Incorporating Design Knowledge into Genetic Algorithm-based White-Box Software Test Case Generators

Matthew C. Makai

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Science and Applications

April 24, 2008
Falls Church, VA

Keywords: Software Testing, Genetic Algorithms, Evolutionary Computation

Incorporating Design Knowledge into Genetic Algorithm-based White-Box Software Test Case Generators

Matthew C. Makai

ABSTRACT

This thesis shows how to incorporate Unified Modeling Language sequence diagrams into genetic algorithm-based automated test case generators to increase the code coverage of their resulting test cases. Automated generation of test data through evolutionary testing was proven feasible in prior research studies. In those previous investigations, the metrics used for determining the test generation method effectiveness were the percentages of testing statement and branch code coverage achieved. However, the code coverage realized in those preceding studies often converged at suboptimal percentages due to a lack of guidance in conditional statements. This study compares the coverage percentages of 16 different Java programs when test cases are automatically generated with and without incorporating associated UML sequence diagrams. It introduces a tool known as the Evolutionary Test Case Generator, or ETCG, an automatic test case generator based on genetic algorithms that provides the ability to incorporate sequence diagrams to direct the heuristic search process and facilitate evolutionary testing. When the generator uses sequence diagrams, the resulting test cases showed an average improvement of 21% in branch coverage and 8% in statement coverage over test cases produced without using sequence diagrams.

Table of Contents

Table of Figures

Tables

1. Introduction

Software testing detects errors unintentionally introduced to programs during design and development [1]. Creating test cases manually, traditionally performed by human testers, is time-consuming and adds additional overhead to the budget of all nontrivial software development projects. Although testing is a resource intensive process, it is also invaluable for validating the correctness of applications. In addition, once a suite of test cases is generated for a project, those tests can be run continuously as part of a software project change management process [3]. To date, the actual generation of test cases through automated measures has been explored in research studies but limited in software engineering practices [5].

The problem domain of this study is evolutionary testing, also commonly known as heuristic search-based test data generation. Automated test data generation dates back to the 1980s, when random testing was first explored as a mechanism for producing inputs to assess the execution of software [5]. In the early 1990s, Korel applied genetic algorithms, a well-known type of heuristic search-based method, to the test data generation process to analyze if improvements over random testing could be made [6]. The results of the study indicated that heuristic search-based test data generation was superior to random testing.

Dozens of studies have identified ways to improve the effectiveness of heuristic search-based test data generation since Korel's original paper. These improvements, as well as other test data methods not based on heuristics, are covered under the related work section. Korel's work inspired further research in the application of genetic algorithms to software testing [5]. Eventually, studies in software testing using genetic algorithms were merged under the field of *evolutionary testing*. Evolutionary testing explored not only genetic algorithms, but other heuristic search-based processes as well. This study is an extension of work in genetic algorithms but it potentially could be applied to other heuristic processes as well.

Heuristic search-based test data generation is heuristic because it is stochastic instead of deterministic. Therefore, chance plays a role in the results received from running genetic algorithms and related heuristic methods. Heuristic methods are required in situations where the number of possible inputs is inexhaustibly large, for example the Traveling Salesman Problem. Generating test data for software also involves an intractable number of inputs unless heuristics are used. In manual software testing, human testers pick the inputs based on intuition or a test plan, which essentially plays the role of the heuristic. Automated test data generation is also described as search-based because it involves the process of finding values that are most suitable as inputs.

1.1 Observations

According to previous studies in genetic algorithms, using data structures and domain knowledge during the population evaluation process can restrict the search space, provide guidance to the search mechanism, or both [7]. By limiting the number of possible solutions to a problem or providing increased direction to the search process the search process becomes more efficient. By applying the same results to genetic algorithms used in evolutionary testing, it should be possible to produce test data more effectively, when measured in terms of the rates of testing code coverage percentage in equivalent elapsed CPU execution time.

1.2 Application Domain

The research in this study is targeted at the application domain of evolutionary testing. In particular, genetic algorithms were used for the evolutionary algorithms method. Genetic algorithms were used for the implementation because they are the most common technique used in prior studies.

The test cases created by the generator with evolutionary testing are targeted at testing non-real-time software systems. Mission-critical systems with stringent requirements for 100% code coverage testing are excluded from the application domain of this research. The test cases created by heuristic search-based methods aim for a reasonable code coverage percentage of seventy to eighty percent as opposed to complete coverage of the code. In addition, environmental variables, such as the state the machine is in, are not exercises by the generated test cases. The test cases are meant for executing a target program with inputs that test a majority of the source code as measured by branch or statement coverage and receiving output without regards to environmental variables.

In this study, test generation is applied to Java because the programming language is widely used in both commercial and academic settings. However, it is possible to generalize this research to all evolutionary testing studies, regardless of the chosen programming language. This generalization assumption is justified because in previous studies, it was shown that evolutionary testing results are applicable to languages other than Java [13, 14].

1.3 Problem Statement

The advantages of evolutionary testing are nullified when there is no guidance to the search. When no guidance is provided, the search process deteriorates into a random search, which has previously been proven ineffective on anything more than trivial programs [5, 11]. In particular, this problem becomes acute during the test data generation process when conditional statements rely on variables that are only affected in otherwise unrelated procedures. In that case, testing code coverage percentages suffer and are therefore suboptimal. Therefore, the problem statement in this study is that evolutionary testing methods currently converge to below optimal and achieved rates of testing code coverage percentages due to a lack of search guidance in source code conditional statements.

1.4 Motivation

Software testing encompasses a sizeable fraction of time required for software development efforts. In 2005, cost estimates for handling errors during the testing phase of the software development lifecycle instead of during earlier phases were pegged at forty billion dollars [12]. In addition to cost savings, if routine unit, functional, and integration test case creation is partially or completely automated, it grants several advantages over current testing practices:

1. Fewer person hours spent creating routine test cases, thereby reducing testing costs.
2. Increased software quality when human testers design creative test cases[1] for more rigorous software testing.
3. Additional time for software engineers to write source code for implementations.

The problem of software testing is important because it adds to the costs of creating software.

1.5 Hypothesis

Based on the observations and problem statement, the tool Evolutionary Test Case Generator (ETCG) is expected to test the following variables and hypothesis. The independent variable is the whether or not design knowledge, which is specified in the form of sequence diagrams, is incorporated into the test case generator's test case generation process. The dependent variable is the code coverage percentage achieved during execution, measured by either branch coverage or statement coverage.

The formal hypothesis is:

Incorporating a sequence diagram into the test generation process will increase branch coverage and statement coverage over an equivalent test generator that does not use sequence diagrams.

The hypothesis function is:

c = f(k), where c is the increase in branch or statement coverage, and k is a binary variable, either true or false, that corresponds to whether or not sequence diagrams are incorporated into the test case generation process.

The rationale for the hypothesis comes from previous work in genetic algorithms that proves the iterative search process is most effective when provided with guidance from domain knowledge [7]. In this study, the domain knowledge is provided by a common design phase artifact, the sequence diagram.

1.6 Design Knowledge and Sequence Diagrams Relationship

Design knowledge is provided by artifacts from the design phase of the software lifecycle. In this study, sequence diagrams represent design knowledge in because they describe the interactions between objects and their corresponding methods in situations that are important to the operation of the overall system. The designer and developers agree during the design phase as to how the system should operate and this knowledge can be used to properly test the system based on their assumptions. Therefore, the sequence diagrams are a concrete conception of a given software system and represent knowledge for how the system is designed.

[1] Creative test cases are defined as those that require human intervention to create, such as evaluating aesthetic requirements.

1.7 Goals

There are two primary goals for what should be accomplished by this research. The first goal is to determine whether or not using design knowledge enhances the effectiveness of automated test case generators. The hypothesis describes the measurement of the effectiveness by the resulting code coverage percentage. The second goal is to implement a tool that demonstrates the code coverage percentage achieved when using design knowledge versus without the aid of the design knowledge.

1.8 Contributions

This paper makes two primary contributions to the field of automated test case generation. First, a novel approach towards automatically integrating sequence diagrams into the genetic algorithm’s evaluation mechanism is introduced. Second, a technique known as object mocking is used in generating test cases. There are no previous studies that use mock objects, which are a well-known mechanism for creating effective unit tests.

Therefore, this study presents two main contributions to the research area of evolutionary testing:

1. Incorporating design knowledge increases the code coverage achievable by automated test generation methods.
2. This is the first study to leverage mock objects to isolate specific methods under test, which accurately mimics recommended software unit testing practices.

2. Problem Background

Existing heuristic test data generators have low potential for creating test data when otherwise separate sections of code are dependent upon each other in limited but important ways. For example, programs often contain conditional statements with dozens of potential outcomes but no other guidance towards one that is most important for successfully executing the program. In that case, the probability of previous test data generators successfully executing the most critical branches of the conditional are no better than what a random test data generator could achieve. Section 2.1 presents a tangible example of this problem. Section 2.2 describes the problem statement in more detail. Section 2.3 reviews previously addressed challenges in evolutionary testing that appear similar to the problem statement, but upon further inspection are actually different.

2.1 Example

A simple example highlights the solution to the problem stated above where two otherwise unrelated areas of code affect the execution of a critical condition statement. In Java, a *bean* is an object that stores data in variables and allows manipulation of those variables only through getter and setter methods. If a class relies on a bean to store data, it will often refer to that object during conditional statements to determine if criteria are met with regards to the state of the variables in the bean. The object will obtain the value of a specific variable through its corresponding getter method. However, if the variable is set by another object during execution and not by the object with the conditional statement, generated test cases may never execute that branch of code because it will ignore the setter method. Even if a test case generator knows in advance to look for setter methods, the problem remains with more complex examples of methods that do not follow the getter and setter pattern.

This study determines whether or not design knowledge, provided in the form of sequence diagrams, can increase the testing code coverage percentages by executing branch statements where this problem occurs. Section 4 contains further information on the details of the solution approach.

2.2 Problem Statement

As described in the introduction, the problem statement refers to situations where testing code coverage is suboptimal due to conditional statements dependent upon variables affected in unrelated subroutines. Section 2.1 reviewed an example of the problem statement but it is worthwhile to review it in further detail.

Conditional statements often cause difficulties during the test data generation process in evolutionary testing [5]. The problem was first discovered during random testing experiments, even before evolutionary testing was attempted. One of the challenges with conditional statements is that they are often only satisfied with a small subset of the overall set of inputs to the conditional. With random testing, there is no guidance in the search. Conditional statements with large input sets and outputs that are obtained only for a small range of inputs often cause the search for satisfactory inputs that exercise all unique outputs to fail. The purpose of using evolutionary testing over random testing is the addition of guidance to the search. However, in certain cases, as described in the problem statement, that guidance may not be enough to find a satisfactory value for the conditional.

2.3 Related Problems and Solutions

Several problems previously addressed in evolutionary testing appear similar to the problem of increasing testing code coverage but each contains important differences. As described below, the solutions provided for each of these problems do not address the problem statement for this study. Further analysis of studies is provided in section 3 under related work in evolutionary testing.

2.3.1 Conditional Statements and Testability Transformations

Testability transformations are a "source-to-source transformation that aims to improve… a given test generation method to generate test data for the original program [8]." The goal of testability transformations to increase testing code coverage percentage is the same as this study. However, the problem that testability transformations address is different. The underlying problem that is addressed is that source code can contain conditional statements with flag variables [9]. Testability transformations are an attempt to remedy this problem by transforming the original source code into a version that can be tested that contains greater guidance for the search process. This is different from this research in that the flag variables deal with primitives that can be replaced by substitution with the original assignment statement.

Testability transformations and design knowledge are complementary to each other. Both approaches can be used in conjunction with each other. In fact, the generator module that will be explained in section 4 contains a subcomponent for applying testability transformations to the original source code before attempting to generate test cases.

