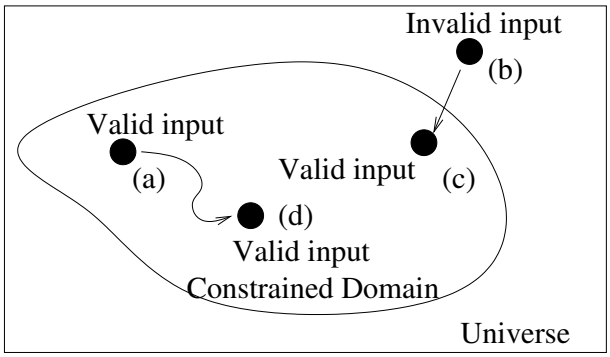


Figure 1.1: Comparison of RANDOOP and our Approach

Figure 1.1 is a visual representation of how our approach and RANDOOP generate object configurations. The rectangle represents the universe of all configurations for a given UML class model. The closed area inside it represents valid configurations, while any point outside the area is an invalid one. A random configuration (e.g., (a)) is first generated without taking into account any constraint. If the configuration is valid, both RANDOOP and our approach accept it. If the configuration (e.g., (b))

is invalid, RANDOOP rejects it and derives another valid configuration (e.g., (d)) from an existing valid configuration (e.g., (a)). Since configuration (d) is obtained by extending configuration (a), they are more likely to be close to each other in the space of the configurations. Our approach does not reject configuration (b) but fixes it to obtain configuration (c), which might be farther away from configuration (a). Thus, our approach is likely to cover different portions of the domain of object configurations.

A criterion for measuring test adequacy in terms of the coverage of possible object configurations is the association-end multiplicity criterion [5]. "An association-end multiplicity specifies how many instances of a class at the opposite end of the association link can be associated with a single instance of a class at the association end." Andrews et al. [5] define the AEM criterion as follows. "Given a test set T and a system model SM, T must cause each representative multiplicity-pair in SM to be created."

Below we provide a concrete example. Figure 1.2(a) shows a partial class model consisting of classes `Shop` and `Product`. According to the multiplicity specifications, a `Product` instance must be associated with a `Shop` instance. The AEM coverage domain of this class model consists of the following multiplicity pairs:

- 1 `ShopInstance`, 0 `ProductInstance`
- 1 `ShopInstance`, 1 `ProductInstance`
- 1 `ShopInstance`, multiple `ProductInstances`

Based on the domain knowledge of the application, a tester can assign an upper limit for the number of `ProductInstances`. Figure 1.2(b) shows the code for the constructors of each class. RANDOOP generates test inputs for this system by creating random sequences of calls to methods and constructors. RANDOOP may initially generate an input that creates a single `Shop` instance. This is shown in Figure 1.2(c).

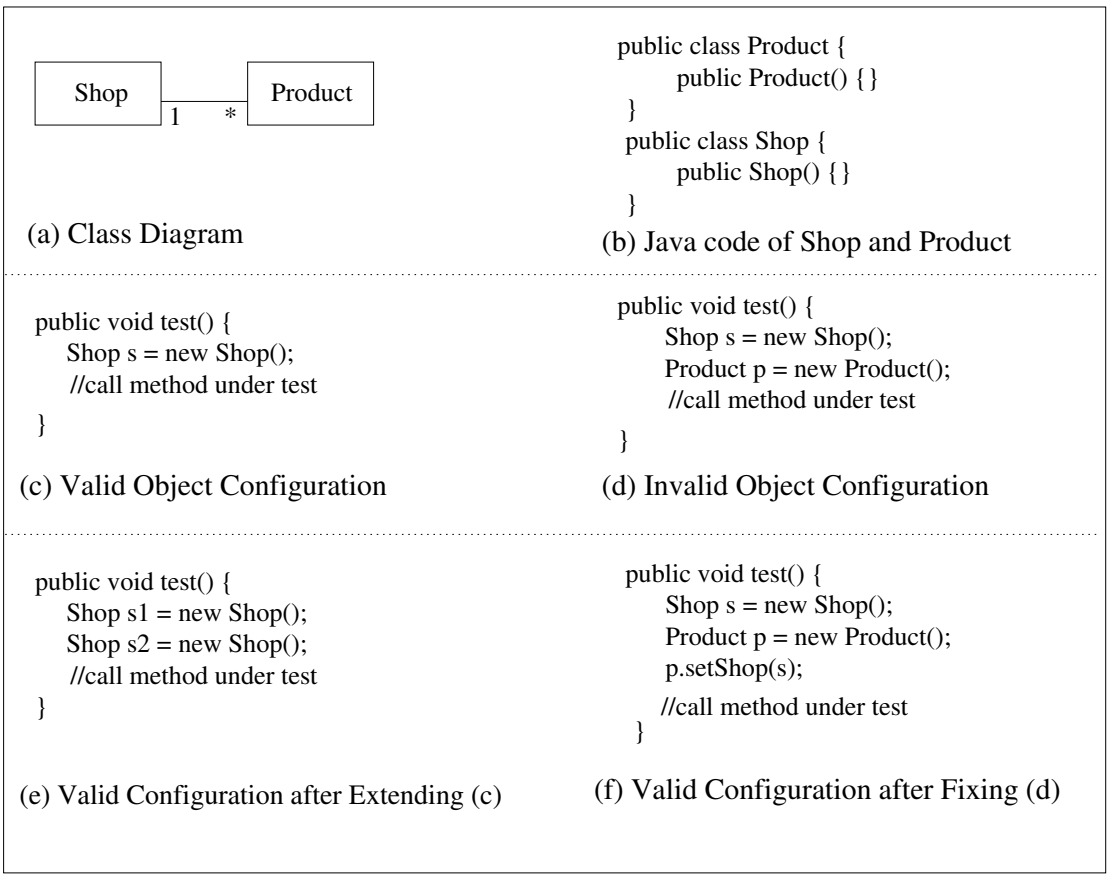


Figure 1.2: Motivating Example

If RANDOOP extends the configuration with a new `Product` instance, the resulting test input is invalid because the product instance is not associated with any shop. Therefore, RANDOOP rejects the test input shown in Figure 1.2(d). RANDOOP creates more valid test inputs by extending the valid test input in Figure 1.2(c) by creating more `Shop` instances, as shown in Figure 1.2(e). The RANDOOP test cases (Figures 1.2(c) and 1.2(e)) cover only 1 configuration in the AEM coverage domain:

- 1 `ShopInstance`, 0 `ProductInstance`

The invalid test input in Figure 1.2(d) can be fixed by adding a link between the `Product` and the `Shop` instances. Figure 1.2(f) shows the fixed test input. However, starting from the test input in Figure 1.2(b), one cannot generate the test input

in Figure 1.2(f) without going through the intermediate invalid test input in Figure 1.2(d). Since RANDOOP is restricted to only extending valid configurations, it cannot generate the input shown in Figure 1.2(f).

In the above example, our approach fixes the invalid configuration and turns it into a valid configuration, as shown in Figure 1.2(f). These test cases (Figures 1.2(c) and 1.2(f)) cover the following configurations:

- 1 `ShopInstance`, 0 `ProductInstance`
- 1 `ShopInstance`, 1 `ProductInstance`

Thus, our test cases achieve higher AEM coverage.

1.3 Contribution of the Thesis

We proposed a test case generation approach which generates random object configurations conformant with the subject program's multiplicity specification, which represents the multiplicity values at each end of an association between two classes in a UML class diagram. Our approach is not dependent on the feedback of previously generated test cases. We do not reject any configuration; if a configuration is invalid, we attempt to fix the inconsistencies within a stipulated time. We implemented our approach in a prototype tool called **RanTGen**. We also created an Eclipse plugin version of our tool. RanTGen generates JUnit test cases, which can be executed using the JUnit testing framework.

We performed a case study to compare the performance of RanTGen and RANDOOP. The results of our evaluation show that RanTGen performed better than RANDOOP in terms of fault detection effectiveness, and coverage of statements, branches, and AEM. RanTGen also took less time than RANDOOP to generate the same number of test cases.

1.4 Organization of the Thesis

The thesis is organized as follows. Chapter 2 presents a summary of related research on systematic test generation and random test generation techniques. Chapter 3 describes the steps of our approach with an example. Chapter 4 explains the architecture of RanTGen. Chapter 5 reports and analyzes the results of our case studies, and lists the threats to validity. Chapter 6 presents our conclusions and outlines directions for future work.

Chapter 2

Related Work

In this section, we discuss existing test generation approaches related to our work. Section 2.1 discusses different systematic test generation approaches. Section 2.2 presents different random test input generation techniques.

2.1 Systematic Test Input Generation

Systematic techniques use well-defined test objectives to generate test inputs. Many techniques have been proposed to systematically explore test input domains. One type of systematic testing, referred to as constraint-based testing, uses constraint solving as the testing objective. Constraint-based test generation approaches determine a set of constraints that their test inputs need to satisfy, and solve the constraints to generate concrete test inputs. Dynamic symbolic execution is an extension of constraint-solving that combines concrete executions with symbolic execution. This idea has been implemented in tools like the Symstra testing framework [25], the Concolic Unit Testing Engine (CUTE) [20], and jCUTE [19].

Symstra is a test generation framework that generates test cases using symbolic execution of method sequences. It systematically explores the object-state space of the class under test and prunes the state space based on symbolic state comparison. It uses symbolic execution to generate test inputs for complex data structures, such as for the Java container data structures.

CUTE [20] and jCUTE [19] perform *concolic testing*, a systematic technique that performs symbolic execution but uses randomly-generated test inputs to initialize the search. Concolic testing [18] first executes a program with a random test input. Then, the same program path executed by the random test input is executed with symbolic values to collect symbolic constraints at the program branches. To generate a new test input, the predicate of a symbolic constraint is negated, and the new set of constraints is solved. The proposed technique is implemented in a tool called CUTE. CUTE works for C programs while jCUTE works for Java programs. While symbolic execution helps in achieving higher coverage by exploring all feasible paths, concrete execution enables symbolic execution to overcome any incompleteness problem.

Bounded exhaustive generation has been implemented in tools like Rostra [24], Java PathFinder (JPF) [22] and Korat [14]. Rostra generates tests using bounded exhaustive generation with state matching. It detects redundant test inputs generated by automatic test generation techniques and removes these redundant tests without decreasing the quality of the test suite. In the Rostra framework, the state of an object is represented by the values of all its fields. A linearization algorithm is used to linearize these values into a representation string [24]. Two states are *equivalent* if their state-representation strings are the same, and are *non-equivalent* otherwise. Rostra first executes the test cases generated by a test case generator (e.g., JPF) and collects the method arguments to form method argument lists. A *method argument list* is characterized by the method name, method signature, and the argument values for the method. Two argument lists are *equivalent* if their method names, signatures are the same and the argument values are equivalent, and are *non-equivalent* otherwise. Rostra explores each possible combination of non-equivalent object states and non-equivalent method argument lists, and likewise outputs the non-equivalent test cases.

JPF [22] is an "explicit-state software model checker for Java programs" that executes the Java bytecode code in the Java Virtual Machine (JVM), and stores,

matches and restores program states. JPF is often used in the model checking of concurrent programs. It systematically explores all potential execution paths of a Java program to find violations of properties, such as deadlocks and unhandled exceptions. JPF, however, does not create the method sequences for exploration by itself. The user needs to manually implement a "driver" program that calls methods of the classes under test. JPF explores method sequences by exploring this driver.

Korat [14] is a systematic test generation technique that generates all possible test inputs satisfying the integrity constraints for the program under test. Korat exhaustively explores a bounded input space and generates all valid test inputs within the bounded domain. While using Korat, the user writes the constraints for the data structures in the "`repOK`" method that returns a boolean value. The `repOK` method represents a predicate that specifies the representation invariant of the data structure. Korat exhaustively generates all non-isomorphic data structures that satisfy the invariant.

Approaches to model-based testing generate valid test inputs by solving constraints associated with a model. In [6, 12], test inputs are generated by solving constraints in UML class, sequence, and state machine models. Dinh-Trong et al. [12] used information from both class and sequence diagrams of a design model to generate test inputs for the model under test. They generated a *Variable Assignment Graph* (VAG) that integrates relevant information from class and sequence diagrams. The VAG is used to derive test input constraints that are solved by a constraint solver. Extending this work, Bandyopadhyay and Ghosh [6] augmented the VAG with information derived from the state machine diagrams of the participating objects. This enhancement is done in order to include the effects of message sequences on the states of the objects participating in the interaction. The extended VAG is called an *EVAG*.

Test input generation for model transformation systems is an active research area. Approaches that generate tests for model transformation systems [7, 21] generate a

source model that conforms to the constraints of the source metamodel. Constraints of the source metamodel are solved to generate valid inputs. Different test strategies, such as random/unguided strategy and input-domain partition based strategy, based on different test adequacy criteria are used to systematically select a finite set of models from an infinite domain of input models.

2.2 Random Test Input Generation

Even though systematic test generation approaches generate valid test inputs, they are time-consuming and computationally more expensive. Random test generation approaches, on the other hand, do not attempt to solve constraints; and hence, comparatively less expensive.

JCrasher [11], a random test generator for Java programs, constructs a *parameter-graph* that enumerates all known ways to generate values for the parameters of a method under test. To create a test input, JCrasher selects a method to test, and uses the graph to find method calls whose return values can serve as input parameters to the method under test. JCrasher creates huge number of test cases randomly, and attempts to find a fault by causing the program under test to crash. JCrasher defines heuristics to decide whether a Java exception should be considered a program fault. These heuristics help to detect the faults early, thereby stopping further propagation of faults in the code.

Jartege [15] is an unguided random test generation technique that generates unit test cases for Java programs. The technique requires the user to provide the formal specifications in JML, which are used to detect and eliminate the redundant test cases. The main purpose of Jartege is to produce numerous test cases to detect a substantial number of errors at a low cost.

Another type of random testing is adaptive random testing (ART) [8]. ART randomly generates the test cases which are distributed evenly across the input domain.

Given a set of test inputs, the first candidate is selected at random. Each subsequent step selects the next candidate input whose distance is greatest from the last-chosen input. Such test cases can be more effective in finding faults. Ciupa et al. proposed an adaptive random test generation strategy to calculate the distance between objects based on their representation, and implemented a tool called ARTOO [10] that uses their strategy to select a subset of randomly-generate test inputs.

An extension of random testing is the *guided random testing* that uses feedback from prior test cases to generate new test cases. One such feedback-directed test generation approach was proposed by Wei et al. [23]. Their approach aims to satisfy the pre-condition of the method under test by using *guided object selection strategy*. After every test execution, they evaluate predicate clauses on some combinations of objects and maintain a pool of pre-condition satisfying combinations. While creating a new test input, they selected object combinations from the pool that satisfy the pre-condition. Their approach uses feedback from the valid test inputs to build the new test inputs. The invalid test inputs are not utilized.