2.3.2 The Species-Per-Path Approach

The Species-Per-Path Approach to search-based test data generation was first outlined in a paper of the same name in 2006 [10]. The motivation behind this paper is that heuristic search methods cannot efficiently generate test data for certain nested conditional code structures. If that problem is solved, the authors' claims that heuristic search-based testing methods will become practical methods for automatically generating test data with full branch coverage. The authors' goal again is to increase the test code coverage percentage achievable by evolutionary testing. The new approach presented in this study is to divide the program into slices that can be executed in parallel by the test data generator. The generation process can then be split among separate subpopulations. The Species-Per-Path (SPP) approach considers these subpopulations to be different "species," and creates one for each path in the source code that requires test cases. By splitting each path into a separate search problem, a new fitness function can be derived for each species. The specialized fitness functions provide enhanced guidance to each species. In addition, it is trivial to parallelize the disparate subpopulations since, at least in this study, they do not share data about the search spaces. The simple areas of the source code are quickly covered by the populations and the more difficult regions can be isolated for further test generation. The solution also utilizes testability transformations, although the authors note that it is not required for the SPP approach to work properly.

The Species-Per-Path approach study is different from the work presented here because SPP approach describes how to generate test cases for programs that are broken up into smaller slices, but not how to more effectively handle conditional statements. In this study, the program would still have an issue with areas of source code that impact common variables and otherwise interact only in a conditional statement. However, once again the Species-Per-Path approach could be used in conjunction with design knowledge to more effectively handle conditional statements within each subpopulation.

3. Related Work

Previous research in the areas of genetic algorithms and evolutionary testing provide the background for this study. Subsection 3.1 analyzes general research into stochastic search and genetic algorithms, which are both optimization methods and also often considered in studies on artificial intelligence. Section 3.2 presents related work in the field of evolutionary testing is covered to show the development of the field from its origins in the early 1990s to its present state.

3.1 Stochastic Search and Genetic Algorithms

Many stochastic search methods, such as genetic algorithms, are similar to local beam search. Local beam search is an adaptation of beam search that keeps n search states in memory (as opposed to a single state in beam search). During the search process, if the n search states do not contain the target goal, local beam search generates all successors to the n states and selects the most promising states for the next round of the search based on a utility function. An important property of local beam search is that search information is transferred between the states during each generation. If one state creates better successors than the others, the other states will effectively give up their poor searches to explore the more promising successors.

However, local beam search is hindered because it often concentrates on small sections of the search space. This usually leads the search to find only sub-optimal solutions to the problem at hand. *Stochastic beam search* attempts to avoid the pitfalls of local beam search by instead randomly selecting successor search states, with some bias towards successors with higher scores from the utility function. Therefore, in local beam search, the selection mechanism is not swayed by random probability as it is in stochastic beam search.

A variant of stochastic beam search is genetic algorithms. The primary difference between the standard stochastic beam search and GAs is that in stochastic beam search modifying a single parent search state creates the successor states. In contrast, genetic algorithms generate successor states by combining two or more parent search states together [10]. This process of creating new search state successors in genetic algorithms is known as reproduction and is akin to sexual reproduction, whereas stochastic beam search is an asexual method. Genetic algorithm definitions are steeped in terminology based on natural selection [16].

To initiate the genetic algorithm process, a *population* of algorithms composed of *candidate solutions*, also known as *individuals,* must be created. Each individual in the population is composed of an encoding, which is a representation of how the algorithm functions. The encoding can be a floating-point number or more commonly a bit string, where each binary value corresponds to a trait of the individual. In a simple form of genetic algorithm development, the bits in the bit string are independent. However, in more complicated representations, the comparison with natural organisms' DNA is more appropriate. In this scenario, bits can represent traits and are dependent on each other, so that altering the value of a single bit can modify the effects of other bits. Regardless of whether or not dependencies are simulated, both the bit string and the floating-point form represent a single individual in a larger population of other individuals [5, 17].

Once the initial population is represented, a *fitness function* (also known as an *evaluation function*) must be used to evaluate each individual's ability to solve the problem in question. This process is known as *selection* and it is modeled after the idea of natural selection in Darwinian evolution [18]. Most GA evaluate each candidate solution with the fitness function and assign a *performance index* to the individual based on how well it performs [16]. After all candidate solutions are assessed by the fitness function, the selection mechanism chooses which solutions will pass on their encoding to the next generation of the population. The individuals that perform the best will have a higher percentage chance of passing on their encoding based on the performance index assigned during the evaluation period. In this way, candidate solutions that are not the best according to the fitness function still have a possibility, albeit a smaller chance, to pass on their encoding. This prevents the GA from converging and then becoming stuck on *local maxima*, which are not the globally optimal solution for the problem.

Once the candidate solutions that will pass on their encoding are chosen, the process known as *crossover* begins. Multiple forms of crossover exist, but in essence it provides a way for two candidate solutions to share their encoding to produce a new, complete candidate solution. This new individual contains (hopefully) the best attributes of parent candidate solutions. To keep the population level consistent, some candidate solutions may be combined several times to pass on their encoding to multiple individuals. After the crossover process creates the successive population of candidate solutions, *mutations* are introduced to facilitate further diversity. Mutations are often simply the flipping of a random bit from 0 to 1 or vice versa. Most genetic algorithm studies also keep the mutation rate low, at roughly 0.1%, to allow slight variations to occur but prevent radical shifts in the population that disrupt finding optimal solutions [19].

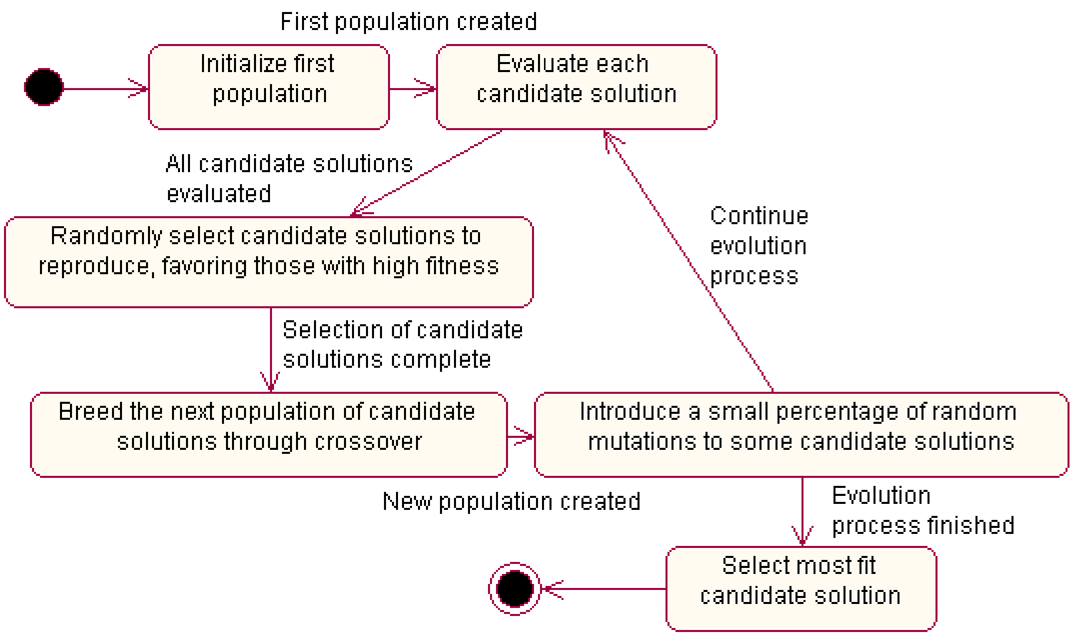

First population created
Initialize first population
Evaluate each candidate solution
All candidate solutions evaluated
Randomly select candidate solutions to reproduce, favoring those with high fitness
Continue evolution process
Selection of candidate solutions complete
Breed the next population of candidate solutions through crossover
Introduce a small percentage of random mutations to some candidate solutions
New population created
Evolution process finished
Select most fit candidate solution

Figure 1: Genetic Algorithm Process

After the crossover and mutation process occurs, the new population of individuals is ready to be evaluated by the fitness function. The process concludes when a predefined limit is surpassed, such as the amount of computation time elapsed or number of population iterations. The activity diagram in figure 1 shows the steps involved in the genetic algorithm process.

3.2 Evolutionary Testing

The research presented in the following sections is based on prior studies performed in the area of heuristic test data generation. As classified by McMinn, there are four cardinal areas where heuristic search-based methods have been applied to automate software testing [5]:

1. White-box testing specific program structures
2. Execution of a specific program feature against a specification
3. Grey-box testing to disprove the validity of software features such as assertions about the safety of a software product
4. Validating non-functional requirements, for example, discovering worst-case execution time

Within these four research areas, the first subject involves white-box structural testing and to date it has produced the most literature in evolutionary testing. White-box structural testing is covered in the next subsection, 3.3.1. Analysis of execution of a specific program feature against a specification, grey-box testing, and validating non-functional requirements are presented in subsections 3.3.2, 3.3.3, and 3.3.4, respectively.

3.2.1 White-box Structural Testing

White-box structural testing is the subject area for this study. Numerous previous studies have been performed in this area. This section will cover only the most influential research accomplishments. Readers interested in further review of this topic should refer to [5].

3.2.1.1 Automated Software Test Generation for Complex Programs

Initial studies in white-box structural testing were applied to trivial programs. In 2003, the authors of [11] wrote that the generation process should also be applied to more complicated programs. The authors' motivation in this study is that they believe automated test case generation must be applied to programs in all types of languages, not only to programs written in simplified sub-languages, for it to be considered successfully applied to complex programs. The authors claim that standard approaches have two cardinal problems: they constrain the language a program is written in, and the function minimization methods are overly simplistic. The solution approach presented by the authors is a tool known as GADGET, which is a test generation system that can use gradient descent, simulated annealing, genetic algorithms (both standard and differential), and random approaches to attempt standard testing metrics such as branch and statement coverage. The system uses coverage tables to determine what conditions have been met and what remain as unexplored. The system compares each of the methods and shows that random testing holds up poorly as the complexity of code increases. The simulated annealing, genetic algorithms, and gradient descent approaches fare much better, leading the authors to conclude that any practical application of automated software testing will require the heuristic search techniques and cannot rely upon random testing.

The study generalizes some conclusions which appear in later evolutionary testing studies. In addition, the authors note that the search space for the heuristic search methods contains many plateaus, which is what embedding design knowledge can overcome.

3.2.1.2 High Volume Software Testing using Genetic Algorithms

Another area of white-box structural testing is the mutation of existing test cases to stress test systems using variations of the original tests [12]. High volume software testing using genetic algorithms do not have to be based upon white-box testing, as black box tests can also be modified to create new tests. In [12], genetic algorithms were applied to decision trees that contained rules for what areas of source code in projects to test. The authors proposed techniques for combining automated test case generation with long sequence testing. In particular, an approach involving genetic algorithms was compared again a random search test. A decision tree was used to generate rules to determine where defects in the code might occur. The generations were kept consistent in both tests. With 25 generations, random testing and genetic algorithm version performed equivalently, which potentially indicated that the genetic algorithm did not have enough time to evolve and distinguish itself from the random search. When the number of generations was increased to 500, the genetic algorithm outperformed the random search by 7% accuracy. However, the authors did not mention if the results were statistically significant.

As a side note, [12] provided a stark contrast to the majority of other software testing studies. Since the authors are from a business school, it focused on the cost savings of handling software errors within a development lifecycle rather than the current after-the-fact methods. The economics of new methods should not be overlooked as it can account for a substantial motivation for undertaking new research.

3.2.2 Specific Program Feature Execution against Specifications

Another area where evolutionary testing has been applied is testing a specific program feature against a formal specification. In most software development projects formal specifications are not available. However, in mission-critical systems and projects that can afford the overhead of formally specifying some or all requirements, evolutionary testing can be used in a different way than white-box structural testing.

3.2.3 Grey-box Feature Validation Testing

Grey-box testing combines structural and functional information sources to create new test cases [5]. Two primary examples of testing in this area are for assertions and exceptions. Essentially these conditions provide metadata about a executing programs.

For assertion testing, test case generators attempt to find data that violate assertions inserted by the original programmers. The assertions specify constraints with how the program should execute. If test data is created that violates an assertion, it indicates an error in the program.

Grey-box testing with exceptions is similar to that of assertion testing. The major difference compared to assertions is that exception testing is often used as an extension of standard conditional statements. The raising of an exception does not always actually indicate an error in the program. Instead, an exception may provide additional assurance that a block of code executed in a way unforeseen by the designer but the resulting data could still be valid.

A drawback of grey-box testing is that the languages must support a mechanism for metadata, most commonly assertions or exceptions. However, this is not a major criticism of the technique since most common languages in use today, such as Java, Python, and C++ support one or more of these constructs. A more serious criticism is that assertions and or exceptions must be embedded in the program during development, which mitigates the advantage of reducing development time by automatically creating test data.

Grey-box testing is different from the work presented in this paper because standard structural white-box testing does not rely on metadata embedded within programs to generate test data. Instead, it relies on the code itself and execution of test cases to create test cases.

3.2.4 Non-functional Requirement Validation Testing

Several previous studies have researched the applicability of evolutionary testing in validating non-functional requirements. The primary example of non-functional validation testing is testing program execution time.

4. Solution Approach

The research tool ETCG was designed to enable logical separation between the heuristic process and the supporting functionality. By making the heuristic search-based process generic, it is possible to switch in other methods for generating test data, such as simulated annealing and random data generation.

4.1 Technical Components

ETCG can be broken down into five major components that support the test case generation process. The cardinal component for research in this study is the design knowledge module. It breaks down sequence diagrams into a form that can be used by the other components to improve the results of the test case generation process. The second most important part is the genetic algorithm engine, which creates new test cases as individuals and executes them against source code. The second part is the generator and its support modules. The support modules include the user interface, file utilities, error handling, and classes for bundling the best test cases together in a test suite. Another component of ETCG is the heuristic translator, which supports the logical separation between the generator, the design knowledge, and the implemented heuristic engine. The heuristic translator allows the generator to interchange various heuristic and random processes to test whether, for instance, how genetic algorithms perform against simulated annealing. Another component is the user interface. There are two main ways to interact with ETCG, including a web browser-based version and a command-line interface. The main technical components are shown below in figure 2.

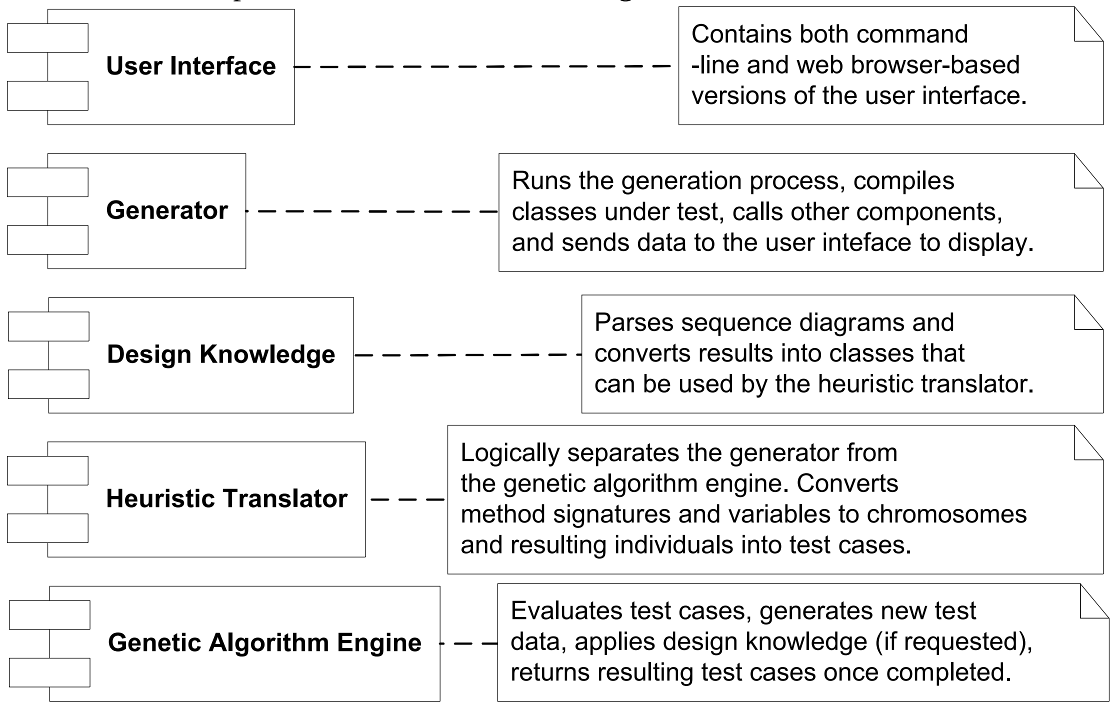

User Interface
Contains both command -line and web browser-based versions of the user interface.
Generator
Runs the generation process, compiles classes under test, calls other components, and sends data to the user inteface to display.
Design Knowledge
Parses sequence diagrams and converts results into classes that can be used by the heuristic translator.
Heuristic Translator
Logically separates the generator from the genetic algorithm engine. Converts method signatures and variables to chromosomes and resulting individuals into test cases.
Genetic Algorithm Engine
Evaluates test cases, generates new test data, applies design knowledge (if requested), returns resulting test cases once completed.

Figure 2: Main Technical Components of ETCG

Several scenarios exist for working with ETCG. These situations are presented in figure 3. First, a user may need to change the properties of ETCG to perform experiments and determine the most efficient way to generate test cases. For this purpose, configuration files in XML format are provided by the system for a user to work with and change pertinent settings. The second and third operations, submitting sequence diagrams and source code, are the most likely scenarios for general users. Source code can be submitted without UML diagrams and ETCG will perform a best attempt at creating test cases for the classes and individual methods. Sequence diagrams can also be submitted at the same time as the source code to assist in the test case generation process. When the user provides multiple files, they should be bundled in Java Archives (JARs) so there is only a single file to submit. The generator takes care of removing the source code and sequence diagrams from the archive before the generation process. After submitting source code and optionally the sequence diagrams, the user will need to obtain the test cases. The generator provides a single archive file with the best test cases and an HTML and XML report with the statistics of how those test cases perform.

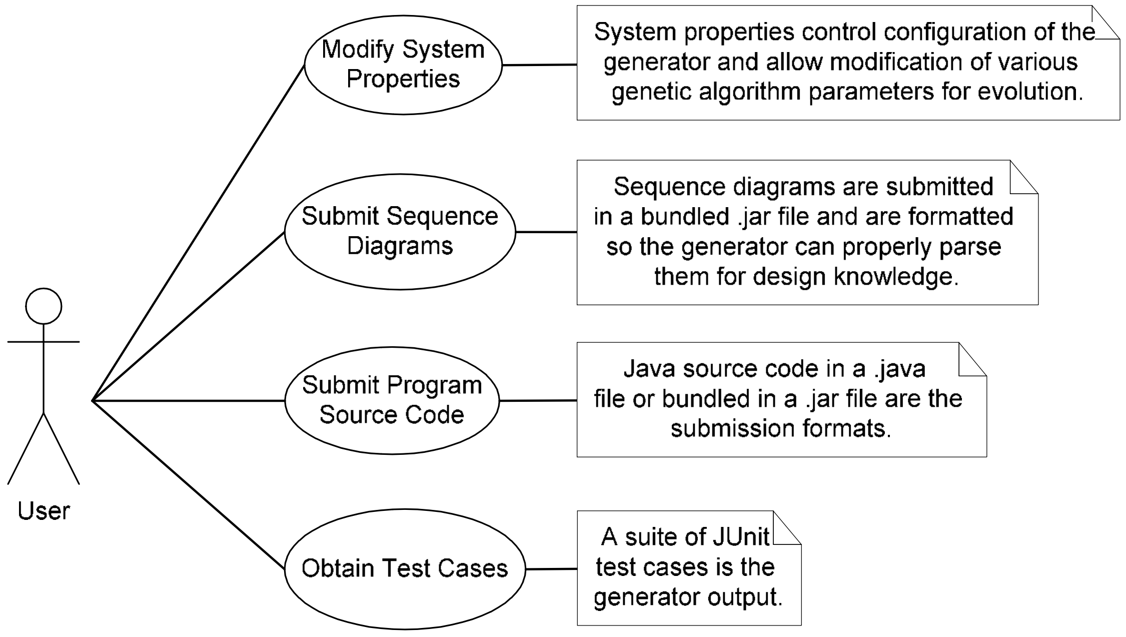

Modify System Properties
System properties control configuration of the generator and allow modification of various genetic algorithm parameters for evolution.
Submit Sequence Diagrams
Sequence diagrams are submitted in a bundled .jar file and are formatted so the generator can properly parse them for design knowledge.
Submit Program Source Code
Java source code in a .java file or bundled in a .jar file are the submission formats.
User
Obtain Test Cases
A suite of JUnit test cases is the generator output.

Figure 3: General Operations the User Will Perform With ETCG

4.2 Genetic Algorithm Engine

The most prominent component of the solution other than the design knowledge module is the genetic algorithm heuristic engine, which is used to generate the test data. Genetic algorithms are used to direct the test case generation process so data created maximizes a given metric, such as branch coverage or statement coverage. In fact, any metric can be used by ETCG as the fitness function as long as it can be calculated after running the generated test cases against the original source code.

The engine is designed so as to maintain the genetic algorithm abstraction where various selection, crossover, and mutation mechanisms can be altered without modifying anything outside this component. In the ETCG implementation, each part of the genetic algorithm process is organized in separate classes, as shown in figure 4.