Eclat [16] is a random test generation tool that uses execution feedback to generate test cases. Eclat generates test cases by performing random generation augmented by automated pruning based on execution results. Eclat infers properties of the object under test from an existing test suite. Using a predefined set of properties, Eclat identifies code sequences that are likely to produce illegal test cases, and discards those sequences, thereby improving the efficiency of its search for fault-revealing inputs.

Another random test generation tool is RANDOOP [17]. Unlike Eclat, RANDOOP does not dynamically derive the object properties, so it does not require any sample execution. RANDOOP creates test cases by incrementally building method sequences. To create a method sequence, RANDOOP randomly selects a method call and selects its arguments from previously generated sequences. As soon as a new sequence is created, it is checked against a set of contracts. Sequences that violate

the contracts are discarded. Only valid sequences are extended to create further new sequences.

Chapter 3

Approach

Our approach generates, for Java programs, random test cases that conform to the program's multiplicity constraints. If a randomly generated test case is inconsistent with the specified constraints, our approach identifies the violation, modifies the test case and turns it into a valid one. The inputs to our approach are: (1) a Java program consisting of Java classes, (2) a method under test, (3) the target class containing the method under test, (4) a list of bounds, one for each class, specifying the maximum and minimum number of instances of the class to be generated, (5) a link bound, specifying the maximum number of links to be generated between the created class instances, and (6) a UML class model of the application, showing the multiplicity constraints.

The class bound and the link bound inputs ensure that a test case should create at least the minimum number of class instances and links. The input class model can be a design class model containing multiplicity information or an implementation class model partially obtained by reverse engineering the code. Since it is difficult, if not impossible, to infer the multiplicity constraints directly from the code, we use a class model of the program and add the multiplicity constraints at the end of each association.

Figure 3.1 shows the components of a random test case. A random $TestCase$ consists of code to create an $ObjectConfiguration$, a $MethodInvocation$ to the method

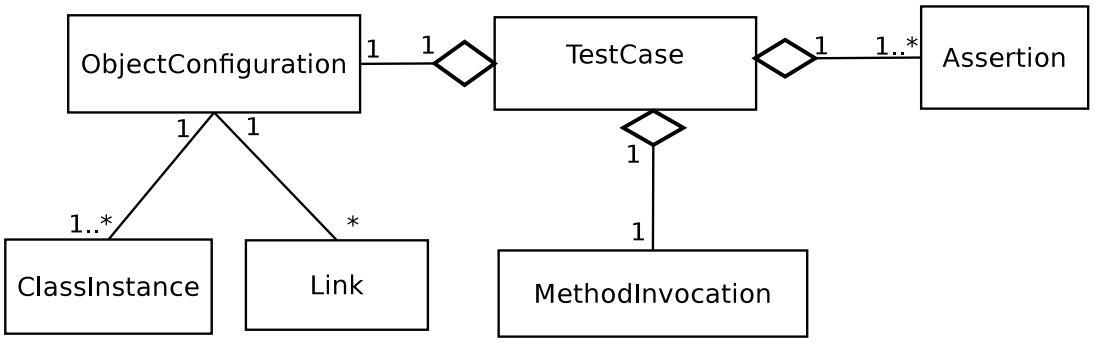


Figure 3.1: Structural Components of a Random Test Case

under test, and an *Assertion* to check the outcome of the test case. Creating an $ObjectConfiguration$ requires (1) the creation of class instances for the application under test, (2) the initialization of attributes of the instantiated classes, and (3) the creation of links between the instances when an association exists between the corresponding classes.

Procedure 1 describes our random test case generation algorithm. First, we generate a random test case. We check the consistency of the object configuration in the test case with the multiplicity constraints in the UML class model of the input application. If the object configuration is valid, the test case is returned. If the configuration is invalid, the multiplicity violations are identified and a violation report is generated. The details of the violation report are discussed in Section 3.2.

Procedure 1 generateValidRandomTestCase(ProgramUnderTest P, MethodUnderTest M, TargetClass T, ClassBoundsMap CB, LinkBound LB, ClassModel CM)

```
1: TestCase tc := generateInitialRandomTestCase(P, T, M, CB, LB)
2: config := tc.getObjectConfiguration()
3: validityCheck := checkValidity(config, CM)
4: while validityCheck.isInvalid() do
5:     violation := validityCheck.getViolation()
6:     config := fixInvalidConfiguration(config, violation)
7:     tc.setObjectConfiguration(config)
8:     validityCheck := checkValidity(config, CM)
9: end while
10: return testCase
```

Each time a violation is removed, the test case is modified to incorporate the fixed object configuration. The modified test case is again checked for validity. The process of checking and modifying is repeated until a valid configuration is obtained.

We use an online shopping application (OSHOP) as an example to illustrate the steps of our approach. Figure 3.2 shows the OSHOP class model. The OSHOP system contains six classes: `Shop`, `Product`, `ProductCatalog`, `Customer`, `CustomerAccount` and `Category`. The method under test is `Shop.makePayment(String customerID, int amount)`, which pays the `amount` on behalf of the `Customer` who is specified by `customerID`.

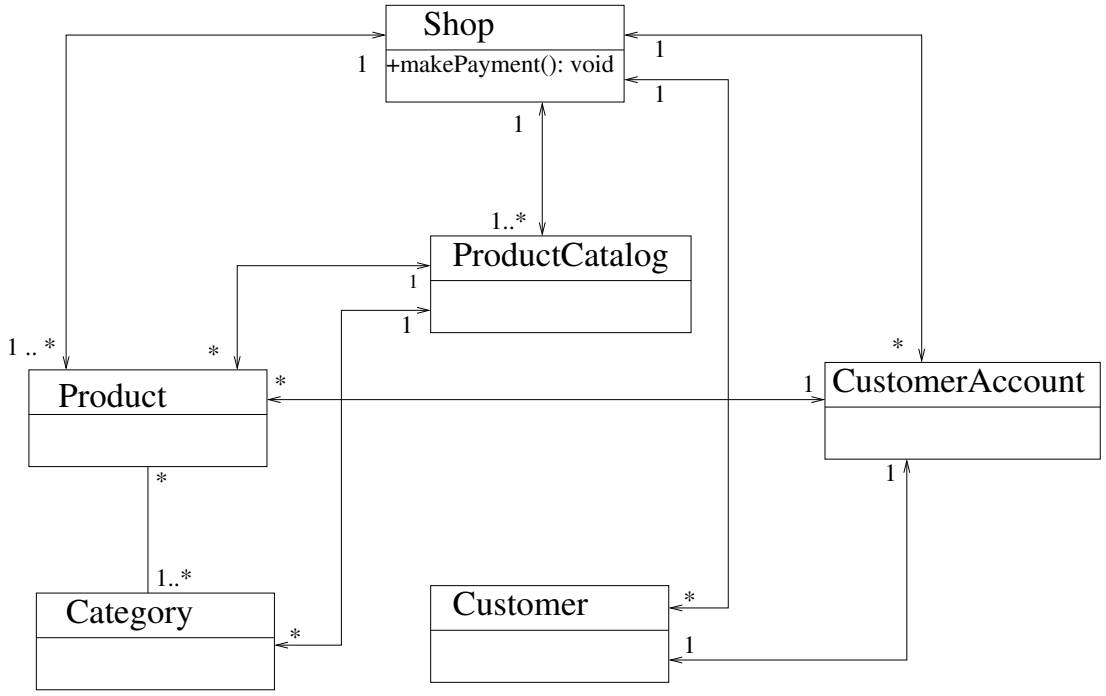


Figure 3.2: OSHOP Class Model

3.1 Generating an Initial Random Test Case

Procedure 2 generateInitialRandomTestCase (ProgramUnderTest P, MethodUnderTest M, TargetClass T, ClassBoundsMap CB, LinkBound LB)

1: ObjectConfiguration $config$:= generateInitialObjectConfiguration(P, T, CB, LB)
2: MethodInvocation $methodInvocation$:= constructMethodInvocation($config$, M)
3: TestCase $initialTestCase$:= new TestCase()
4: $initialTestCase$.setConfiguration($config$)
5: $initialTestCase$.setMethodInvocation($methodInvocation$)
6: $initialTestCase$.setAssertions()
7: return $initialTestCase$

Procedure 2 describes how we generate an initial random test case. First we randomly generate an object configuration for the program under test, P, as described later in Procedure 3. The object configuration, $config$, consists of class instances of P and the links created between them. Once a configuration is created, we construct a method invocation that calls the method under test, M, on the target class, T. An initial test case is generated, and set with the $config$ and the M. Finally, we add the assertions for the method under test. Details of the assertions are explained in Section 4.1.3.

The first step for test case generation is to create an object configuration. Procedure 3 shows how an initial random object configuration is generated. We maintain a $config$ that stores the instances, the links, and the arguments of the method under test. The $config$ is initialized as an empty configuration. As shown in lines 2-7, for each class in the program under test, we generate a random number of instances between the lower bound and the upper bound for that class. Once an instance is created, the instance attributes are initialized by calling their corresponding `public set` methods in the code. We generate random values for the attributes of type `int`, `float`, `double`, `long`, `boolean`, or `String`. We maintain a list of the attributes (lines 9-10) that are initialized because they are candidates for using as parameter values in method calls. The instance is added to the $config$.

Procedure 3 generateInitialRandomConfiguration (ProgramUnderTest P, TargetClass T, ClassBoundsMap CB, LinkBound LB)

```
1: ObjectConfiguration config := new ObjectConfiguration()
2: for all Class C ∈ P do
3:    lowerBound :=CB.getLowerBound(C)
4:    upperBound := CB.getUpperBound(C)
5:    numInstances := generate a random value between lowerBound and upperBound
6:    for i = 1 to (numInstances) do
7:       o := C.newInstance()
8:       for all  Attributes attribute of o do
9:          o.initializeAttribute(attribute)
10:         List attributeList := new List()
11:         attributeList.add(attribute)
12:      end for
13:      config.addInstance(o)
14:   end for
15: end for
16: if config has one or more instances of type T then
17:   targetInstance := a randomly selected instance of type T from the configuration
18:   targeObject := targetInstance.markAsTargetObject()
19: else
20:   targetObject := T.newInstance()
21:   for all  Attributes attribute of targetObject do
22:      targetObject.initializeAttribute(attribute);
23:      attributeList.add(attribute)
24:   end for
25:   config.addInstance(targetObject)
26: end if
27: config.addAttributeList(attributeList)
28: while (configuration has number of links less than LB.getUpperBound() between two
    instances in config) do
29:   Select objects o1 and o2 randomly from the configuration
30:   if there is an association a between the classes of o1 and o2 then
31:      l := create a new link that is an instance of a between o1 and o2
32:      config.addLink(l)
33:   end if
34: end while
35: return config
```

Lines 16-26 describe the instantiation and initialization of the target instance in which the method under test will be invoked. If the configuration already contains one or more instances of the target class, T, we randomly select a $targetInstance$ of type T and mark it as our target object. Otherwise, we create a new instance $targetObject$

of type T, initialize the attributes of *targetObject*, and add *targetObject* to the configuration.

Once all the necessary instances are generated, we generate links between the instances, as shown in lines 28-34. We randomly select two instances and create a link between them, provided that an association exists between their corresponding classes. This process is repeated as long as the number of links does not exceed the *linkBound*. Finally $config$ is returned.

Procedure 4 constructMethodInvocation (ObjectConfiguration $config$, MethodUnderTest M)

```
 1: MethodInvocation methodInvocation := new MethodInvocation()
 2: targetObject := config.targetObject()
 3: methodInvocation.setMethod(targetObject, M)
 4: for all Parameter Param of methodUnderTest do
 5:    attributeList := config.attributeList
 6:    if Param is primitive or String then
 7:       if (Param.type matches with the type of an element in attributeList) then
 8:          paramVal := select a random value from attributeList
 9:       else
10:          paramVal := generate a random value for Param
11:       end if
12:       methodInvocation.addArgs(paramVal)
13:    else
14:       ParamClass := class of Param
15:       if config has one or more instances of P then
16:          paramVal := a randomly selected instance of P from the config
17:       else
18:          paramVal := ParamClass.newInstance()
19:          config.addInstance(paramVal)
20:          if there is an association a between Param and any other class then
21:             l := create a new link that is an instance of a between paramVal and the
                 other class instance
22:             config.addLink(l)
23:          end if
24:       end if
25:       methodInvocation.addArgs(paramVal)
26:    end if
27: end for
28: return methodInvocation
```

Once the object configuration is created, we construct a $MethodInvocation$ to

M, as shown in Procedure 4. In lines 1-3, we create a method invocation that calls M on the target object. As explained earlier. Procedure 3 creates a *targetObject* of the target class type T. In Procedure 4, we obtain this *targetObject* from $config$. Next, we create the arguments for invoking the method under test. If a parameter is primitive or a string, we first check whether it matches with the type of any element in the *attributeList* that the $config$ maintains. If it does, we randomly select a value of that type from the *attributeList*. Otherwise, we create a random value for the parameter. If the parameter is of a class type, we first check the configuration for the existence of instances of the class. If one or more such instances already exist, we randomly select one. Otherwise we create a new instance of the parameter's class and add it to the configuration. We identify the associations between the parameter class and other classes, and create the corresponding links between their class instances. The arguments are added to $MethodInvocation$.

Figure 3.3 shows an initial test case randomly generated by the process described in Procedure 2 for testing the operation `makePayment()`.