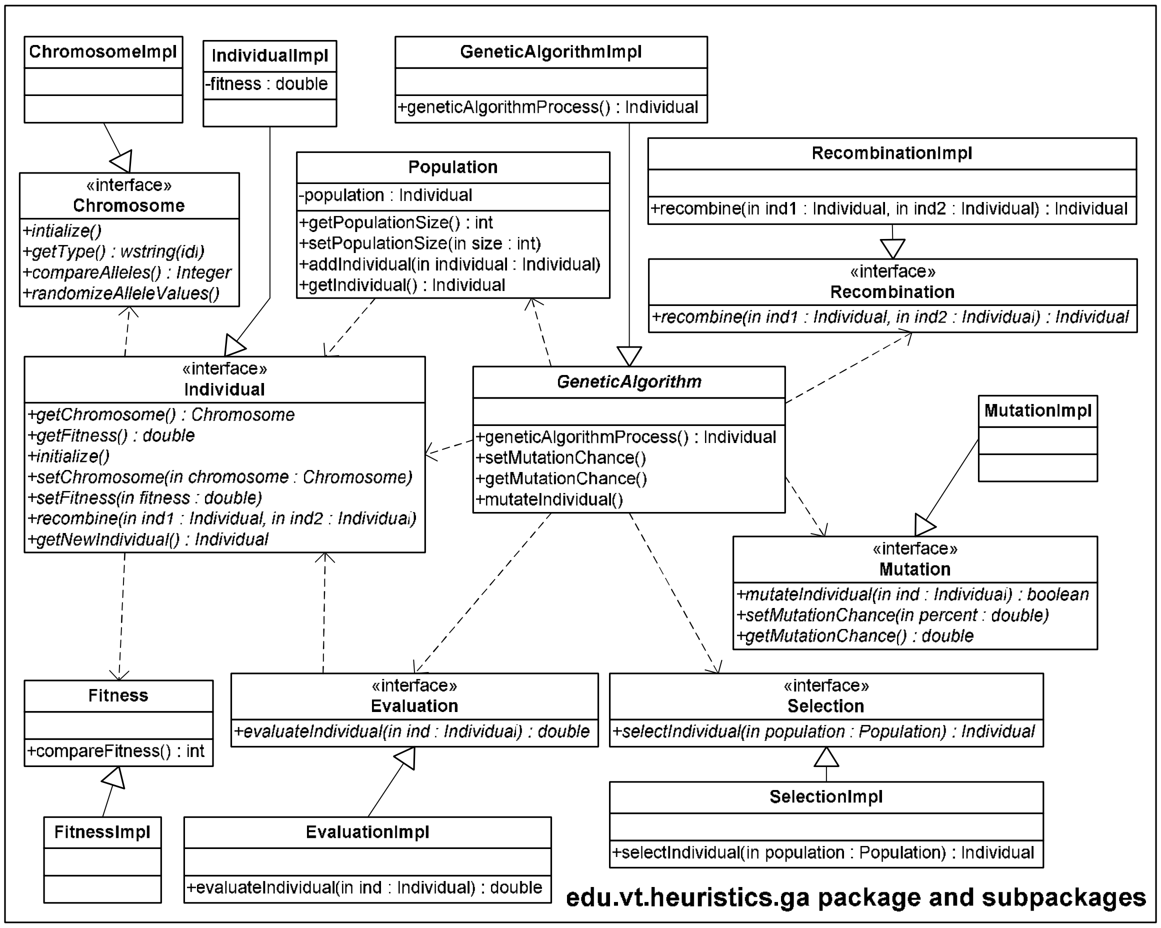

ChromosomeImpl
IndividualImpl
-fitness : double
GeneticAlgorithmImpl
+geneticAlgorithmProcess() : Individual
«interface»
Chromosome
+initialize()
+getType() : wstring(idl)
+compareAlleles() : Integer
+randomizeAlleleValues()
Population
-population : Individual
+getPopulationSize() : int
+setPopulationSize(in size : int)
+addIndividual(in individual : Individual)
+getIndividual() : Individual
RecombinationImpl
+recombine(in ind1 : Individual, in ind2 : Individual) : Individual
«interface»
Recombination
+recombine(in ind1 : Individual, in ind2 : Individual) : Individual
«interface»
Individual
+getChromosome() : Chromosome
+getFitness() : double
+initialize()
+setChromosome(in chromosome : Chromosome)
+setFitness(in fitness : double)
+recombine(in ind1 : Individual, in ind2 : Individual)
+getNewIndividual() : Individual
GeneticAlgorithm
+geneticAlgorithmProcess() : Individual
+setMutationChance()
+getMutationChance()
+mutateIndividual()
MutationImpl
«interface»
Mutation
+mutateIndividual(in ind : Individual) : boolean
+setMutationChance(in percent : double)
+getMutationChance() : double
Fitness
+compareFitness() : int
«interface»
Evaluation
+evaluateIndividual(in ind : Individual) : double
«interface»
Selection
+selectIndividual(in population : Population) : Individual
SelectionImpl
+selectIndividual(in population : Population) : Individual
FitnessImpl
EvaluationImpl
+evaluateIndividual(in ind : Individual) : double
edu.vt.heuristics.ga package and subpackages

Figure 4: Genetic Algorithm Classes

The main component is the *GeneticAlgorithm* class, which drives the evolutionary process, which in this study creates test cases. However, the class is flexible enough to be used for any general genetic algorithm study. The purpose of the implementation was to build a general genetic algorithm component instead of a one use only program.

Standard selection mechanisms provided by ETCG include n-individual tournament selection, fitness-proportionate selection, and sigma scaling [3]. Each can be used in conjunction with elitism, where the best individual is saved during the creation of new populations, to boost efficiency under circumstances where the users find it appropriate. Crossover methods depend upon the representation encoding the user chooses. The standard encodings such as binary strings and real numbers are implemented by default. The user can also choose to create a new representation by creating a new class with the required fields specified in the Representation interface. New individuals can be created using standard crossover methods such as single-point, double-point, and allele averaging. Again, the genetic algorithm engine is abstract and extendible enough so that users can create new crossover types by implementing the Crossover interface and its standard methods. Another interface provides mutation to maintain diversity in each newly generated population. The Mutation interface supports bit-flipping and random floating-point addition and subtraction by default. Table 1 presents a summary of each of these interfaces and the implementations provided by ETCG. For further explanation of these search-based method types, refer to [5].

Interface	**Standard ETCG Implementation Types**
Selection	*N*-individual tournament, fitness-proportionate selection, random selection
Representation	Binary string, real number string, dynamic variable string based on a common superclass, multiple test cases per allele representation
Crossover	Single-point, double-point, allele averaging, random new individual
Mutation	Bit-flipping, random floating-point addition and subtraction
Evaluation	Branch coverage attained by testing, statement coverage attained by testing, random evaluation

Table 1: Genetic Algorithm Interfaces Provided By ETCG

4.3 Generator and Support Modules

The generator and its support modules provide the front-end of ETCG. The modules work together as follows:

1. User interface – two user interfaces are provided, a web-browser version for uploading files to a web application server, and a command-line interface for submitting local files
2. Testability transformations – an optional component for applying testability transformations, described in section 2.2.1, to the source code before compilation
3. File utilities – numerous utilities for opening and creating Java Archive (JAR) files, reading and writing files, compiling single and groups of classes, and copying files to necessary directories are required for the generator to automatically create test cases
4. Generation process – the generator's execution is dependent upon the completion of numerous steps, which is configured and ordered in the main generator module

The support modules are not a novel component to the test data generation process, but they are nevertheless an important piece that takes care of the mundane tasks necessary for successful operation.

4.4 Heuristic Translator

The heuristic translator provides logical separation between the generator and the heuristic method used to generate the test cases. There are two subcomponents to the heuristic translator: a source code to chromosome translation and a chromosome to executable translation. In addition, this section also describes how the resulting test cases use EasyMock as substitutes for real objects as unit testing often requires.

4.4.1 Translation of the Source Code Under Test To Chromosomes

First, a source code to chromosome translation is required to convert files submitted to the generator into a form that can be used by genetic algorithms. If another heuristic method was used in place of genetic algorithms, then a conversion mechanism to translate source code into a form that could be manipulated by that heuristic process would be required instead.

4.4.2 Translation of Chromosomes to Executable Test Cases

The second part to the heuristic translator is the conversion from a chromosome into a test case file. The conversion is required so that the individuals created by the genetic algorithm can actually be run against the source code under test. Once the chromosome is translated into an executable test case, it can be analyzed by the testing code coverage tool to determine its performance. The results of how well the test case perform are then gathered by the generator to analyze which test cases should have a higher probability of recombining to create the next generation of test cases.

4.4.3 EasyMock and EasyMock Class Extension

Worth noting during the translation process is the use of EasyMock and the EasyMock Class Extension to create substitutes for interfaces and objects that are otherwise difficult to instantiate. EasyMock creates mock objects that execute in place of real objects so that unit tests can focus on testing *units* of code, as opposed to the entire system [15].

Mock objects are used as a test harness in object-oriented programming to isolate sections of code and perform unit tests. Although previous studies have not used mock objects, the results in this thesis are still applicable for comparison with those prior studies because the mock objects perform the same functions as true objects. The mock objects are simply used as stubs and the use of mock objects is a common object-oriented unit testing technique.

4.5 Design Knowledge Component

The primary component of ETCG for experimentation is the design knowledge module. There are two phases to the operation of the design knowledge component. First, the parser breaks down sequence diagrams into their constituent pieces and creates objects. The second phase uses the design knowledge objects to insert additional statements into the test cases and passes the objects to the heuristic translator so the objects can be used by an evaluation function, if necessary.

4.6 Automated Test Case Generator Execution

ETCG execution will follow the sequence described in figures 5 and 6. There are two scenarios that are used to evaluate the effectiveness of using design knowledge during test case generation. The first scenario, shown in figure 5, is a standard test case creation process without design knowledge. The second scenario, shown in figure 6, uses design knowledge to enhance the test case generation effectiveness as outlined in the hypothesis.

4.6.1 Cobertura – Testing Code Coverage Tool

The open source tool Cobertura is used by ETCG to obtain the percentages of branch coverage and statement coverage achieved by individuals in generated populations. Cobertura is “a free Java tool that calculates the percentage of code accessed by tests [2].” Another tool known as jcoverage, provides the basis for Cobertura’s capabilities to determine branch and statement coverage.

One reason for choosing Cobertura was the HTML and XML reports that it automatically generates during execution. A feedback mechanism for the genetic algorithms was required so the individuals created by ETCG, each of which represented multiple test cases, could be evaluated. The XML reports are parsed by ETCG using the Document Object Model (DOM) standard, which is a common way to extract data from hierarchical XML files [20].

One disadvantage of using Cobertura is that evaluation of a typical population can take tens of minutes because every individual must be written to disk before it is tested. This downside can be overcome by using a portion of RAM as a hard drive so that Cobertura’s access time is much faster. ETCG was successfully tested using the “RAM as a hard drive” method and the results did indicate it ran roughly an order of magnitude faster, but for consistency all executions in this thesis were performed using a standard hard drive as the storage location for generated test cases.