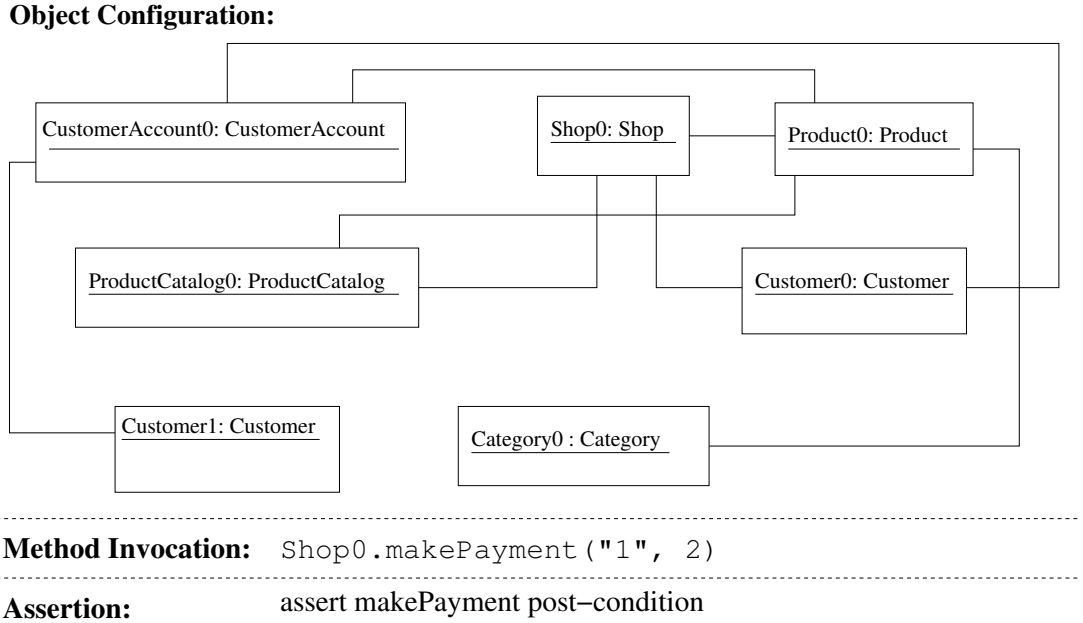


Figure 3.3: Random Test Case for `makePayment()`

3.2 Checking the Consistency of a Configuration

The object configuration associated with a test case is valid if it is consistent with the multiplicity constraints specified in the UML class model. Consider an instance 'a' of class A. Suppose that class A has an association $assc$ with class B. We count the number of instances of B linked to 'a' by $assc$. If the number is not within the lower and upper bounds of the corresponding multiplicity, we construct a violation report. This process is repeated for all instances and all links.

A violation report contains information regarding the violation type, violating instance, violated association, specified upper and lower bounds of the violated association, and the observed number of links. The violation type can take on three values: (1) $redundantLinks$, when there are more links than the upper bound, (2) $tooFewLinks$, when there are fewer links than the lower bound, and (3) $isolatedObjects$, when objects exist without being linked to other objects.

The configuration in Figure 3.3 violates certain multiplicity specifications. Here is an example of a violation report showing one of the violations. According to the class model in Figure 3.2, a `CustomerAccount` instance must be linked to one `Shop` instance. However, in Figure 3.3, the instance `CustomerAccount0` is not linked to any `Shop` instance. The consistency check results in a violation report with the following fields:

- *type*: violation type, $tooFewLinks$
- *obj*: violating instance, `CustomerAccount0`
- *assc*: violated association `Shop_CustomerAccount`
- *specifiedUB*: upper bound of violated association end, 1
- *specifiedLB*: lower bound of violated association end, 1
- *observed*: number of links observed, 0

3.3 Modifying the Test Case

Procedure 5 shows how an invalid configuration of a test case is modified. The variable *violation* represents the violation report.

Procedure 5 fixInvalidConfiguration (ObjectConfiguration $invalidConfiguration$, MultiplicityViolation *violation*)

```
1: if violation.getType() = redundantLinks then
2:    return removeLink(invalidConfiguration, violation)
3: else if violation.getType() = isolatedObjects then
4:    return removeObject(invalidConfiguration, violation)
5: else if violation.getType() = tooFewLinks then
6:    return addLink(invalidConfiguration, violation)
7: end if
```

The modifications are of three types, one for each violation: (a) *removeLink* (Procedure 6), (b) *removeObject* (Procedure 7), and (c) *addLink* (Procedure 8). Each procedure returns a modified configuration. First we check whether the generated object configuration consists of any redundant links. If so, we remove them. Removal of links may render a object as an isolated object, not linked to any other objects in the configuration. Such isolated objects are removed. Next, we check for missing links between the existing objects and create the required ones. While creating a link, if we find an object is missing, we create the object and then create the required links. We have not used any optimization technique to list the modification types in the above order.

Procedure 6 shows how extra links are removed when the *observed* number of links is greater than $specifiedUB$. We need to remove $(observed - specifiedUB)$ links to achieve conformance. In line 3, we first identify all the links that are candidates for removal. Next, we randomly select $(observed - specified)$ links from the candidates and remove them from $config$.

For example, in the object configuration shown in Figure 3.3, the instance `Customer-Account0`, is associated with two `Customer` instances, while the class model specifies

Procedure 6 removeLink (ObjectConfiguration $invalidConfiguration$, MultiplicityViolation $violation$)

```
1: ObjectConfiguration config := invalidConfiguration
2: if violation.observed > violation.specifiedUB then
3:    links := All candidate links in configuration for deletion
4:    linksToDelete := Randomly select (observed − specifiedUB) number of links from
      links
5:    for all l ∈ linksToDelete do
6:       Remove l from config
7:    end for
8: end if
9: return config
```

that a `CustomerAccount` instance must be associated with exactly one `Customer` instance. Therefore, one of the two links needs to be removed. We randomly select one of the links (the one between `CustomerAccount0` and `Customer1`) for removal, producing the test case shown in Figure 3.4.

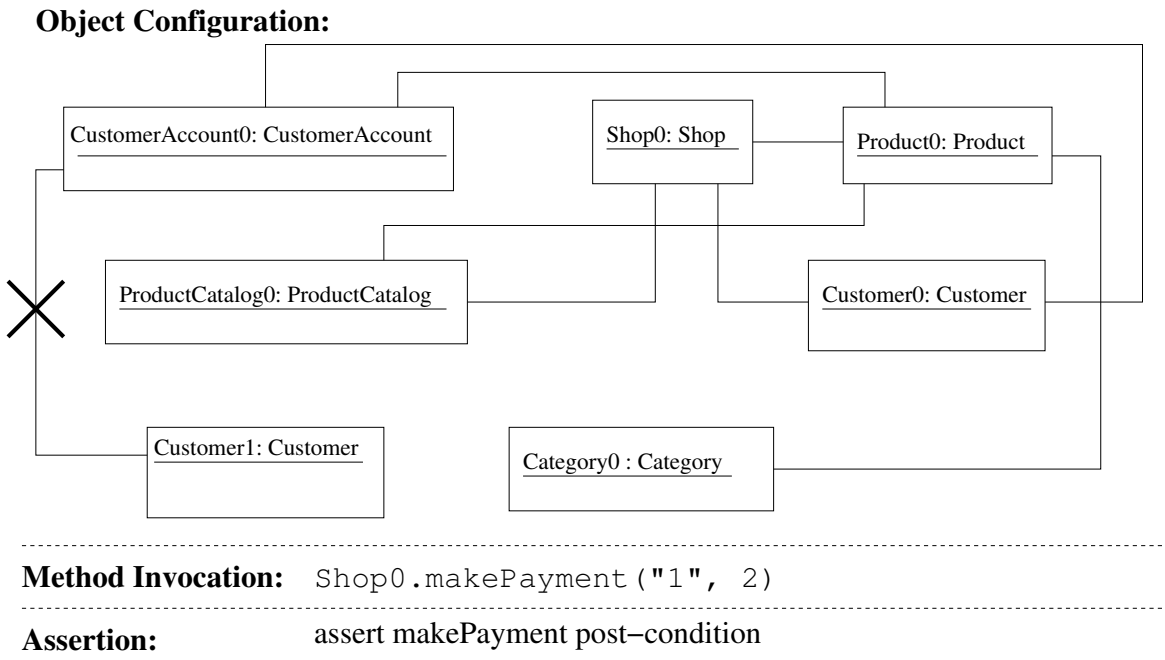


Figure 3.4: Random Test Case for makePayment() After Deleting a Link

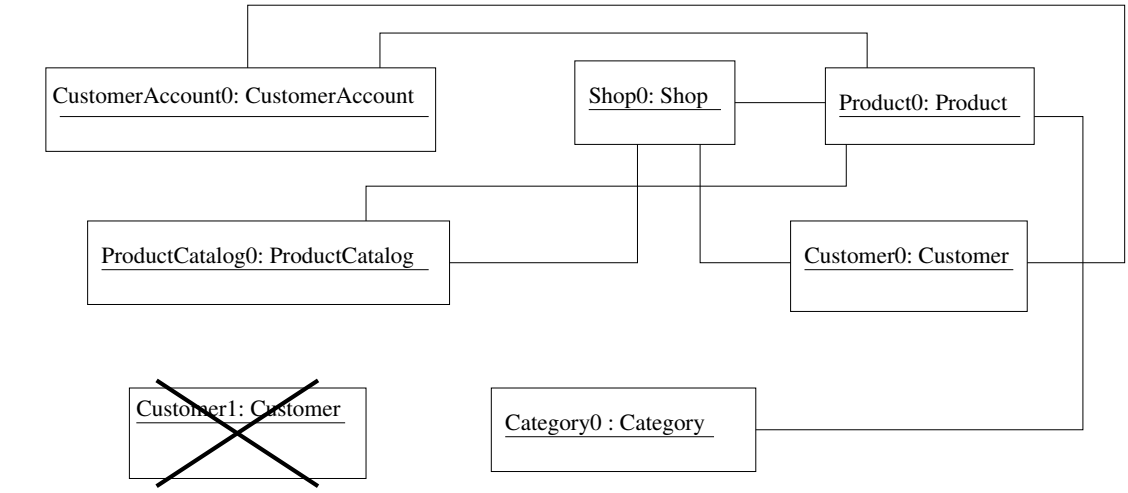
Once the redundant link between `CustomerAccount0` and `Customer1` is deleted, the validity of the resultant configuration is checked again. Link removal may result in the existence of isolated objects, such as the `Customer1` instance in Figure 3.4.

Procedure 7 removeObject (ObjectConfiguration $invalidConfiguration$, MultiplicityViolation $violation$)

```
 1: ObjectConfiguration config := invalidConfiguration
 2: for all Object o ∈ config do
 3:    if o is not linked to any other object then
 4:       Target := Class containing the methodUnderTest
 5:       if o is not the only instance of Target then
 6:          Remove o from config
 7:       end if
 8:    end if
 9: end for
10: return config
```

Procedure 7 removes an object 'o' if it is not linked to any other object in the object configuration. First, we find all such 'o' instances in the invalid configuration that are not linked to any other object. Next, in lines 4-5, we check whether or not 'o' is the only instance of the class containing the method under test. If it is, we do not remove such an instance. Otherwise, we remove 'o' from the invalid configuration. There is a possibility that the only object that must be removed is also the target object for the method under test. If that happens, the fixing procedure will no longer continue checking the validity of the configuration. The produced configuration will only contain the modifications performed until then.

Object Configuration:



Method Invocation: `Shop0.makePayment("1", 2)`

Assertion: assert makePayment post–condition

Figure 3.5: Random Test Case for makePayment() After Deleting an Isolated Object

In Figure 3.4, the `Customer1` instance is not linked to any other instance. Since *Customer* is not the *target* class but *Shop* is, we delete `Customer1` as described in Procedure 7. The resulting test case is shown in Figure 3.5.

Once this fix is applied, the validity check is performed once again. The configuration still contains a multiplicity constraint violation that requires the addition of new link to ensure that `CustomerAccount0` is linked to exactly one `Shop` instance.

Procedure 8 shows that new links are added when *observed* is less than $specifiedLB$, i.e., the violating instance ($violation.obj$) has fewer links than necessary. We need to add ($specifiedLB - observed$) links to achieve conformance. Let C be the class at the end of the violating *assc* opposite to the class corresponding to 'o'. We determine all such instances of C in the invalid configuration that are not already linked with 'o'. If the invalid configuration has a sufficient number of instances of C, we select ($specifiedLB - observed$) instances of C and link them to 'o'. Otherwise, we create the required number of instances of C and add them to $config$. Once an adequate

number of instances of C is present in $config$, we add the links as explained above.

Procedure 8 addLink (ObjectConfiguration $invalidConfiguration$, MultiplicityViolation $violation$)

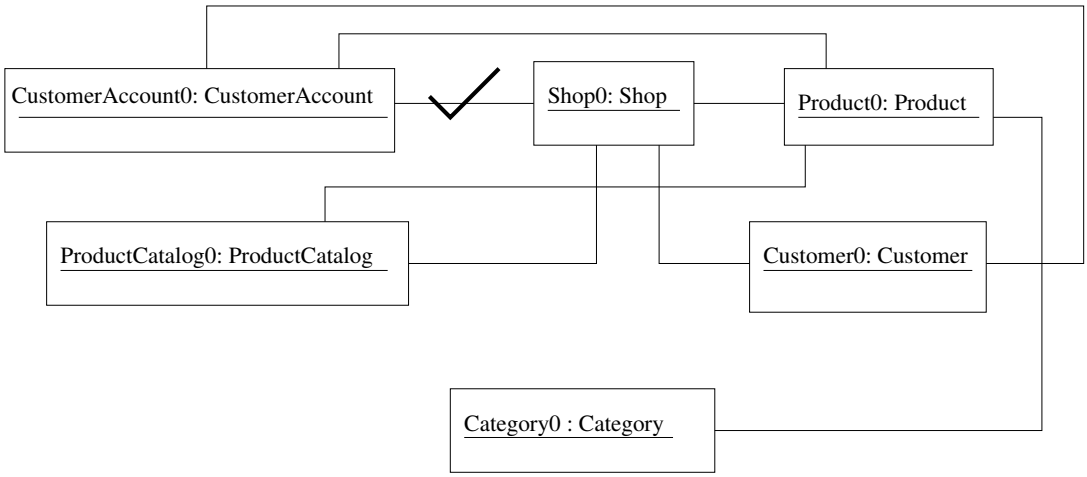
```
1: ObjectConfiguration config := invalidConfiguration
2: if violation.observed < violation.specifiedLB then
3:    C := class at the end of violation.assc opposite to the class of violation.obj
4:    t := List of all instances of the C in config that are not linked to violation.obj
5:    minimumLinksNeeded := violation.specifiedLB − violation.observed
6:    if (t.count() < minimumLinksNeeded) then
7:       Create (minimumLinksNeeded − t.count()) instances of C
8:       Add each instance to config
9:    end if
10:   instancesToBeLinked := Select minimumLinksNeeded number of instances of C
      from config that are not linked to violation.obj
11:   for all i ∈ instancesToBeLinked do
12:      l := create a link between violation.obj and i
13:      Add l to config
14:   end for
15: end if
16: return config
```

In Figure 3.5, `CustomerAccount0` is not linked with any `Shop` instance. Since `Shop0` is the only instance of *Shop*, it is selected to be linked with the `CustomerAccount0`. Adding this link results in the test input shown in Figure 3.6.

Thus, in our modification approach, we address three types of violations. After the modifications are performed, our approach produces an object configuration consistent with the class model, as shown in Figure 3.6.

An invalid configuration can, in general, be fixed in multiple ways. In the above example, instead of removing the isolated `Customer1` instance, we could have created another instance of `CustomerAccount` and then created the necessary links between the instances. This is what we show in Figure 3.7. We create a new instance `CustomerAccount1` and link it to `Customer1`. As per the multiplicity specifications given in the OSHOP class diagram (Figure 3.2), each `CustomerAccount` instance should be linked with a `Shop` instance. `Shop0` being the only instance of in this

Object Configuration:



Method Invocation: `Shop0.makePayment("1", 2)`

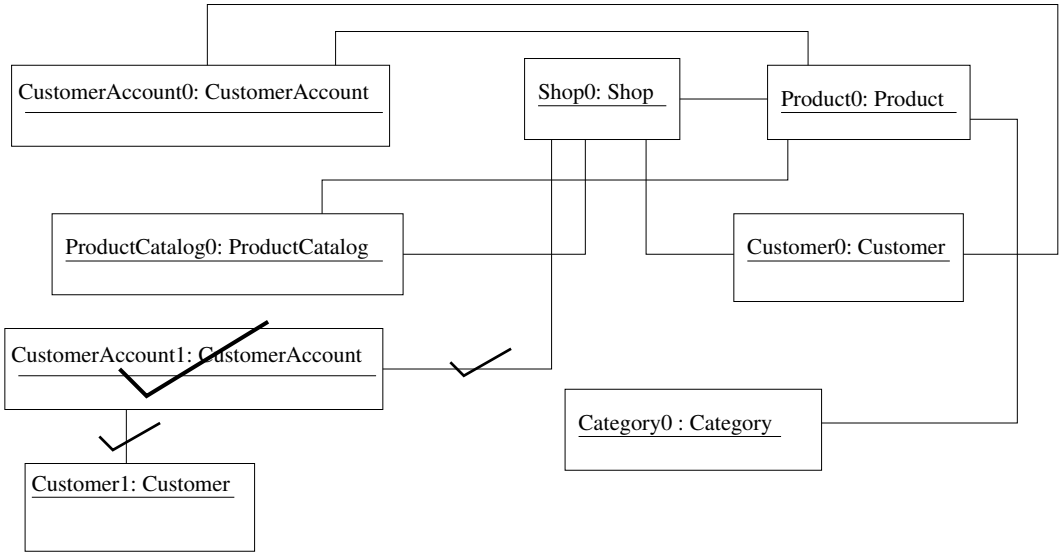
Assertion: assert makePayment post–condition

Figure 3.6: Random Test Case for makePayment() After Adding a Link

object configuration, we link `CustomerAccount1` to `Shop0`.

We observed that even though the above modifications fixed the violation, it required more fixes than the previous fix. The object configuration in Figure 3.5 requires only one fix, deletion of an isolated object, while the configuration in Figure 3.7 requires three fixes. Our approach does not use an optimization technique for selecting the fixes that will require the fewest modifications, and hence the least time to output a valid configuration. It is coincidence that the fix applied by our approach in Figure 3.5 requires fewer modifications than that in Figure 3.7. There might be cases where our current approach may require more number of fixes.

Object Configuration:



Method Invocation: `Shop0.makePayment("1", 2)`

Assertion: assert makePayment post–condition

Figure 3.7: OSHOP Class Model

Chapter 4

Tool Implementation

We implemented our test generation approach in a tool called **RanTGen**, which generates JUnit test cases. A generated test case consists of Java statements that create new objects, initialize the attributes, and create links between the objects to form the object configuration specified in the system's class diagram. These statements are followed by a Java statement that invokes the method under test. Finally, there are Java assertions on the results of the method under test. These test cases are executable in JUnit3.8.1. Section 4.1 presents the RanTGen architecture and explains the functionality of its components. Section 4.2 describes the architecture of RanTGen as an Eclipse plug-in. Section 4.3 describes how a tester may use RanTGen.

4.1 RanTGen Application Architecture

Figure 4.1 shows the components of RanTGen and their dependencies. The components are arranged in two layers. The lower layer represents the components from third party libraries. The upper layer shows the components that we developed. An arrow between two components represents a static dependency between them. The RanTGen implementation contains 4,438 lines of Java code in 24 classes.

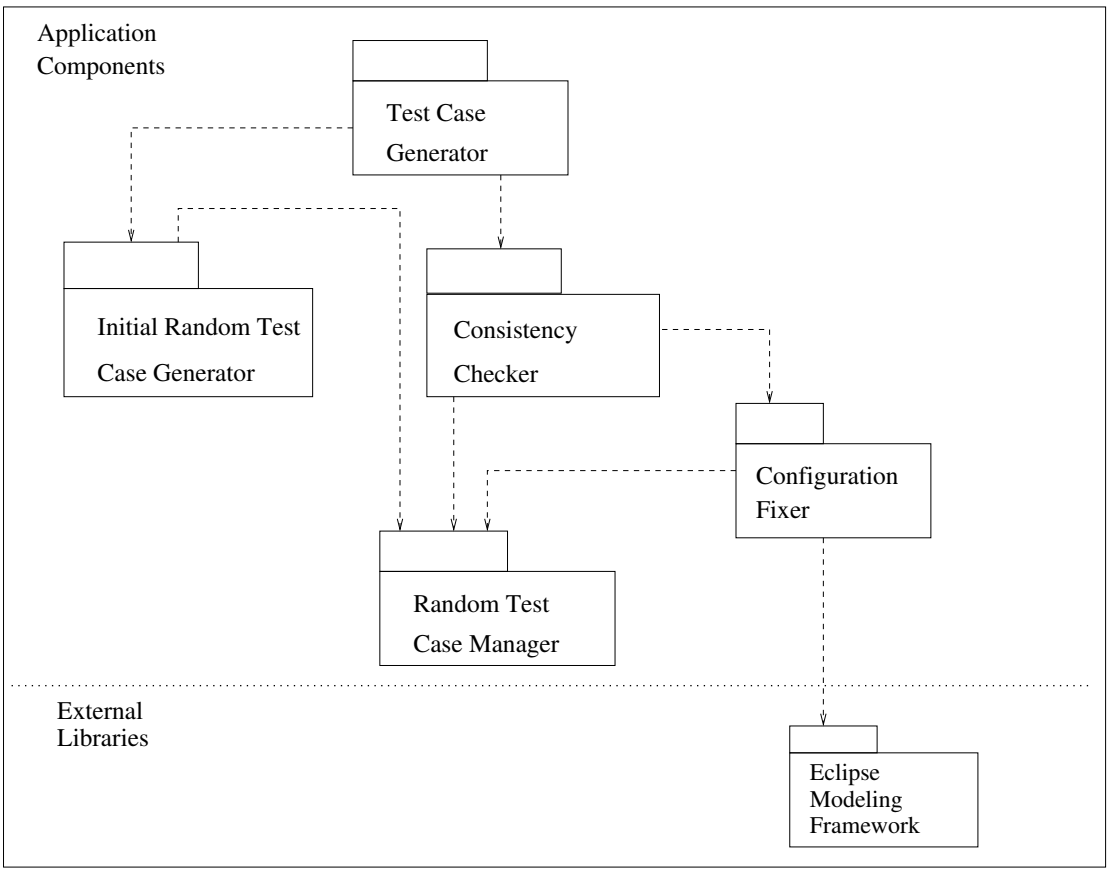


Figure 4.1: Tool Architecture

The `Test Case Generator` initiates the random test case generation process. First, it invokes the `Initial Random Test Case Generator` to randomly generate JUnit test cases for the program under test. These test cases are maintained by the `Random Test Case Manager`. Once the initial test cases are generated, the `Test Case Generator` invokes `Consistency Checker` to check the validity of the object configuration in each test case. In case a violation occurs, the `Consistency Checker` invokes the `Configuration Fixer` to fix the violation.

Figure 4.2 is a high-level sequence diagram showing the interactions between each RanTGen component. The `Test Case Generator` takes the runtime arguments and passes them as parameter values in the `generateInitialTestCase()` operation. Due to lack of space, we are not able to show these arguments in the Figure 4.2. Details

of the runtime arguments are stated in Section 4.3.

When the `generateInitialTestCase()` operation is called on the `Initial Random Test Case Generator`, the latter generates the random instances and the links between them, and creates an object configuration (*objConfig*) with them. Next, the initial generator constructs a method invocation (*methodInvocation*) for the method under test and, creates the assertions (*assertion*). The `Initial Random Test Case Generator` calls the `constructTestCase(objConfig, methodInvocation, assertion)` on the `RandomTestCaseManager` to create a test case. Once the test case is generated, the `Test Case Generator` invokes the `Consistency Checker` to check the consistency of the object configuration with respect to the class model's multiplicity specifications. The `Consistency Checker` gets the class model specifications using the Ecore model of the Eclipse Modeling Framework (EMF) [2]. If the object configuration is inconsistent, the `Consistency Checker` creates an instance of the `Configuration Fixer` and fixes the invalid configuration. The modified configuration is checked again. The checking and fixing of violations continue until a valid configuration is generated or a time-out occurs. The `Consistency Checker` implements a time-out so as to put an upper bound on the time allowed for producing a valid test case. If a time-out occurs, the test case produced will only contain the modifications that were performed till then and some inconsistencies will remain. The `checkAndFixConfig(objConfig)` operation on the `Consistency Checker` finally returns a test case with a fixed configuration to the `Test Case Generator`.

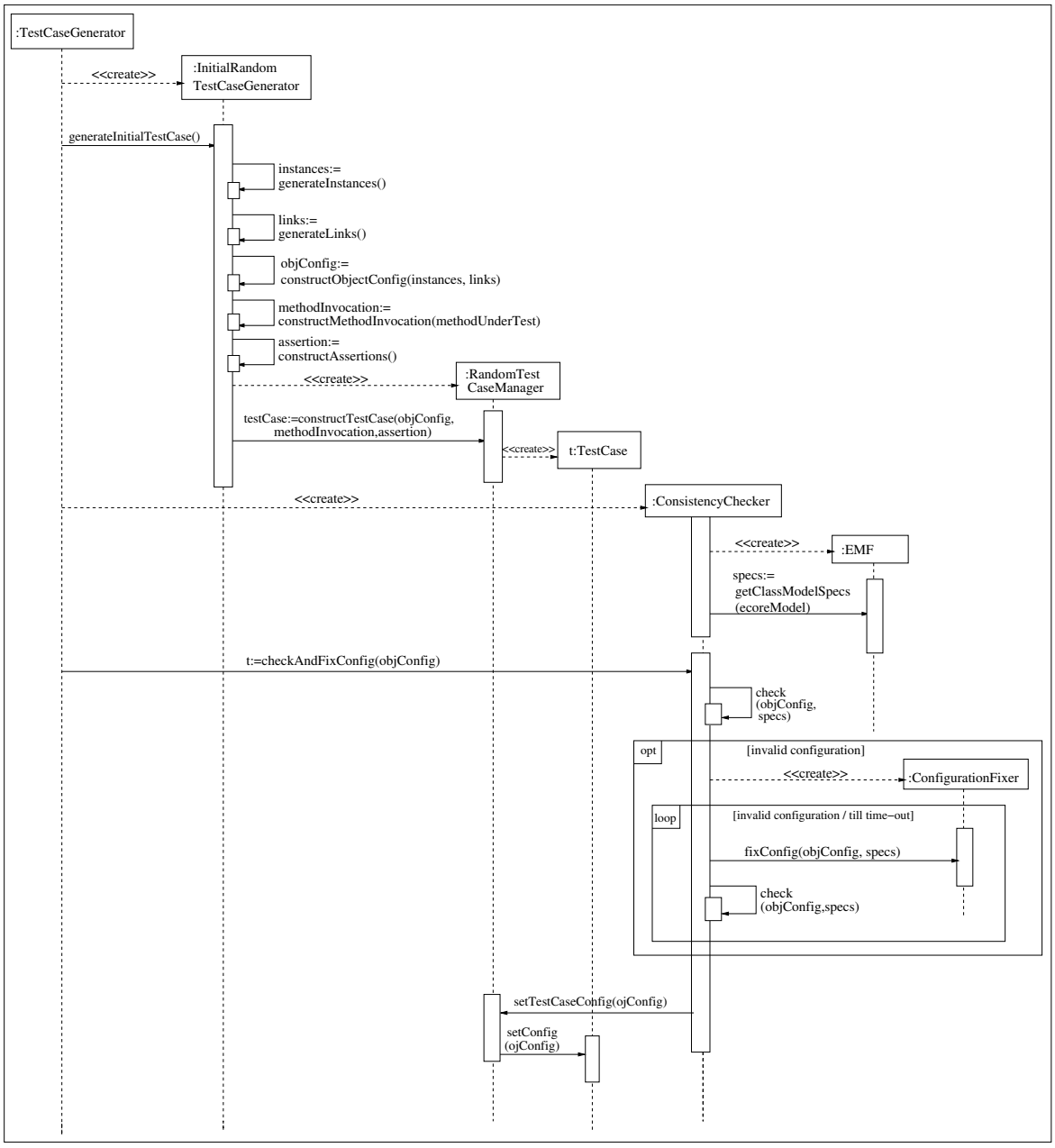


Figure 4.2: Sequence Diagram showing Component Interaction

The following subsections explain in detail the functionality of each RanTGen component.

4.1.1 Third Party Library: EMF

We used the EMF [2] external library to read the UML class model of the system under test. We use the Ecore API that EMF provides to input the class model specifications in XMI. The class model specifications, such as the association type between two classes and the multiplicity values at each end of an association, are provided in Ecore in the XMI format. EMF provides interfaces to retrieve the associations between classes, and the multiplicity specifications on the association ends. This information is used to check the consistency of the object configuration in a randomly generated test case.

4.1.2 Test Case Generator

The component "`Test Case Generator`" is the main driver of the random test case generation process. This component takes the runtime arguments and passes them to the other components that it invokes. The "`Test Case Generator`" sets up the state of the RanTGen components and wires up the dependencies between them by providing the runtime arguments as the parameter values.

4.1.3 Initial Random Test Case Generator

The component "`Initial Random Test Case Generator`" generates JUnit test cases as described in Section 3.1. This initial test case generator creates an initial object configuration of the program under test, values of attributes and, parameters to be passed to the system operation call. It generates instances of a class by calling the constructor of that class. Instance attributes of type `int`, `long`, `double`, `float`, `boolean` and `String` are initialized by calling their corresponding `public set` methods in the program. Once the instances are created and initialized, links between two instances are created, provided an association exists between their corresponding classes. We assume that an association between two classes is implemented as a reference, which

is assigned a value by a `set` method in the code.