4.6.2 Generator Execution without Design Knowledge

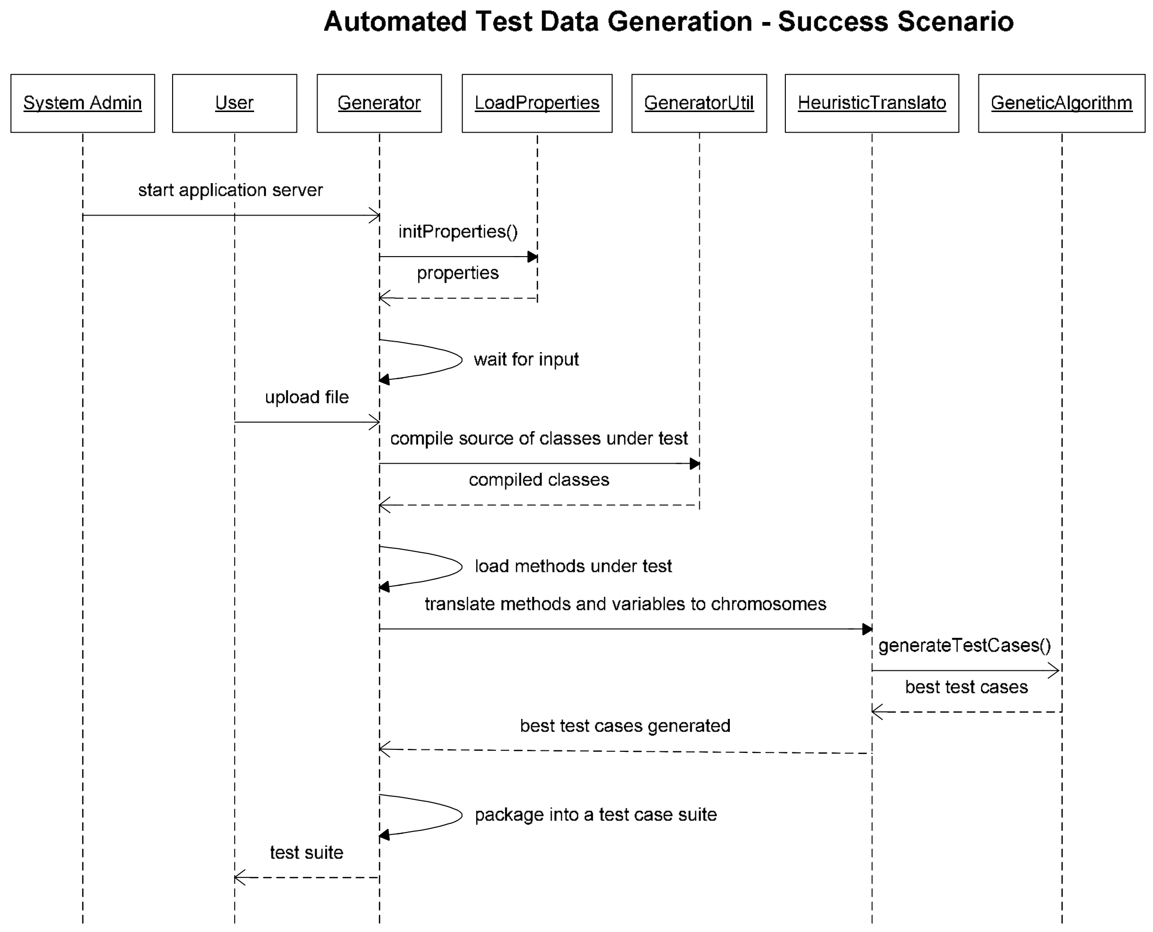


Figure 5: ETCG Test Generation Success Scenario without Design Knowledge

The operational scenario presented in figure 5 begins with the initialization of the ETCG system by a system administrator. Upon start up, ETCG loads runtime properties from two separate XML files. The first one is for the generator itself and it contains locations for important files that are used during the test case generation process. It also contains the "design knowledge" property, which can be set to true or false to perform the test case creation with or without the UML sequence diagrams. The generator then goes into a waiting mode until a .java or .jar file is submitted through the web browser-based interface. Once a user uploads a file, the main test case generation process commences. If a Java Archive (JAR) file is uploaded by the user, the source files are extracted from it and stored in a temporary location (this optional step is not shown in the sequence diagram). Once the source files are extracted or if a single Java source file is uploaded, the generator compiles the files into classes that can be loaded by the Java Virtual Machine (JVM). The classes loaded by the JVM are known as the classes under test (CUT). After loading the CUT, they can be inspected and manipulated through Java reflection, which is necessary to determine which methods to create JUnit test cases for. After loading the classes, the heuristic translator component is used to convert the method signatures, parameters, and associated variables into chromosomes, which are the structure used by the genetic algorithm engine. If another heuristic search method was used, such as simulated annealing, the heuristic translator would have to change the method signatures, parameters, and associated variables into a different usable form.

4.6.3 Generator Execution Using Design Knowledge

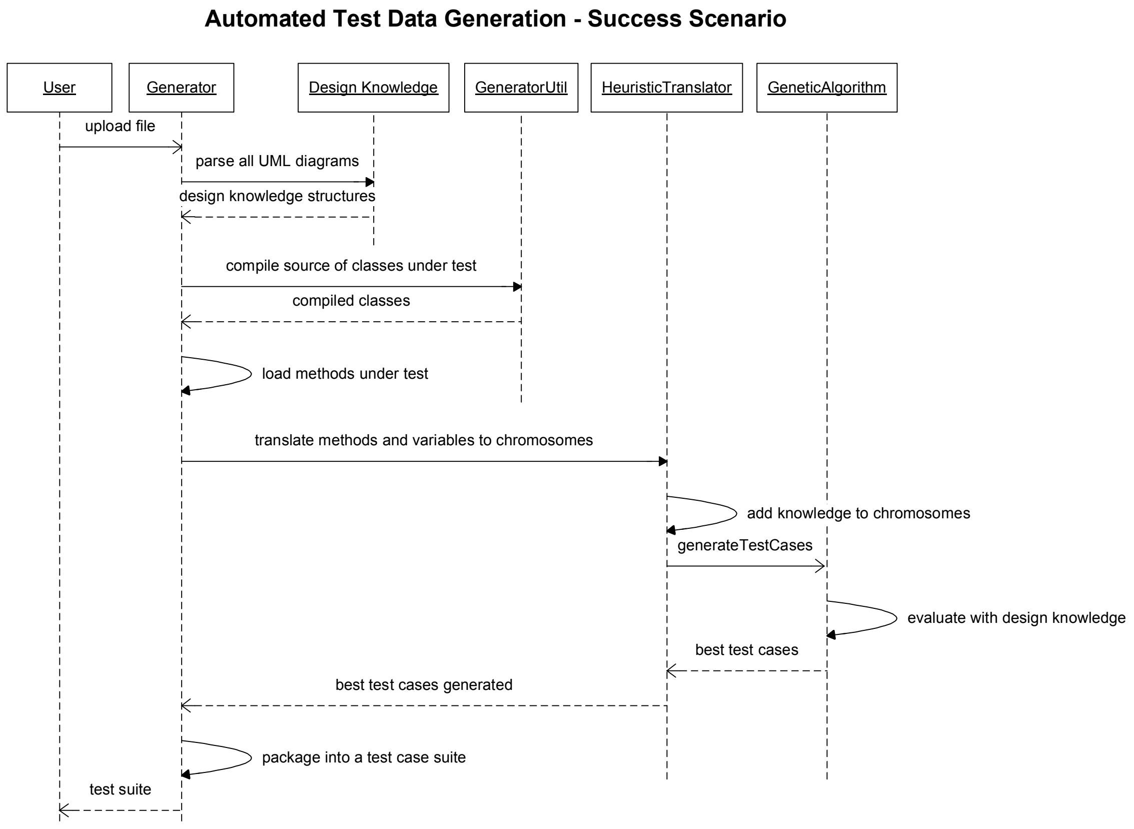

Automated Test Data Generation - Success Scenario
User
Generator
Design Knowledge
GeneratorUtil
HeuristicTranslator
GeneticAlgorithm
upload file
parse all UML diagrams
design knowledge structures
compile source of classes under test
compiled classes
load methods under test
translate methods and variables to chromosomes
add knowledge to chromosomes
generateTestCases
evaluate with design knowledge
best test cases
best test cases generated
package into a test case suite
test suite

Figure 6: ETCG Test Generation Success Scenario with Design Knowledge

Once the chromosomes are created, the heuristic translator then calls the genetic algorithm engine component with the newly-created chromosomes and lets it perform iterations of the evaluation, selection, crossover, and mutation phases. These iterations produce test cases that are more suited to the CUT than the original, randomly-generated values created for the translated chromosomes.

Figure 6 does not show the system administrator starting the system, but the beginning of the process remains the same as in figure 5. The test case generation scenario using design knowledge diverges from the situation where design knowledge is not used during the heuristic translation phase. The Heuristic Translator component uses the design knowledge to embed new method calls into the test cases. This allows for the creation of test cases that invoke methods that are critical to satisfying condition statements in the methods under test. Some of these conditions could otherwise not be satisfied, which decreases the achievable testing code coverage. At this point, the process continues in the same way as without design knowledge, as the genetic algorithm engine receives the chromosomes and begins the iterative process of evaluating, selecting, and creating new test cases.

4.6.4 Test Case Generation Activities

Several phases of activities are required for successful test case generation. An overview of the flow of activities is presented below in figure 7.

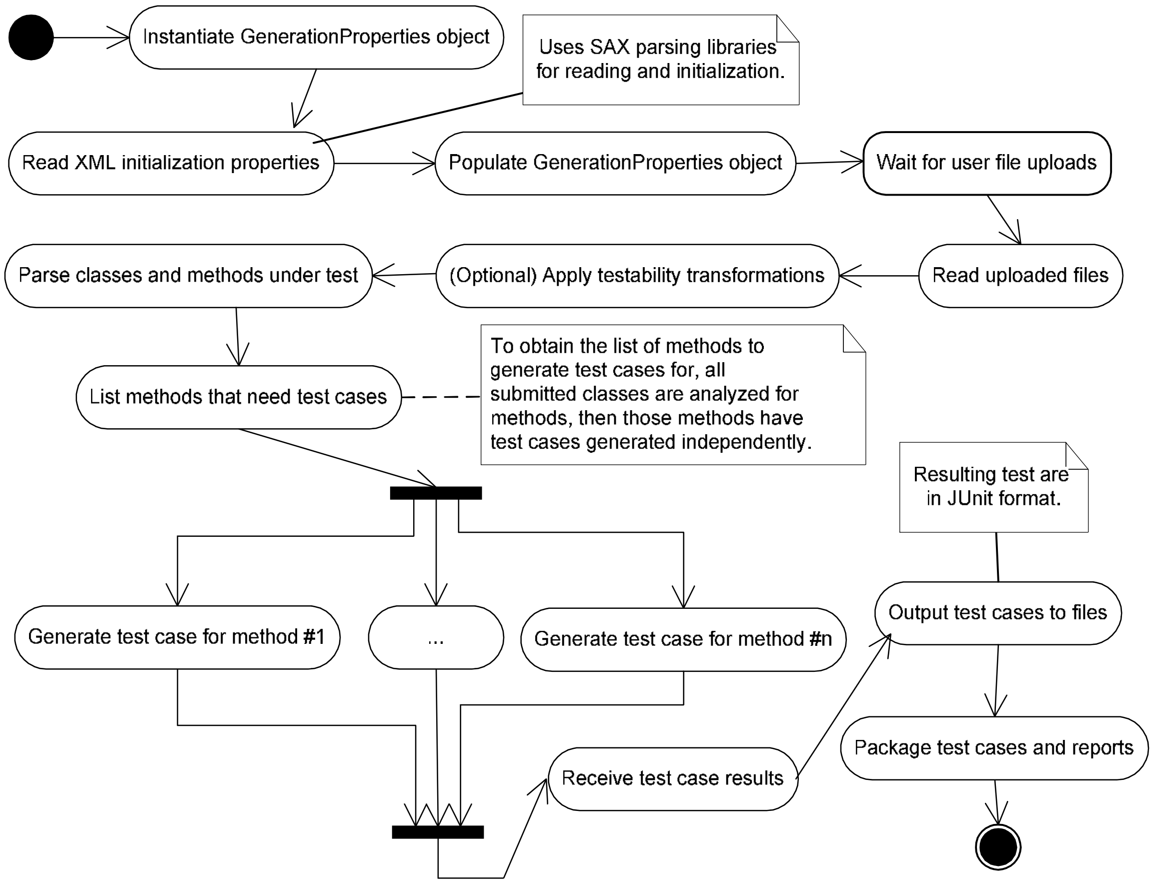

Instantiate GenerationProperties object
Uses SAX parsing libraries for reading and initialization.
Read XML initialization properties
Populate GenerationProperties object
Wait for user file uploads
Read uploaded files
(Optional) Apply testability transformations
Parse classes and methods under test
List methods that need test cases
To obtain the list of methods to generate test cases for, all submitted classes are analyzed for methods, then those methods have test cases generated independently.
Resulting test are in JUnit format.
Generate test case for method #1
...
Generate test case for method #n
Output test cases to files
Package test cases and reports
Receive test case results

Figure 7: Overview of ETCG Generation Activities

In figure 7, the diagram starts in the same manner as the sequence diagrams where the generator properties are loaded into a GenerationProperties object. The properties are read from an XML initialization file using a Simple API for XML (SAX) parser. After the properties are loaded, the system waits for a user to upload a file. Once an acceptable file is submitted in the Java Archive .jar or source code .java format, the process continues by reading and parsing the source files. There is an optional step after reading the source code where testability transformations can be applied. In this experiment, testability transformations were not used, but this would be the point where the generator could apply them. Next, the generator breaks down the classes under test into a list and the methods into a secondary list. Methods are the smallest unit that the generator produces test cases for, as unit testing refers to running method-level tests. At this point, ETCG cycles through the list of methods and runs the genetic algorithm process on each one to create unit tests for each method. The best resulting test cases, as measured by either the branch or statement testing code coverage, are returned to the generator. The generator then saves the test cases to files, creates reports on the testing coverage achieved, and bundles the result together in a single Java Archive file.

5. Experiment Design

The hypothesis was stated in the introduction. Further clarifications to the hypothesis are presented in section 5.1. Section 5.2 presents the parameters of the experiment, including the independent and dependent variables.