If there are two classes A and B, and there is a uni-directional one-to-one association from A to B, then we implement the association in class A as follows:

```
private B newb;
public void setB(B b) {
    this.newb = b;
}
```

In a test case, if `a` and `b` are instances of A and B respectively, a link between the two instances is created by invoking the `set` method as `a.setB(b)`.

If there is a bi-directional association between A and B, then we implement a similar association in class B as well.

```
private A newa;
public void setA(A a) {
    this.newa = a;
}
```

For a bi-directional association, a link from `a` to `b` is created by invoking `a.setB(b)`, while a link from `b` to `a` is created by invoking `b.setA(a)`.

If an instance of class A must be linked to multiple instances of class B, then we implement the association in class A as follows:

```
private List<B> newb;
public void setB(B b) {
    newb.add(b);
}
```

In a test case, if `a` is an instance of A, and `b1, b2, and b3` are instances of B, then calling the above `set` method for each instance of B (`a.setB(b1)`, `a.setB(b2)`,

`a.setB(b3)`) will add these instances to the collection of B instances maintained in `a`.

There can be other ways of maintaining a list of multiple links. We implemented all types of link creation (single or multiple) using the `set` methods. We followed this convention so that at the time of link creation, the `Initial Random Test Case Generator` simply calls the appropriate `set` methods in the program to create the links. By appropriate `set` methods, we mean those `set` methods in the program whose parameter types are of type `java.lang.Class` and whose parameter names do not start with "`java.`". We made an assumption that the name of the classes in the program do not start with "`java.`". We require that these `set` methods be `public`. This is because RanTGen generates JUnit test cases, and JUnit, with its default settings, cannot access methods that do not have `public` visibility.

Once the objects are instantiated and initialized, the `Initial Random Test Case Generator` introduces a method call to the method under test, as described in Procedure 4. Finally, the assertions are added for the method under test. The `Initial Random Test Case Generator` inserts the following default assertions at the end of each test case:

- `assertNotNull`: Check that values of the objects created in the configuration are not null. If the method under test makes an object null reference, the tester needs to negate the assertion.

- `assertEquals`: Check for equality of objects created in the configuration. Whenever a new object is created, we maintain its attributes in a list. The list consists of the type of the object, the name and the value of each of its attributes that are initialized. The `assertEquals` assertion will check if two objects of same type are created with the same values for its attributes. This can help the tester get rid of duplicate objects while running the test cases.

In addition, RanTGen provides a `CheckAssertions` interface that can be used by a tester to write specific assertions (e.g., asserting the post-conditions of the method under test). The interface has a method, `assertCondition`, of return type `String`. A tester can implement this interface and write the specific assertions in the concrete implementation of the `assertCondition` method. The `assertCondition` method will return the assertions, which get appended automatically at the end of the test cases. While `assertNotNull` and `assertEquals` are the actual assert statements, the `assertCondition` method is an assertion generator that generates specific assertions.

In Figure 4.3, we show a concrete implementation of the `assertCondition` method in the `CheckAssertions` interface. In our test case, the method invocation to the method under test `Shop.makePayment(String customerID, int balance)` created the following Java statement:

```
Shop0.makePayment("6", 4)
```

`Shop0` is an instance of the target Class *Shop*. We want to assert that the `customerID`, "6", passed as an argument to the `makePayment` method is for an existing Customer, `CustomerInstance0`. We create a class `AssertCustID` which implements the interface `CheckAssertions`.

```
import java.lang.reflect.Method;

public class AssertCustID implements CheckAssertions {

    public String assertCondition(RandomTestMethodGenerator generator)
    {
        String assertStatement = "";
        Instance instance = generator.generate().getObjectStructure().
                                getTargetInstance();
        if (instance.getOwningClass().getSimpleName().equals("Shop"))
        {
            if (generator.generate().getTestMethodInvokation().
                    getMethod().getName().equals("makePayment"))
            {
                assertStatement = "Assert.assertEquals(\"";
                assertStatement += generator.generate().
                                        getTestMethodInvokation().
                                             getArguments(1);
                assertStatement +="\", CustomerInstance0.
                                            getCustomerID()); ";
            }
        }
        return assertStatement;
    }
}
```

Figure 4.3: Example of Specific Assertion

In Figure 4.3, the assertion generator checks whether the target instance is of type `Shop` and the test method is `makePayment`. If it is so, it creates the appropriate assert statement. We maintain a list of the arguments of the test method in our implementation. In an assertion, the first argument is the expected value. In our assertion, the first argument is the value of the `customerID` passed in the `makePayment` method (i.e., "6"). The second argument in an assertion is the actual value. In this case, it is the `customerID` of `CustomerInstance0`.

The `Initial Random Test Case Generator` component calls the `assertCondition` method while creating a test case. In this example, it instantiates the `AssertCustID`

class as follows:

```
CheckAssertions assertion = new AssertCustID();
```

Next, the `Initial Random Test Case Generator` calls `assertion.assertCondition()`, which returns the assert statement,

```
Assert.assertEquals("6", CustomerInstance0.getCustomerID()).
```

This assert statement gets added to the end of the test case.

4.1.4 Random Test Case Manager

The component "`Random Test Case Manager`" maintains the JUnit test cases generated by the `Initial Random Test Case Generator`. The `Random Test Case Manager` maintains the three components of a test case: object configuration, method invocation, and assertions. Since our approach is focused in generating a valid configuration, this component provides the following interfaces for the object configuration: (1) to query a configuration, and (2) to modify a configuration by adding and deleting instances and links.

4.1.5 Consistency Checker

The component "`Consistency Checker`" checks the validity of a configuration by obtaining the multiplicity specification with the help of the EMF libraries. This component selects each test case maintained by the `Random Test Case Manager`, and queries its corresponding object configuration. It compares the generated configuration's multiplicity values with the multiplicity specifications given in the program's class model. We use the `EReference` property of EMF to query the multiplicity bounds at the end of each class association in the class model. If the object configuration is inconsistent with the class model specifications, the `Consistency Checker` generates a violation report, as described in Section 3.2, and invokes the `Configuration Fixer` component.

4.1.6 Configuration Fixer

The component '`Configuration Fixer`" gets the violation reports from the "`Consistency checker`" and modifies invalid configurations. Depending on the type of violation, as listed in Section 3.2, the `Configuration Fixer` implements the fixes that are described in Section 3.3. The `Configuration fixer` uses the interface of the component `Random Test Case Manager` to modify a configuration of a test case. The fixing procedure continues until a valid configuration is generated or a time-out occurs.

4.2 RanTGen Eclipse Plugin Architecture

RanTGen can be run as a stand-alone application or as an Eclipse plug-in. We created a user interface for the RanTGen plug-in. The RanTGen Eclipse plug-in is built on the Helios edition of Eclipse (v3.6.2). We extended the *org.eclipse.core.runtime.Plugin* class, which provides generic facilities for managing plug-ins, and is an abstract class of the Eclipse SDK. We used the Eclipse Plug-in Development Environment (PDE) [3] to develop the plugin. The Eclipse PDE generates two files for the plugin, `plugin.xml` and `MANIFEST.MF`.

The `plugin.xml` file describes how the RanTGen plug-in extends the Eclipse platform. The information in this file tells Eclipse when and where to use new views, perspectives, dialogs, and wizards that need to be created in the plugin. As a result, Eclipse performs the following extensions:

1. Create an item for the RanTGen plug-in in the Eclipse workbench main menu, by extending *org.eclipse.ui.main.menu*.

2. Create a shortcut icon for the plug-in in the Eclipse workbench toolbar, by extending *org.eclipse.ui.main.toolbar* URI. The icon for the plug-in is provided

as an image file with `.gif` extension. The location of the icon is provided in this file.

The `plugin.xml` file also specifies the name of the concrete handler (the `UI Event Handler` component), which implements the actions of the event that launches the RanTGen plug-in in the Eclipse workbench.

The `MANIFEST.MF` file defines the runtime information of the plugin. This file specifies all the dependencies of the plugin, such as the RanTGen application and the Eclipse JDT plugin.

The plugin architecture of RanTGen is shown in Figure 4.4. We added a layer at the top to the architecture shown in Figure 4.1 to include the plugin components. The middle layer shows the same application components, as shown in Figure 4.1. We also added an external library, Java Development Tools (JDT) [1] to the External Libraries layer, that are needed to build the plugin. In the following subsections, we explain the architecture of the RanTGen plug-in.

4.2.1 UI Event Handler

The component "`UI Event Handler`" extends the `AbstractHandler` class of the *org.eclipse.core.commands* package. The behavior of a UI event is defined via handlers. A user can launch the RanTGen plugin by clicking on the RanTGen icon in the toolbar, or by choosing the RanTGen item from the menu. Once the event to start the RanTGen plugin is fired via Eclipse, the `execute` method in the handler class is executed. The `execute` method launches the user interface (UI) form for the RanTGen plug-in.

4.2.2 UI Form Generator

The component "`UI Form Generator`" generates an entry form to the user to enter the inputs to the RanTGen test generation process. This component extends the

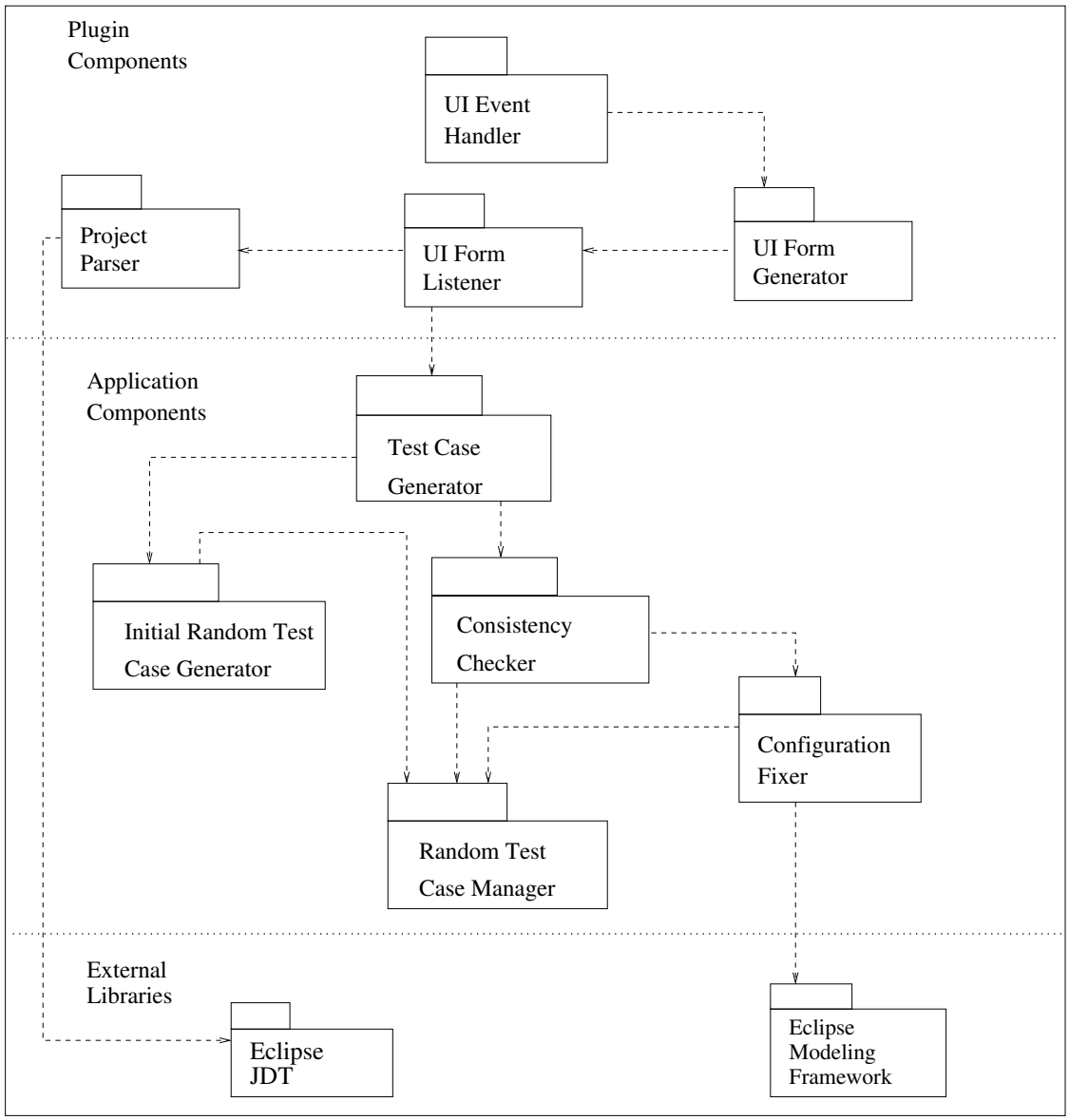


Figure 4.4: Eclipse Plugin Architecture of RanTGen

`JFrame` class of the Java Swing library. The entry form consists of different widgets, such as *combo-boxes* and *textfields*, where the user can select or provide the input arguments, such as selecting the project under test, the package name, the class name, and the name of the method under test. The UI form also provides the user with *browser* dialogs with which the user can browse the file system and select the files that need to be loaded. The UI widgets were created by using the Java Swing

toolkit. The UI form consists of different buttons created for varied purposes, such as to generate test cases or to cancel test case generation. The item events fired up by the UI widgets are performed by the "`UI Form Listener`" component.

4.2.3 UI Form Listener

The component "`UI Form Listener`" listens to events in the entry form and perform actions for those events. In our implementation, whenever an item in a combo-box is changed, an event is fired. The items in the combo-boxes are populated by invoking the interfaces of the "`Project Parser`" component. The `UI Form Listener` component also implements actions for the UI button-click events. When the button to generate the RanTGen test cases is clicked, the `UI Form Listener` gathers all the input information from the UI form and invokes the test case generation method in the `Test Case Generator` component of the RanTGen application, by passing all the gathered information. The action for the cancel button is handled by the `UI Form Listener`, by stopping the test generation process.

4.2.4 Project Parser

The component "`Project Parser`" parses the code of the program under test and helps to populate the items in the combo-boxes in the UI entry form. The combo-boxes that are populated are: (1) the package name combo-box, (2) the class name combo-box, and (3) the method name combo-box. The `Project Parser` sends the project information of the program under test to the Eclipse JDT API. The JDT API finds all the necessary information for the program under test (such as, packages, classes in each package, and methods in each class) and populates the corresponding combo-boxes with relevant information.

4.2.5 Third Party Library: JDT

The Java Development Tools (JDT) library, which is already included in the Eclipse SDK, is used to build plug-ins on the Eclipse platform. JDT provides APIs to access and manipulate Java projects in the Eclipse IDE. It allows a plug-in to access the existing projects in the workspace, and modify and read existing projects. We used JDT to read the program under test and help the `Project Parser` to parse the program. Figure 4.5 shows the Eclipse plug-in for RanTGen.

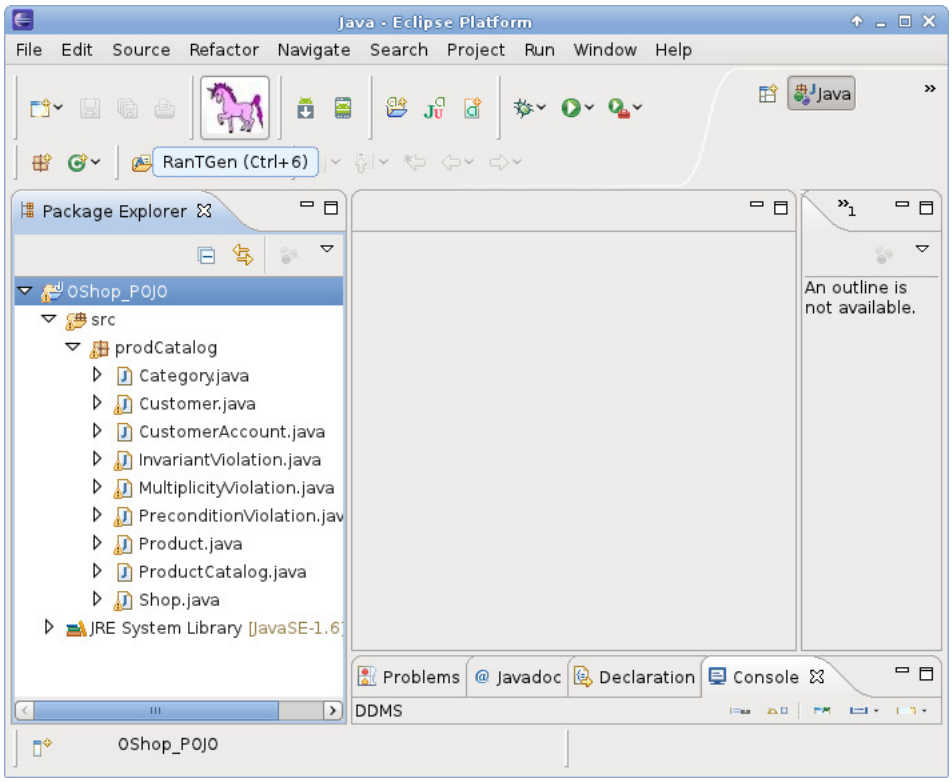


Figure 4.5: RanTGen Eclipse Plug-in

4.3 RanTGen User Manual

In this section, we illustrate the steps for using RanTGen with the help of screenshots.

1. Given a subject program implemented in Java, the user first needs a class model of the program to provide RanTGen the information about the association between classes and the multiplicity specifications. This class model can either be a UML design class model or a class diagram obtained by reverse engineering the Java code. The user uses the Ecore Model Editor of Eclipse Modeling Framework [4] to add the multiplicity values at each association end. The Ecore model has the following components to model a class diagram:

   (a) `EClass`: represents a class, with zero or more attributes and zero or more references.

   (b) `EAttribute`: represents a class attribute that has a name and a type.

   (c) `EReference`: represents one end of an association between two classes. It has flag to indicate a reference class to which it points. It has two fields, **Bound** and **Upper Bound**, to represent the multiplicity values at the end of an association.

   (d) `EDataType`: represents the type of an attribute (e.g., `int`, `float` and `String`).

   Figure 4.6 shows how the inputs are provided in the Ecore Editor.

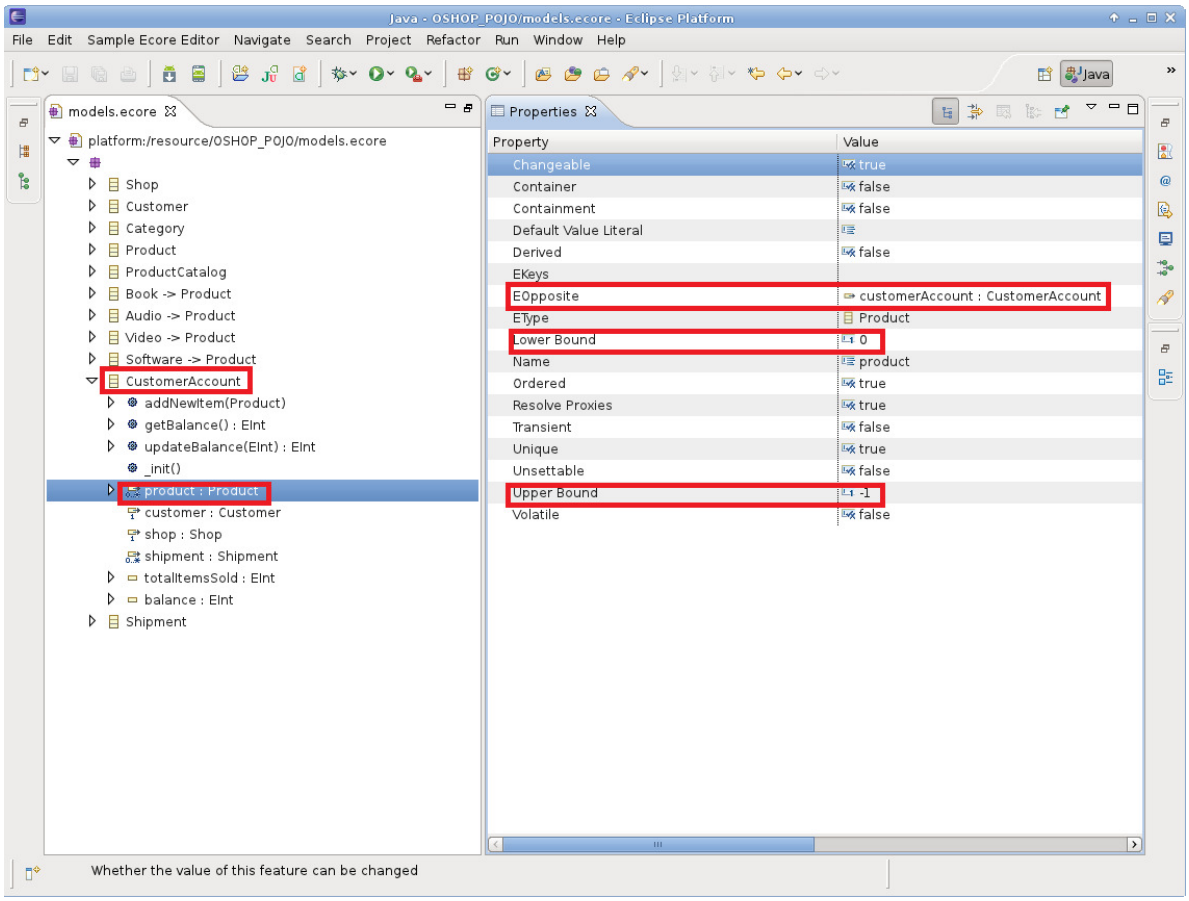


Figure 4.6: Class Model Specifications in Ecore Model Editor

2. The user provides the following arguments to run RanTGen. If RanTGen is run as a stand-alone application, these arguments are provided in the run configuration.

   - Location of the subject program's bytecode (the path where the subject program has its class files).
   - Name of a file that lists the classes that need to be loaded for testing. A program under test may have several packages and many classes. The user may not want to load all the classes to generate a test case. The user lists the names of the classes to be tested in a text file. Each class under

test is specified by its fully qualified name (`packageName.className`) on a separate line.

- Name of the package containing the test class.
- Name of the test class.
- Name of the method under test.
- Number of test cases that need to be generated.
- Name of the directory to which the JUnit test files should be written.
- Name of the class-bounds list file. The user provides in a text file the list of instance bounds, one for each class. Figure 4.7 shows the format of such a file. The first field is the name of the class, and the second and third fields specify the minimum and maximum number of class instances to be generated.

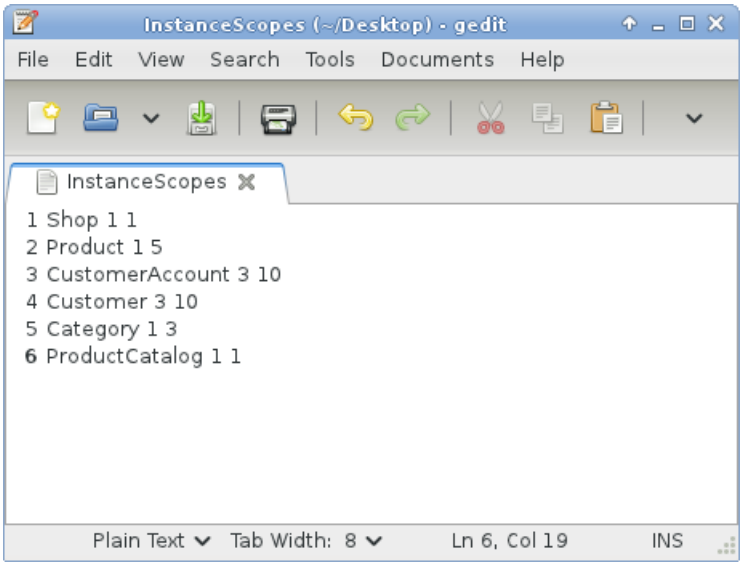


Figure 4.7: File Format for Instance Bounds

Figure 4.8 shows the arguments provided through a run configuration to run RanTGen as a stand-alone application.

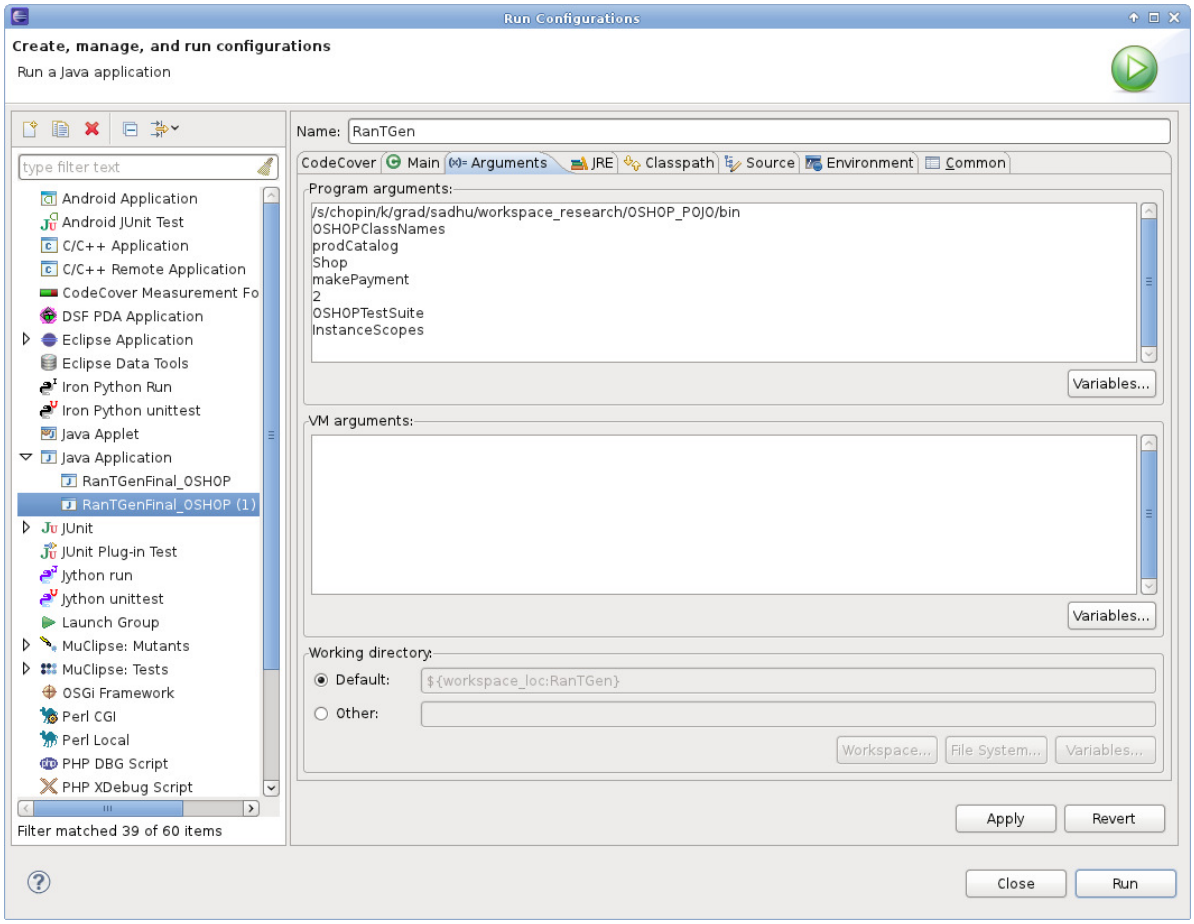


Figure 4.8: Run Configuration for RanTGen Application

If RanTGen is run as an Eclipse plug-in, Figure 4.9 shows the user entry form where the arguments are provided in the corresponding fields. The UI form consists of combo-boxes and browser dialogs that facilitates a user to select the arguments.