5.1 Hypothesis

The hypothesis was described in the introduction, but it is expanded upon here. The hypothesis function, , **c = f(k)**, is an attempt to depict the relationship between the amount of design knowledge, k, used as a parameter into the function, that affects c, the testing code coverage percentage realized by the test case generation process.

5.2 Experiment Parameters

To prove or disprove the hypothesis, independent and dependent variables to the hypothesis function were measured. Further details on the independent variable, the number of objects used parsed from sequence diagrams to guide the search process, are presented in subsection 5.2.1. Subsection 5.2.2 describes the dependent variable, which is the percentage of testing code coverage achieved by the produced automated test cases.

5.2.1 Independent Variable

As stated above in the hypothesis, **c = f(k)**, where k is the number of objects used in all sequence diagrams incorporated into the evaluation function. In this study, k is the independent variable because the amount of design knowledge incorporated into the test generation process can be varied by providing different amounts of sequence diagrams. When more objects are used in the evolutionary testing process, the process has more guidance for the search. If the hypothesis is true, then as the number of sequence diagrams and objects provided to the test generator increase, the testing code coverage percentage will grow as well.

5.2.2 Dependent Variable

The dependent variable is the percentage of testing code coverage, either conditional or statement, achieved at the end of the test generation process. Therefore in the hypothesis, **c = f(k)**, c is the final coverage realized. The value of the dependent variable determines whether the hypothesis is proven true or false.

Testing for statement coverage and branch coverage produced results that are shown in section 6. Block coverage, defined as a statement or group of statements which execute in series without branching or delays, was not used as a coverage metric [21]. The open source tool Cobertura used by ETCG to evaluate individuals' fitness does not support analyzing block coverage. If another code coverage tool that supported block coverage were substituted for Cobertura, it would be possible to obtain block coverage results.

5.2.3 Additional Experiment Parameters

There were several additional parameters in the experiment that were kept constant during all tests. First, the termination condition was the number of generations created and evaluated and it held steady at ten. Each generation contained ten individuals, where each individual represented a suite of five test cases. The selection mechanism was an n-individual tournament using eight randomly chosen individuals with the one with the best fitness used as a parent. Mutation happened at a 2% rate with one of the five test cases within an individual recreated if the mutation occurred. Finally, the recombination type was single point crossover, where the first two test cases were used from one parent and the last three test cases came from the second parent. Table 2 sums up the experiment parameters.

Parameter Type	Value During Experiment
Individuals per population	10
Test cases per individual	5
Termination condition	Number of generations elapsed
Generations	10
Selection Mechanism	Tournament selection with 8 individuals
Mutation	2%, with random regeneration of a single test case within an individual
Recombination	Single point crossover

Table 2: Constant Experiment Parameters

These parameter constants were chosen for the balance of a reasonable execution time versus the need to run the genetic algorithm for several generations to allow the search for the best test cases to show results. Exploratory analysis showed these parameters to be the most effective with ETCG with a subset of four programs from the larger 16 program sample set. The number of individuals per population and test cases per individual provided a balance between long execution time for large populations and lack of results from a small population size. The termination condition of using the number of generations elapsed is a standard practice in genetic algorithms. The tournament selection, mutation rate, and single point crossover are also standard genetic algorithm parameters used by previous studies.

5.2.4 Hardware Configuration

The hardware configuration for this experiment should not affect the results as execution time was not an important benchmark. However, the specifications for machine that ran all of the generator experiments remained consistent throughout all executions. The configuration was gathered by CPU-Z as follows [4]:

- Intel Pentium D 840 3.2GHz
- 2 gigabytes of DDR2 Dual Channel memory running at 266Mhz
- RADEON X300 SE with 128 megabytes
- 250 gigabyte hard drive

The operating system was Microsoft Windows XP Professional running Service Pack 2.

5.3 *Program Sample Collection*

5.3.1 Method For Obtaining Sample Programs

The programs used for the experiment sample in this study were gathered from public websites on the Internet. Searching was performed over several weeks using Google, Yahoo, and Microsoft search engines to find websites with sequence diagrams and related Java code or imperative programming paradigm-style pseudo-code.

The criteria for selecting program samples followed these rules:

1. Java source code or imperative-style pseudo-code (including actual code in Java-style languages) must be provided
2. Sequence diagrams must directly reference provided source code or pseudo-code
3. Source code must be well-contained without libraries of other code that do not provide source code
4. Source code must be manageable size, no more than several hundred lines of code
5. Source code must have conditional branches that would make random testing unable to achieve 100% code coverage

Several programs did not match the criteria. Numerous sequence diagrams are available on the Internet, but generally they are not provided with source code. Some of the programs found required a number of external libraries to compile properly. They were rejected since often the libraries to the source code were not available or over several hundred lines of code.

5.3.2 Program Samples

The complete set of programs that were used by the generator to create test cases for in this experiment is shown in Appendix B.

6. Experiment Results and Analysis

Experiments were conducted with ETCG to obtain both branch and statement testing code coverage. Section 6.1 details the results of the experiment. Section 6.2 reviews the outcome of the hypothesis and section 6.3 analyzes the results.

6.1 Experiment Results

Testing experiments were performed on the sixteen sample programs to determine the statement coverage and branch coverage that could be achieved with and without design knowledge. The statement coverage results are presented first, followed by the results for branch coverage.

6.1.1 Statement Coverage Achieved

Statement coverage percentages were obtained from the best individual of the final population in the experiment. Elitism was used in the genetic algorithm process, which means the best individual in the last population was also the best individual in the set of all populations during that specific run of the experiment. Without design knowledge, ETCG produced an average code statement testing coverage of 50% across all samples in the experiment. Using design knowledge produced an average code statement testing coverage of 72%. Therefore there was an average increase of 22% in statement coverage when design knowledge was used. Figure 8 shows the results of the experiment results for statement testing code coverage. The results indicate that the code coverage produced was more consistent at achieving the target values in the range of 70-80% statement coverage. These results match the goals previously stated in section one for achieving high coverage but not necessarily 100%.

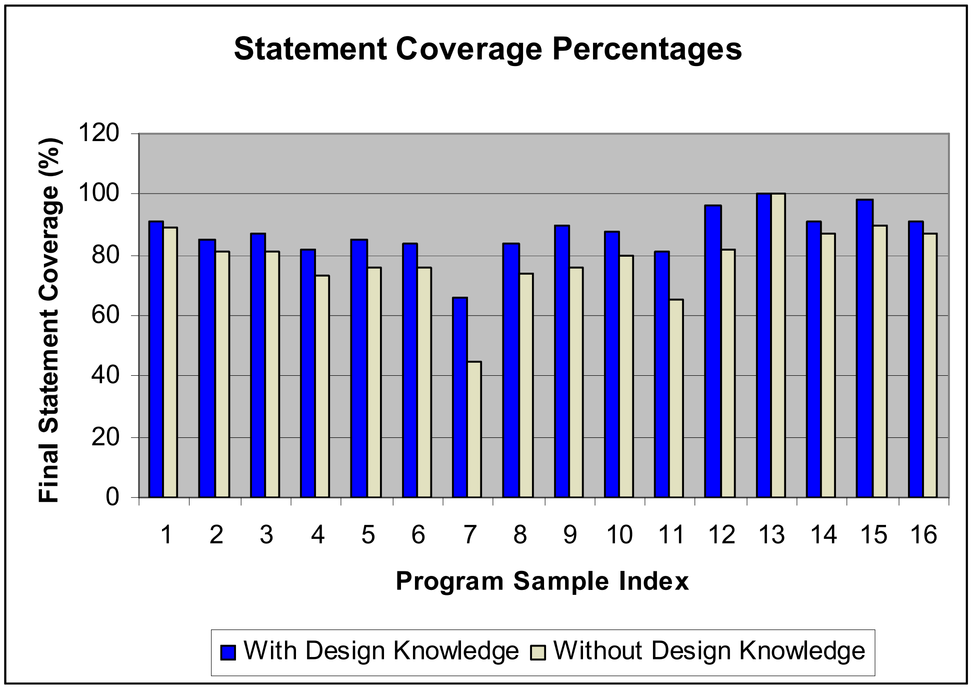


Figure 8: Statement Coverage Percentages Achieved by ETCG

6.1.2 Branch Coverage Achieved

Branch testing code coverage also used elitism in the population to retain the best individual during each generation of the program. Without design knowledge, ETCG produced an average code statement testing coverage of 79% across all samples in the experiment. Using design knowledge produced an average code statement testing coverage of 87%. Therefore there was an increase of 8% in statement coverage when design knowledge was used. Figure 9 shows the results of the branch testing code coverage. As was the case with the statement coverage results, the branch coverage outcomes for the sample programs were consistently higher when using design knowledge and generally fell in the 70-80% coverage range.

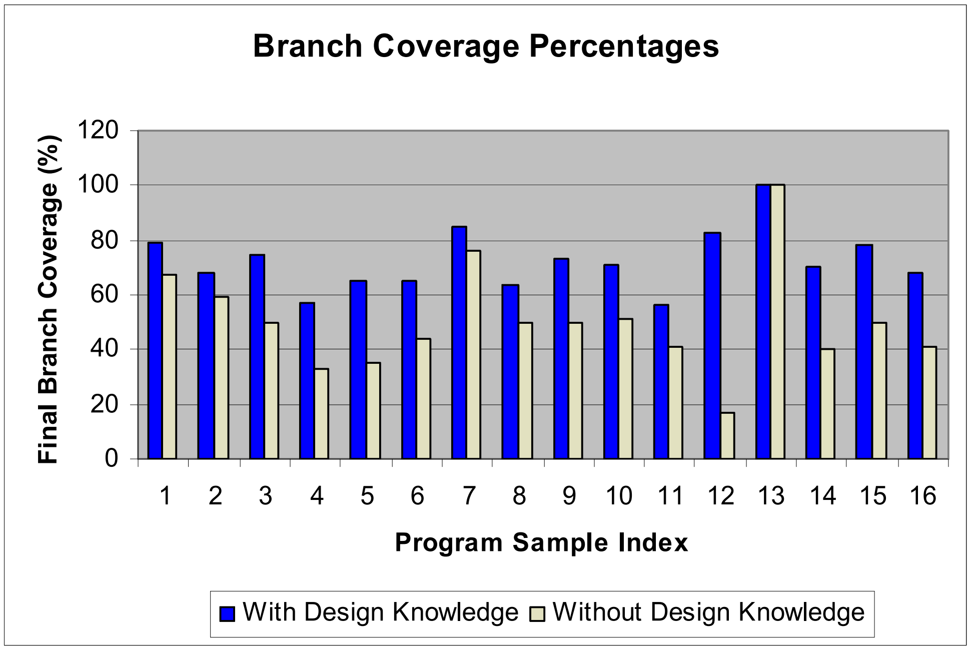


Figure 9: Branch Coverage Percentages Achieved by ETCG

6.1.3 Testing Coverage Results Averages

As shown in figure 10, the branch testing code coverage using sequence diagrams produced an increase of 21% over the results of test generation without incorporating the sequence diagrams as design knowledge. For statement testing code coverage, the increase was less pronounced, at 8% over the results without design knowledge.

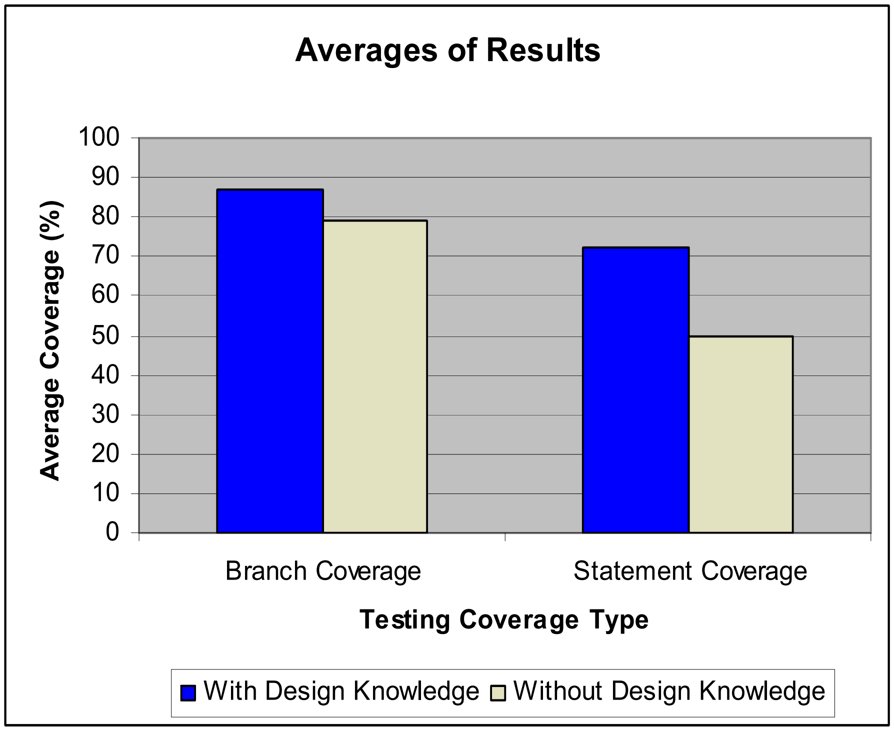


Figure 10: Average Results for Each Coverage Type

6.1.4 Branch Coverage Notched Box Plots

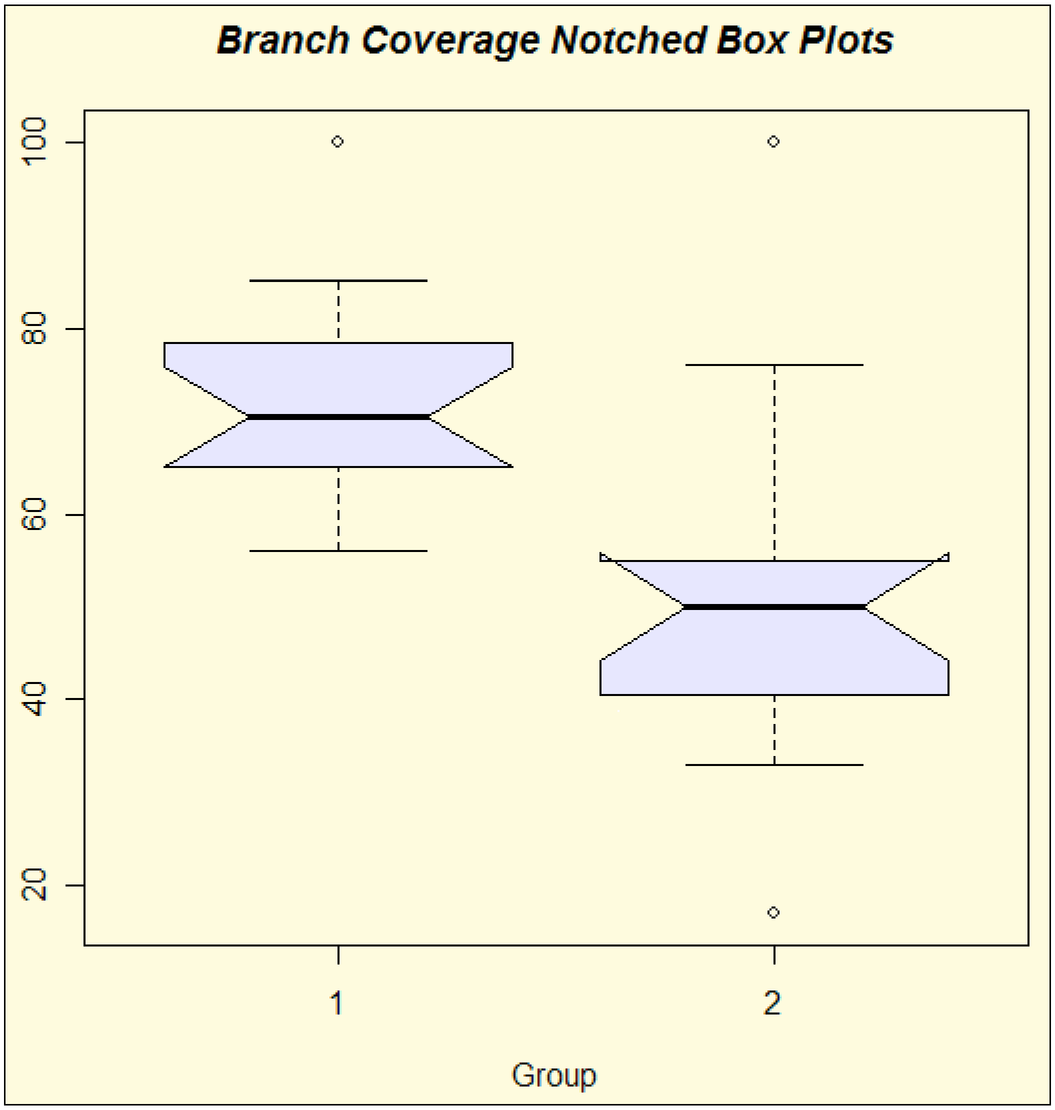


Figure 11: Branch Coverage Results, Group 1 Uses Design Knowledge, 2 Does Not

The results in figure 11 indicate that group 1, which uses design knowledge, tends to higher branch coverage results than group 2, which did not incorporate design knowledge. Both group 1 and group 2 contain outliers that show one of the sample programs was covered 100% by test cases generated with and without the incorporation of design knowledge. It is possible that this program was less complex than the other programs in the sample, but complexity measures were not analyzed for all sample programs. The most likely explanation for those outliers is that the conditional statements within the program were easily satisfied since they relied on Boolean variables as opposed to more complex integer or floating-point values.

Another outlier in group 2 was much lower than the branch coverage achieved for other sample programs. This sample program was the reverse of sample program that spawned the other two outliers because it contained many complex conditional statements that were not easy to satisfy. Therefore the branch coverage in that program suffered greatly because there was no guidance to the heuristic search methods' hunt for appropriate test data.

6.1.5 Statement Coverage Notched Box Plots

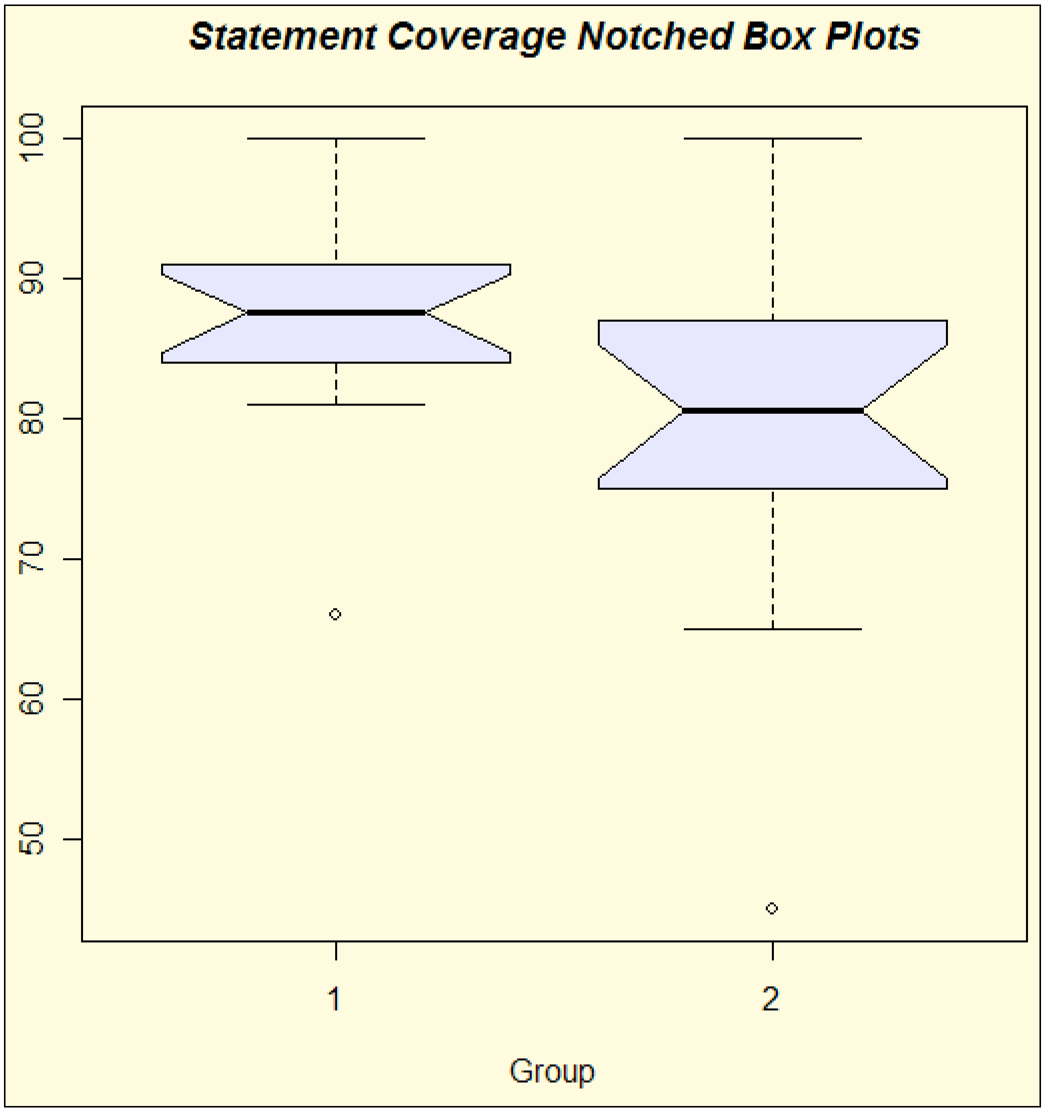


Figure 12: Statement Coverage Results, Group 1 Uses Design Knowledge, 2 Does Not

The results for the notched box plots show the increased statement coverage achieved in group 1 due to the incorporation of design knowledge into the test generation process. There were two outliers in the statement coverage notched box plot results. This program sample contained conditional statements that were difficult to satisfy even using the design knowledge. The resulting outcome from not satisfying those conditionals was that many of the statements inside those branches were not executed. Therefore the statement coverage dropped quickly since a large number of statements were clustered in the unsatisfied branches.

6.2 Hypothesis Test Results

The hypothesis test was to determine if the branch coverage and statement coverage would increase using design knowledge. Using design knowledge in the generation process proved both of these conditions true. The statement coverage did not increase by as much as the branch coverage for reasons discussed in the analysis section.

6.3 Results Analysis

Results from the experiment prove the hypothesis true for this set of sample programs. An interesting note is that the increase in branch coverage was higher than the increase in statement coverage in all programs where there was improvement that could be made. The exception is the single program that was already at 100% branch and statement coverage without the design knowledge added in. The reason for the higher increase in branch coverage than statement coverage was due to the code structure of the sample programs. Many branches of the code only contained a single statement to execute. In other cases, the number of statements within branches that were executed only with design knowledge was proportionally smaller than the code contained within other branches. Therefore, if more statements were contained within these more difficult to test branches, it is likely the statement coverage would have been higher.

7. Conclusion

Broad implications of the experiment results are presented in section 7.1. Section 7.2 presents potential areas for future work in evolutionary testing that build upon this study. A final review of the study concludes the paper in section 7.3.