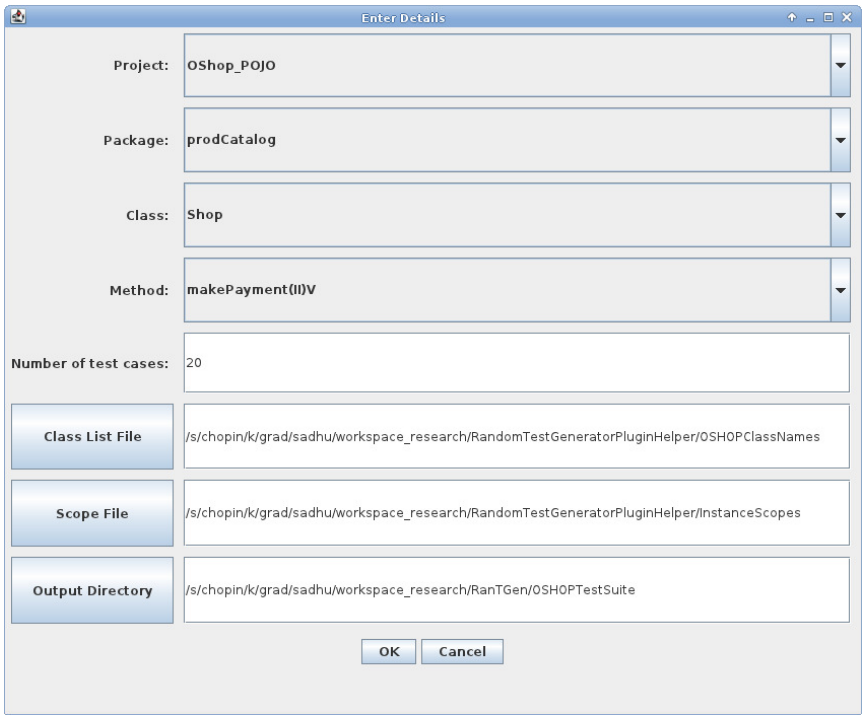


Figure 4.9: Run Configuraion for RanTGen Eclipse Plug-in

In Figure 4.10, we show a sample JUnit test case generated by RanTGen.

```
import junit.framework.TestCase;
import prodCatalog.*;

public class SampleTestCase extends TestCase {
    public void test() {
    /**
    * Instantiate class instances and initialize the attributes
    **/
        Shop ShopTarget = new Shop();

        Category CategoryInstance0 = new Category();
```

```
    CategoryInstance0.setCategoryID("15");
    CategoryInstance0.setCategoryDescription("xyz");

    Customer CustomerInstance0 = new Customer();
    CustomerInstance0.setCustomerID("6");

    Customer CustomerInstance1 = new Customer();
    CustomerInstance1.setCustomerID("29");

    CustomerAccount CustomerAccountInstance0 = new CustomerAccount();

    CustomerAccount CustomerAccountInstance1 = new CustomerAccount();

    Product ProductInstance0 = new Product();
    ProductInstance0.setProductID("43");

    Product ProductInstance1 = new Product();
    ProductInstance1.setProductID("28");

    ProductCatalog ProductCatalogInstance0 = new ProductCatalog();

/**
* Create links between the instances created, if an association
* exists between their corresponding classes
**/
    CustomerInstance0.setCustomerAccount(CustomerAccountInstance0);
    ShopTarget.setProductCatalog(ProductCatalogInstance0);
```

```
    ShopTarget.setCustomer(CustomerInstance0);
    ShopTarget.setCustomerAccount(CustomerAccountInstance1);
    ProductInstance0.setProductCatalog(ProductCatalogInstance0);
    CustomerAccountInstance0.setProduct(ProductInstance1);
    CustomerInstance1.setCustomerAccount(CustomerAccountInstance1);
    ShopTarget.setCustomer(CustomerInstance1);
    ShopTarget.setCustomerAccount(CustomerAccountInstance0);
    ShopTarget.setProductCatalog(ProductCatalogInstance0);
    ProductInstance1.setCategory(CategoryInstance0);
    ShopTarget.setProduct(ProductInstance1);
    ProductInstance0.setCategory(CategoryInstance0);
    Category.setProductCatalog(ProductCatalogInstance0);
    ShopTarget.setProduct(ProductInstance0);

/**
* Invoke method under test
**/
    ShopTarget.makePayment("6",4);

/**
* Insert default NotNull assertions
**/
    Assert.assertNotNull(CategoryInstance0);
    Assert.assertNotNull(CustomerInstance0);
    Assert.assertNotNull(CustomerInstance1);
    Assert.assertNotNull(CustomerAccountInstance0);
    Assert.assertNotNull(CustomerAccountInstance1);
```

```
        Assert.assertNotNull(ProductInstance0);

        Assert.assertNotNull(ProductInstance1);

        Assert.assertNotNull(ProductCatalogInstance0);



    /**
    * Insert default Equals assertions
    **/
        Assert.assertEquals("15", CategoryInstance0.getCategoryID());
        Assert.assertEquals("xyz", CategoryInstance0.
                            getCategoryDescription());
        Assert.assertEquals("6", CustomerInstance0.getCustomerID());
        Assert.assertEquals("29", CustomerInstance1.getCustomerID());
        Assert.assertEquals("43", ProductInstance0.getProductID());
        Assert.assertEquals("28", ProductInstance1.getProductID());
        Assert.assertFalse(CustomerInstance0.getCustomerID().
                           equals(CustomerInstance1.getCustomerID()));
        Assert.assertFalse(ProductInstance0.getProductID().
                           equals(ProductInstance1.getProductID()));

    /**
    * Specific assertions that are inserted automatically
    **/
        Assert.assertEquals("6", CustomerInstance0.getCustomerID());

    /**
    * A tester can also manually write assertions in the test case
    **/
    }
}
```

Figure 4.10: Sample RanTGen Test Case

Chapter 5

Evaluation

The goal of our evaluation is to compare RanTGen and RANDOOP with respect to the following criteria:

1. Mutation score
2. Statement and branch coverage
3. AEM coverage
4. Time taken to generate test cases

The following sections discuss the steps involved in our evaluation. Section 5.1 describes the details of the subject programs that we designed and implemented for this study. Section 5.2 describes how the test cases are generated by the two test generation techniques. In Section 5.3, we describe how the mutants are generated for the subject programs. Section 5.4 presents the results of our evaluation and analyzes them. Section 5.5 lists the threats to validity.

5.1 Subject Programs

We conducted our evaluation on three subject programs: an online shopping system (OSHOP), a model composition system (COMP), and a UML to VAG transformation tool (UML2VAG) [12]. The design class models of these programs were created by

teams of graduate software engineering students at Colorado State University. The implementations of these programs were written in Java.

Table 5.1 shows the details of the subject programs. The number of classes in each program and the total number of associations between the participating classes are listed in the table. For each program, we list a system operation as the method under test.

Table 5.1: Subject Programs and Methods Under Test

Subject Program	Number of classes	Number of associations between classes	Method under test
OSHOP	6	9	`Shop.makePayment(String customerID, double balance)`
COMP	5	6	`Operation.merge(Operation op)`
UML2VAG	17	20	`Visitor.visitCreateMessage (CreateMessage crt, VAGEdge incomingEdge)`

5.2 Test Case Generation Phase

For both RanTGen and RANDOOP, we generated 600 test cases for the three methods under test in the subject programs. For each of the methods, we created 10 test suites, each containing 20 test cases.

5.2.1 RanTGen Tests

To output 20 RanTGen test cases in each test suite, we used different random seeds. To check the post-condition of the methods under test, we wrote specific assertions at the end of the test case. We implemented the `assertCondition` method of the `CheckAssertions` interface to generate program specific assertions in the test case. The default assertions, `assertNotNull` and `assertEquals` are inserted automatically in the test case by RanTGen.

5.2.2 RANDOOP Tests

We customized RANDOOP's test generation process so that the generated test cases are comparable with RanTGen's test cases. We describe the details of the customizations below.

Invoking the method under test: RanTGen test cases call the method under test after creating the configuration. RANDOOP, by default, does not generate test cases targeting a method under test. RANDOOP generates test cases by creating random call sequences, and the method under test may or may not be called at the end of such a sequence. We customized RANDOOP so that it always calls the method under test at the end of a call sequence.

Generating assertions: RANDOOP, by default, does not check for post-conditions, other than trivial ones such as, checking for `null` values or `equality` of objects. We customized RANDOOP so that it inserts assertions that check the post-condition of the method under test.

To restrict RANDOOP to generate 20 test cases in each test suite, we used the option `--outputlimit=20` in the run configuration. Each test suite uses a different seed.

5.3 Mutation Analysis Phase

We used MuJava [13] to generate mutants for each of the three subject programs. We applied the following mutation operators:

1. COI: Conditional Operator Insertion
2. ROR: Relational Operator Replacement
3. AOIS: Arithmetic Operator Insertion (shortcut)
4. AOIU: Arithmetic Operator Insertion (unary)

5. AORB: Arithmetic Operator Replacement (binary)
6. ASRS: Assignment Operator Replacement (shortcut)
7. EMM: Modifier method change
8. JID: Member variable initialization deletion
9. PRV: Reference assignment with other comparable type
10. JTI: *this* keyword insertion
11. JTD: *this* keyword deletion
12. JSI: *static* modifier insertion

We manually identified and removed the equivalent mutants.

5.4 Results and Analysis

Once the test suites for each approach are generated and the mutants for the subject programs are created, we run the test suites on each mutant. In the following subsections, we present and analyze the results of our experiments for each of the four evaluation criteria.

5.4.1 Mutation Score

We measured the fault detection effectiveness of each approach by comparing the mutation scores of their test cases.

Table 5.2 shows the mutation analysis results of RanTGen and RANDOOP test cases for all the programs. If a mutation causes a test to fail, which means the test is detecting the fault, that mutant is said to be "killed" by the test case. The column, "*Total # of mutants killed*", shows the total number of mutants killed by running all the test suites generated using each approach. The columns under "*Avg Mutation*

Table 5.2: Mutation Score

Subject Program	Total # of non-equivalent mutants	Total # of mutants killed		Average Mutation Score	
		RanTGen	RANDOOP	RanTGen	RANDOOP
OSHOP	131	115	92	82.8%	66.4%
COMP	77	62	53	76.2%	67.4%
UML2VAG	187	169	122	87.2%	62.0%

Score" present the average mutation score over all the test suites for each approach. RanTGen test suites achieve a higher mutation score than RANDOOP test suites for all types of mutants in all three programs. All the mutants killed by RANDOOP are also killed by RanTGen test cases. For programs with a larger number of classes (e.g., UML2VAG), RanTGen obtains a higher mutation score than RANDOOP because RanTGen uses a per-class bound. A RanTGen test case must instantiate each class a certain number of times to satisfy the lower bound constraint. RANDOOP, on the other hand, does not use a per-class bound. It randomly calls constructors and methods of the program under test. Therefore, in programs containing many classes, RANDOOP test cases did not instantiate some classes and were not able to kill the mutants in the classes that were not instantiated.

Table 5.3 shows the mutant killing effectiveness of both the test generation approaches for each mutation operator. The results show that RANDOOP never killed more mutants than RanTGen. In fact, for most of the mutation operators, the number of mutants killed by RanTGen is higher compared to RANDOOP.

For the class mutant operators, RanTGen killed more mutants generated by some operators (*EMM* and *JSI*) than RANDOOP. This happened because RanTGen ensures creation and initialization of at least one instance of each participating class of the subject program. To kill a *JSI* mutant, the test case must contain more than

Table 5.3: Mutation Results for Each Mutation Operator

Mutation Operators	OSHOP			COMP			UML2VAG		
	Total # of mutants	Number of mutants killed by		Total # of mutants	Number of mutants killed by		Total # of mutants	Number of mutants killed by	
		RanT-Gen	RAND-OOP		RanT-Gen	RAND-OOP		RanT-Gen	RAND-OOP
COI	16	14	11	12	9	8	25	22	17
ROR	18	16	12	7	6	5	28	25	19
AOIS	32	26	22	11	8	7	35	30	23
AOIU	15	13	10	13	11	10	19	17	14
AORB	16	16	13	9	7	6	16	15	12
ASRS	4	4	4	3	3	3	12	11	9
EMM	6	6	3	3	3	2	9	9	5
JID	6	6	5	2	2	2	9	8	5
PRV	3	3	3	4	3	3	7	7	5
JTI	1	1	1	1	1	1	3	3	2
JTD	1	1	1	1	1	1	5	5	4
JSI	13	9	5	11	9	5	19	15	7

one instance of the mutated class. Due to the use of a per-class bound, a RanTGen test case is more likely to create multiple instances of each class. RANDOOP test cases, which were generated without a per-class bound, did not create multiple instances of a class in many cases, and thus, failed to kill the *JSI* mutants. The *EMM* operator mutates the modifier methods in the code, such as the `set` methods. For a test case to kill a *EMM* mutant, it should call the modifier method. As discussed in Section 4.1.3, each RanTGen test case initializes the instantiated class attributes and creates links between class instances by calling the corresponding `set` methods. Thus, RanTGen test cases killed most of the *EMM* mutants created in the subject programs. RANDOOP, however, does not ensure invocation to these `set` methods in its test cases, and thus, fails to kill these mutants.

For the traditional operators, RanTGen killed more mutants generated by the arithmetic, conditional and relational operators (such as, *AORB*, *COI* and *ROR*),

than RANDOOP. This is because in our subject programs, these operators are often present in a method that needs to be checked in the post-conditions of the method under test. In a test case, we check the post-conditions of the method under test. The post-condition may not always be in the same target class which contains the method under test. If a RANDOOP test case does not create an instance of the class containing the method that needs to be checked in the post-condition, the post-condition cannot be checked; and thus, the mutant in that method is not killed. For example, the post-condition for the method under test `Shop.makePayment` is to update the balance in the `CustomerAccount`. In the method `CustomerAccount.updateBalance(int paymentAmount)`, the original code was

`balance = balance - paymentAmount`.

A mutant created by *ROR* was

`balance += balance - paymentAmount`,

while an *AOR* mutant was

`balance = balance + paymentAmount`.

Now, if RANDOOP does not create an instance of `CustomerAccount`, it is not possible to invoke the method `CustomerAccount.updateBalance(int paymentAmount)`. Thus, the above mutants cannot be killed by any RANDOOP test case. RanTGen, on the other hand, uses the per-class bound and instantiates each class in its test cases. Thus, even if a method is not in the class containing the method under test, it can be accessed and checked. Thus, given an instance of the `CustomerAccount`, a RanTGen test case can call the `updateBalance` method, and thus, has a higher likelihood to kill the mutants.

We observed that both RanTGen and RANDOOP test cases did not kill many AOIS mutants. Insertion of arithmetic operators did not alter some functionality of the subject programs, and the test cases were not able to kill those mutants. For

example, for the method `setCustomerID`, the original code was

`this.customerID = custID`.

A mutant created was

`this.customerID = custID++`.

According to the program, each customer should get a unique `customerID`. Even though the mutant increments the `customerID` by 1, it still assigns a unique `customerID` to each `Customer`. Moreover, none of the test cases required searching based on the original `customerId`. Thus, the mutant is not killed by any test case.

5.4.2 Statement and Branch Coverage

We used the *CodeCover*[1] tool to measure the statement and branch coverage. Figure 5.1 shows the average statement coverage results by RanTGen and RANDOOP test cases for all three programs. RanTGen tests, on average, achieved a higher statement coverage than RANDOOP. Owing to the per-class bounds, RanTGen test cases create instances of all participating classes, initialize the variables of those instances, and create the links between them. This ensures coverage of the constructors and the set methods of all classes in the subject programs. RANDOOP, on the other hand, cannot ensure coverage of certain parts of the code.

---

[1] http://codecover.org/

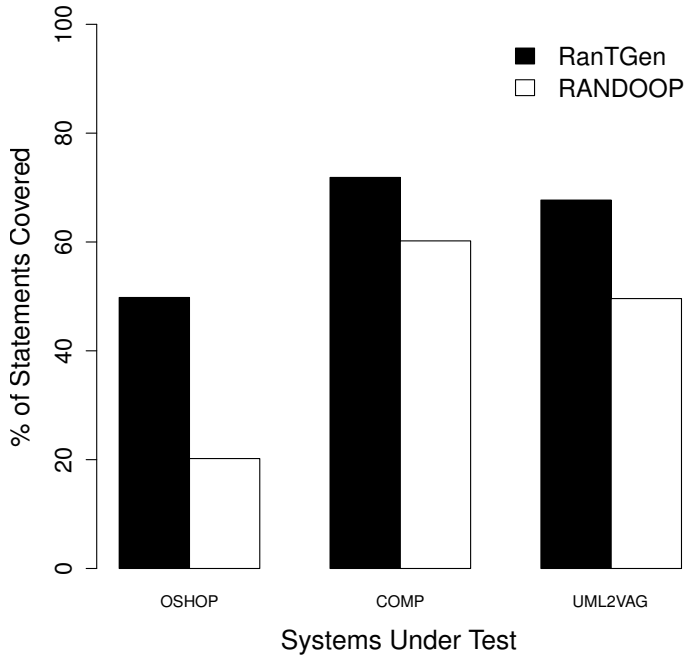


Figure 5.1: Statement Coverage Results of RanTGen and RANDOOP Tests

In Figure 5.2, we observe that the average branch coverage obtained by the RanTGen test cases is higher than that of RANDOOP test cases for all the three programs. This happens because RANDOOP rejects a test case that is invalid and does not extend it any further. Thus, even though a test case would have covered a branch, it never gets extended along that branch if the test case is invalid. RanTGen does not reject any test case and thus, the test cases achieve higher branch coverage.

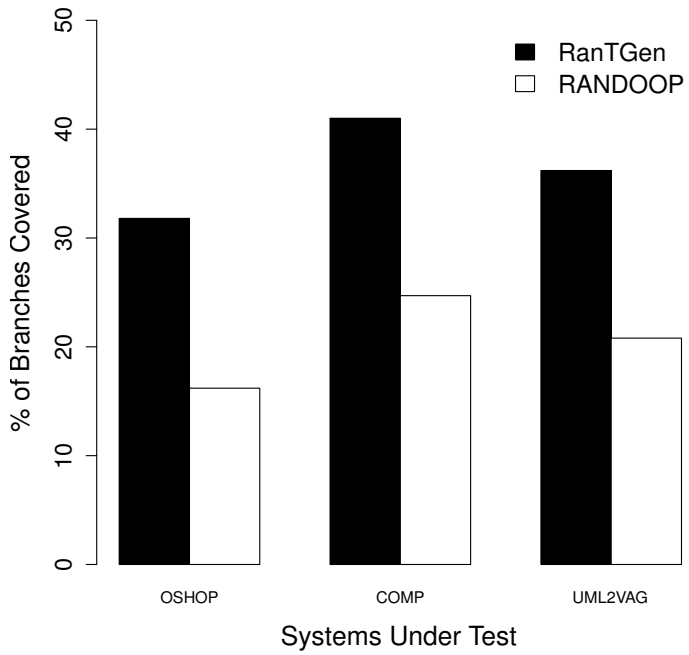


Figure 5.2: Branch Coverage Results of RanTGen and RANDOOP Tests

5.4.3 AEM Domain Coverage

For the AEM criterion [5], for each subject program, we listed all possible association-end multiplicities between the participating objects in the program. We do not have a tool for measuring AEM coverage. That is why once the test cases for both approaches were generated, we had to manually inspect the test cases and identify the types of association-end multiplicities created in the configurations. In Figure 5.3 we present the average AEM domain coverage by RANDOOP and RanTGen test cases. For each subject program, RanTGen test cases achieve higher coverage in the AEM domain than RANDOOP. The results support what we hypothesized in Section 1.2.

Since RANDOOP rejects invalid configurations, it might ignore the generation of certain valid configurations and thus, not cover a part of this domain. In addition, because RANDOOP test cases are feedback dependent, we observe that successive test

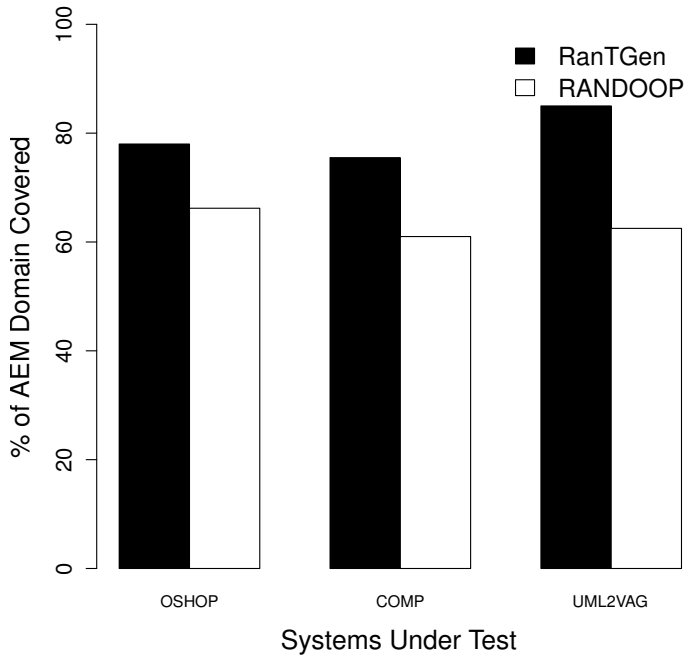


Figure 5.3: AEM Domain Coverage Results of RanTGen and RANDOOP Tests

cases create configurations similar to the prior test cases. In our approach, neither we reject a configuration, nor we build new configurations depending on previously generated ones. Hence, RanTGen generates more varied configurations than RANDOOP.

5.4.4 Time Required for Test Case Generation

Table 5.4 shows the time taken to generate test cases using RanTGen and RANDOOP. We measured the time taken (in milliseconds) to generate the test cases of all the test suites by each approach. We used a 64-bit Linux machine running Fedora15 with 8GB RAM and configuration HP-Z200-Xeon3450.

RanTGen generated the test cases in less time than RANDOOP. RANDOOP generated a huge number of configurations and then removed a majority of them because they were invalid. In order to output 20 test cases, RANDOOP, on an

Table 5.4: Time of Test Case Generation

Subject Program	Avg time to generate 20 tests (in ms)		Time per test case (in ms)	
	RanTGen	RANDOOP	RanTGen	RANDOOP
OSHOP	938	100293	47	5015
COMP	553	100180	28	5009
UML2VAG	1218	101312	61	5065

average, generated 5378 configurations and rejected 5358 of them. On the other hand, for 20 test cases, RanTGen generated 20 configurations, and if any of them was invalid, RanTGen fixed it and turned it into a valid test case.

Analyzing our evaluation results, we found that our approach generated more effective test cases in less time compared to RANDOOP.

5.5 Threats to Validity

Threats to external validity questions whether the findings of our study can be generalized for other applications. An external threat to validity is that our study was conducted on small applications. Further studies need to be performed on large-scale applications to compare the scalability of the approaches and to generalize the results. However, we performed our evaluation on subject programs that implemented different types of associations, such as one-to-one associations, one-to-many associations, uni-directional as well as bi-directional associations, and composite associations.

We conducted our experiments on different domains of applications to evaluate the effectiveness of our approach. OSHOP is an online shopping application, whereas COMP is a model composition system that relies on signature matching. UML2VAG, on the other hand, is a test input generation system that uses control flow graphs (VAG) to test UML design models. However, we used only one application for each domain. Further studies need to be performed on more applications for each such

domain as well as on applications for other domains in order to infer any domain-specific performance of our approach.

Another external threat to validity is that each of our subject programs have one system operation. In an application containing many utility methods, RANDOOP test cases may achieve higher statement coverage than RanTGen because RANDOOP gives equal priority to all the methods in an application. It may call different methods in a test case. On the other hand, a RanTGen test case is restricted to call only the following methods: (1) the constructors of the classes, (2) the setters that create links between the objects, and (3) the method under test.

We compared the fault detection effectiveness of the test generation approaches by measuring their mutation scores. These mutants are seeded faults generated on purpose, by MuJava. An external threat to validity is that we did not run the test cases on real-life projects and detect unknown faults.

Threats to construct validity questions whether the measures used actually represent the intent of the study. We used the AEM criterion to measure the diversity of the object configurations. Since we are concerned about the multiplicity constraints specified in the UML class model of a program, we chose the association-end multiplicity criterion to measure test adequacy of the test cases generated by each test generation technique. However, this might not be the best way to measure the diversity of the configuration population. Another alternate measure that can be used is the distance measurements between the object configurations. Ciupa et al. [9] measured *object distance* between random test inputs in order to have a distribution of inputs spread out evenly over the input domain.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We proposed a test input generation approach that creates test cases containing random object configurations. For a test case to be valid, it should be consistent with the constraints given in the class model of the program under test. In our test generation approach, we are concerned about the multiplicity constraints specified in the class model. We make sure a test case generated by our approach produces an object configuration that is consistent with the class model's multiplicity specifications. The two principles guiding our approach are:

- We do not take feedback from previously generated test cases, and hence our test cases are independent of any bias. Each test case in our approach is generated from scratch with new random configurations.
- We do not reject any test case. If an invalid object configuration is generated, we attempt to fix them within a stipulated amount of time.

Since we do not discard any test case, and each test case is independent of any feedback from prior test cases, we proposed that our approach will generate more diverse configurations, compared to an existing random feedback-directed test generation approach, RANDOOP.

We implemented a prototype tool, RanTGen, to generate JUnit test cases for the program under test. We add the multiplicity specifications in the class model of the program and provide the class model as an input to our approach. Once a test case is generated, RanTGen checks whether the test case object configuration is consistent with the class model's multiplicity constraints. If so, RanTGen outputs the test case as a valid one. If the configuration is inconsistent, RanTGen identifies the violation, fixes it and checks again for further discrepancies. This checking continues until a valid configuration is produced. RanTGen then outputs the corresponding test case. The type of fixes that we handled are of the following types:

(a) *removeLink*, when there are redundant links between two class instances

(b) *addLink*, when there are insufficient links between two class instances

(c) *removeObject*, when objects exist without being linked to other objects

We evaluated our approach with respect to RANDOOP. Our results show that our approach is more effective than RANDOOP, and the time taken to generate test cases by our approach is less compared to RANDOOP. We used the *association-end multiplicity* (AEM) test adequacy criterion to define the domain of possible object configurations. We measured the coverage of the configuration domain for both test generation approaches. RanTGen test cases achieve a higher AEM coverage than RANDOOP test casess implying that our approach generated more varied object configurations compared to RANDOOP for the systems that we studied.

6.2 Future Work

The case studies reported in this paper are the beginning of a large scale validation activity to investigate the effectiveness of our approach. We plan to perform further evaluation on larger systems as well as on different domains to generalize our conclusions.

In our current implementation, we address violations one by one. It is possible that addressing one violation may simultaneously address some other violations. We plan to prioritize the violations such that fixing the violation with the highest priority addresses a large number of other violations. This would save the time required to generate a valid configuration. It is also possible that the fix chosen to address a particular violation may cause a larger number of subsequent modifications than another fix. Addressing this issue essentially leads to a search problem where there are two objectives, reduce the number of fixes but also create a diverse set of configurations, where the distance between two configurations is maximized. In our future work, we plan to investigate the ways a configuration can be fixed and optimize the number of fixes needed to create a valid configuration.

In our current implementation, we focus only on multiplicity specification, more specifically on the bounds of the multiplicity constraints. There are other constraints related to multiplicity specification, such as ordering of values and uniqueness of values. In addition, a UML class model of a program consists of several other constraints, such as OCL constraints and invariants. A possible extension of our work can address these constraints while checking the consistency of an object configuration.

REFERENCES

[1] Eclipse Java development tools. `http://eclipse.org/jdt/`.

[2] Eclipse modeling framework project. `http://www.eclipse.org/modeling/emf/`.

[3] Eclipse PDE. `http://www.eclipse.org/pde/`.

[4] Ecore tools. `http://wiki.eclipse.org/index.php/Ecore_Tools/`.

[5] Anneliese A. Andrews, Robert B. France, Sudipto Ghosh, and Gerald Craig. Test adequacy criteria for UML design models. volume 13, pages 95–127, April-June 2003.

[6] Aritra Bandyopadhyay and Sudipto Ghosh. Test input generation using UML sequence and state machines models. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, Denver, Colorado, USA, 2009.

[7] Benoit Baudry. Testing model transformations: A case for test generation from input domain models. In *Model Driven Engineering for Distributed Real-time Embedded Systems*. 2009.

[8] Tsong Yueh Chen. Adaptive random testing. In *Quality Software International Conference*, page 443, Jeju, Korea, 2008.

[9] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Object distance and its application to adaptive random testing of object-oriented programs. In *Proceedings of the 1st International Workshop on Random testing*, pages 55–63, Portland, ME, USA, 2006.

[10] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: Adaptive random testing for object-oriented software. In *Proceedings of the 30th international conference on Software engineering*, pages 71–80, Leipzig, Germany, 2008.

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

[12] Trung T. Dinh-Trong, Sudipto Ghosh, and R. B. France. A systematic approach to generate inputs to test UML design models. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 95–104, Raleigh, North Carolina, USA, 2006.

[13] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: an automated class mutation system: Research articles. *Software Testing Verification and Reliability*, 15(2), pages 97–133, 2005.

[14] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. Parallel test generation and execution with Korat. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 135–144, Dubrovnik, Croatia, 2007.

[15] Catherine Oriat. Jartege: a tool for random generation of unit tests for java classes. In *2nd International Workshop on Software Quality (SOQUA 2005)*, pages 242-256, Erfurt, Germany, 2005.

[16] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *European Conference on Object-Oriented Programming*, pages 504–527, Glassgow, UK, 2005.

[17] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, 2007.

[18] Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, Atlanta, Georgia, pages 571–572, USA, 2007.

[19] Koushik Sen and Gul Agha. Cute and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, pages 419–423, Seattle, WA, USA, 2006.

[20] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference*, pages 263–272, New York, NY, USA, 2005.

[21] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic model generation strategies for model transformation testing. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, pages 148–164, Zurich, Switzerland, 2009.

[22] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for Java containers using state matching. In *Proceedings of the 2006 International Symposium on Software testing and analysis*, pages 37–48, New York, NY, USA, 2006.

[23] Yi Wei, Serge Gebhardt, Manual Oriol, and Bertrand Meyer. Satisfying test preconditions through guided object selection. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation*, Paris, France, April 2010.

[24] Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proceedings 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Linz, Austria, 2004.

[25] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, Edinburgh, UK, 2005.