7.1 Research Implications and Practicality

Automated test case generation cannot currently replace all forms of manual testing. Humans are able to test aspects that are difficult to specify, such as how the program is organized, whether or not it matches the informal requirements specifications, and the user interface. After the test cases are generated, an oracle is required to make sure the test results are correct. In addition, automated testing is no substitute for many of the intricate test cases that humans can develop. However, this research shows that heuristic testing can provide a more effective method for generating test cases by incorporating design knowledge in the form of sequence diagrams from the design phase. These generated test cases are important for testing program aspects that may not appear closely related to each other in the source code but are still critical for proper operation of the application. Therefore ETCG provides an effective program testing tool that can be used in conjunction with other testing methods.

ETCG and similar automated test case creation tools are practical for testing module interfaces after code is refactored. Since refactoring should not change the external interface of a software module, the original version can provide an oracle to determine if defects were added during the refactoring process. When an external interface is provided an input, the output from the refactored module should exactly match the output of the original module. If the output does not match, a defect was introduced during refactoring.

Another practical use for ETCG is to check changes in a code base over time. Source code control tools are widely used on development projects and monitor the revisions produced by developers. When a set of tests is automatically generated for one version and run against a previous or subsequent version, the tests that fail are due to changes in the source code. Some of the test failures indicate negative side effects of the code changes, which can be fixed before defects are propagated to a later testing stage. Further research is needed for how to most effectively manage finding defects with such a process, but automated test case generation tools could provide the large number of tests that would be required.

7.2 Future Work

The independent variable in the hypothesis is defined as a Boolean variable based on whether or not sequence diagrams are incorporated into the test case generation process. In future studies, the independent variable should be expanded from a true or false to the number of sequence diagrams. Since using a single sequence diagram improves the code coverage of resulting test cases, it should be tested whether additional sequence diagrams add further coverage. If so, there may also be a point at which diminishing returns eliminate the necessity of using more than a given number of sequence diagrams.

Another area of future work is code cyclomatic complexity as an independent variable. There may be a relationship between the complexity of the source code and the resulting branch or statement code coverage achieved by generated test cases. Since heuristic search-based methods were originally used to improve upon poor performance of random testing in complex software programs, it may hold that cyclomatic complexity of source code predicts how successful a generator would be in reaching a specific code coverage percentage.

The application of program slicing to control flow graphs and testability transformations are two of the newest developments in evolutionary testing. Both of these could provide benefits to test data generation without compromising the usefulness of the incorporation of design knowledge presented in this study. However, the use of both program slicing and testability transformations with design knowledge has not yet been tested. There may be additional hurdles to overcome before there are conclusive results on how well these techniques complement each other.

7.3 Conclusion

Incorporating design knowledge into evolutionary testing is a worthwhile method for improving testing code coverage in cases where there are difficult conditional statements. Some programs, especially when trivial, may already be suited for evolutionary testing without using the design knowledge of sequence diagrams. However, most applications that require test cases are complex and therefore do benefit from the inclusion of design knowledge into the evolutionary testing process.

References

[1] Roger S. Pressman, *Software Engineering: A Practitioner's Approach, 6th ed.*, New York, New York: McGraw Hill, 2005.
[2] "Cobertura," http://cobertura.sourceforge.net/.
[3] C. Rankin, "The Software Testing Automation Framework," *IBM Systems Journal*, Vol. 41, Number 1, 2002. Retrieved from http://www.research.ibm.com/journal/sj/411/rankin.html on December 18, 2007.
[4] Franck Delattre, CPU-Z, retrieved from http://www.cpuid.com on October 14, 2007.
[5] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105--156, 2004, retrieved from http://citeseer.ist.psu.edu/725641.html on November 30, 2007.
[6] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach, 2nd ed.*, Upper Saddle River, New Jersey: Prentice Hall, 2003.
[7] Piero P. Bonissone, Raj Subbu, Neil Eklund, and Thomas R. Kiehl, "Evolutionary Algorithms + Domain Knowledge = Real-World Evolutionary Computation." *IEEE Transactions on Evolutionary Computation*, 2006.
[8] M. Harman, L. Hu, R.M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability Transformation," IEEE Trans. Software Eng., vol. 30, no. 1 pp. 3-16, Jan. 2004. Retrieved from http://citeseer.ist.psu.edu/harman04testability.html on November 17, 2006.
[9] Baresel, A., Binkley, D., Harman, M., and Korel, B, "Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach." In *Proceedings of the 2004ACM SIGSOFT international Symposium on Software Testing and Analysis* (Boston, Massachusetts, USA, July 11 - 14, 2004). ISSTA '04. ACM, New York, NY, 108-118. Retrieved from http://doi.acm.org/10.1145/1007512.1007527 on November 15, 2007.
[10] T. Bäck, D.B. Fogel, and T. Michalewicz, *Evolutionary Computation 1: Basic Algorithms and Operators*, Bristol, United Kingdom: Institute of Physics Publishing, 2000
[11] C. Michael and G. McGraw. Automated software test data generation for complex programs. In 13th IEEE International Conferance on Automated Software Engineering, pages 136--146, October 1998. Retrieved from http://citeseer.ist.psu.edu/michael98automated.html on November 20, 2006.
[12] D.J. Berndt and A. Watkins, "High Volume Software Testing using Genetic Algorithms," HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on System Sciences, 03-06 Jan. 2005, pages 318b- 318b.
[13] Pargas, R. P., Harrold, M. J., and Peck, R. R. Testdata generation using genetic algorithms. The Journal of Software Testing, Verification and Reliability 9 (1999), pages 263-282. Retrieved from http://citeseer.ist.psu.edu/pargas99testdata.html on November 18, 2006.
[14] Korel, B. 1996. Automated test data generation for programs with procedures. *SIGSOFT Softw. Eng. Notes* 21, 3 (May. 1996), 209-215. Retrieved from http://doi.acm.org/10.1145/226295.226319 on November 17, 2006.
[15] "EasyMock," http://www.easymock.org/.
[16] John Holland, *Adaptation in Natural and Artificial Systems*, Cambridge, MA: MIT Press, 1992. 1st edition: 1975, The University of Michigan Press, Ann Arbor.
[17] Melanie Mitchell, *An Introduction to Genetic Algorithms*, Cambridge, MA: MIT Press, 1998.
[18] Xiao, M., El-Attar, M., Reformat, M., and Miller, J. 2007. *Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. Empirical Software Engineering.* 12, 2 (Apr. 2007), 183-239. Retrieved from http://dx.doi.org/10.1007/s10664-006-9026-0 on November 29, 2007.
[19] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. *Application of genetic algorithms to software testing (Application des algorithmes g_en_etiques au test des logiciels)*. In 5th International Conference on Software Engineering and its Applications, pages 625-636,Toulouse, France, 1992.
[20] "W3C Document Object Model," http://www.w3.org/DOM/.
[21] "Code coverage", http://www.vlsi-world.com/content/view/63/47/.

Appendix A – Acronym List

API – Application Programming Interface

DOM – Document Object Model

ETCG – Evolutionary Test Case Generator

JAR – Java Archive

JVM – Java Virtual Machine

SAX – Simple API XML

SPP – Species-Per-Path

UML – Unified Modeling Language

XML – Extensible Markup Language

Appendix B – Sample Programs

B.1 *Program #1: Getting Dynamic: Java and UML Interaction Diagrams*

http://www.informit.com/articles/article.aspx?p=29582&seqNum=2&rl=1
Retrieved on November 27, 2007.

B.1.1 UML Diagram

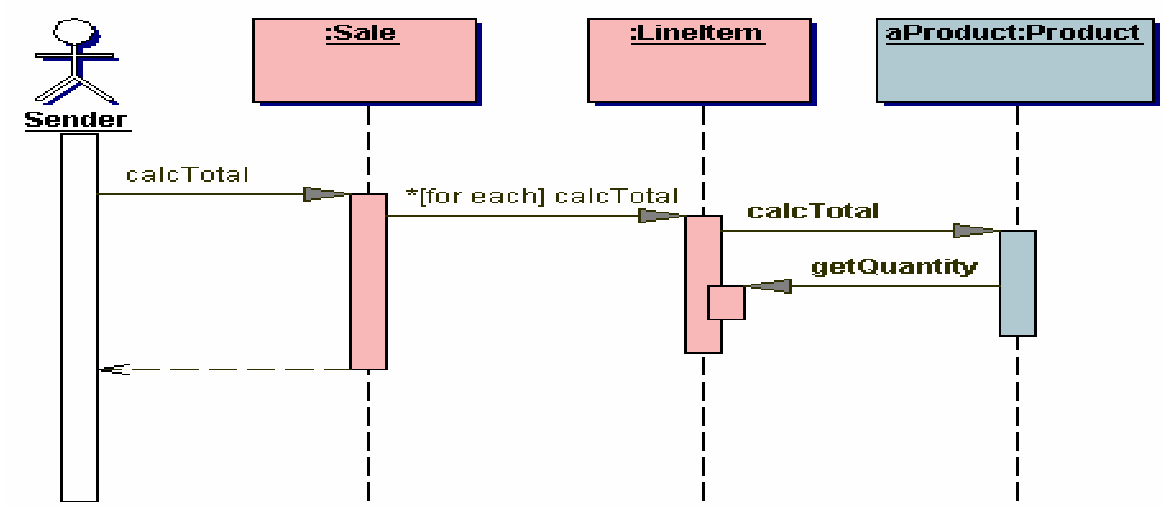


Figure 13: Program Sample #1 Sequence Diagram

B.1.2 Converted Sequence Diagram

```
program1.Sender main -> program1.Sale calcTotal()
program1.Sale calcTotal -> program1.LineItem calcTotal() returns lineItem
program1.LineItem calcTotal -> program1.Product calcTotal(lineItem)
```

B.1.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**2.5**
79	**67**	**8**	**91**	**89**	**2**	

Table 3: Results for Sample Program #1

B.2 *Program #2: An Address Book – Add Person*

http://www.math-cs.gordon.edu/courses/cs211/AddressBookExample/index.html
Retrieved on November 27, 2007.

B.2.1 UML Diagram

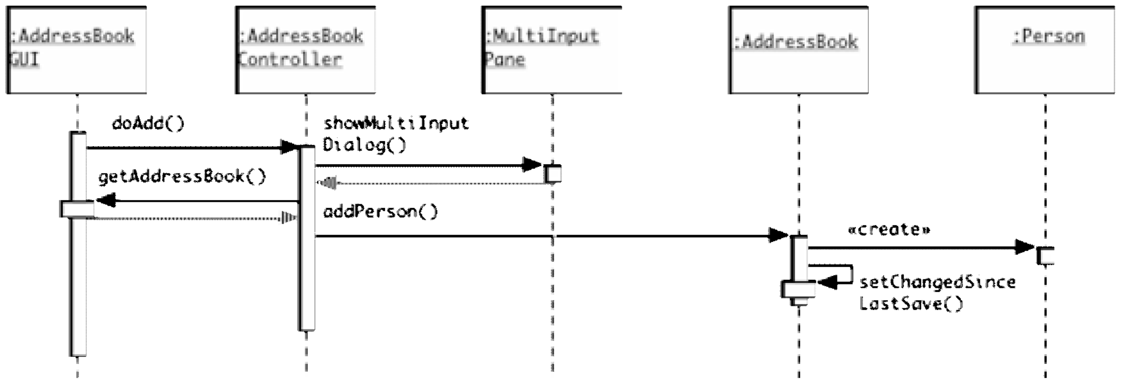


Figure 14: Program Sample #2 Sequence Diagram

B.2.2 Converted Sequence Diagram

```
program2.AddressBookGUI main -> program2.AddressBookController doAdd()
program2.AddressBookController doAdd -> program2.MultiInputPane showMultiInputDialog()
        returns bool
program2.AddressBookController doAdd -> program2.AddressBookGUI getAddressBook()
        returns AddressBook
program2.AddressBookController doAdd -> program2.AddressBook addPerson(Person)
program2.AddressBook setChangedSinceLastSave -> program2.AddressBook
        setChangedSinceLastSave()
```

B.2.3 Experiment Results

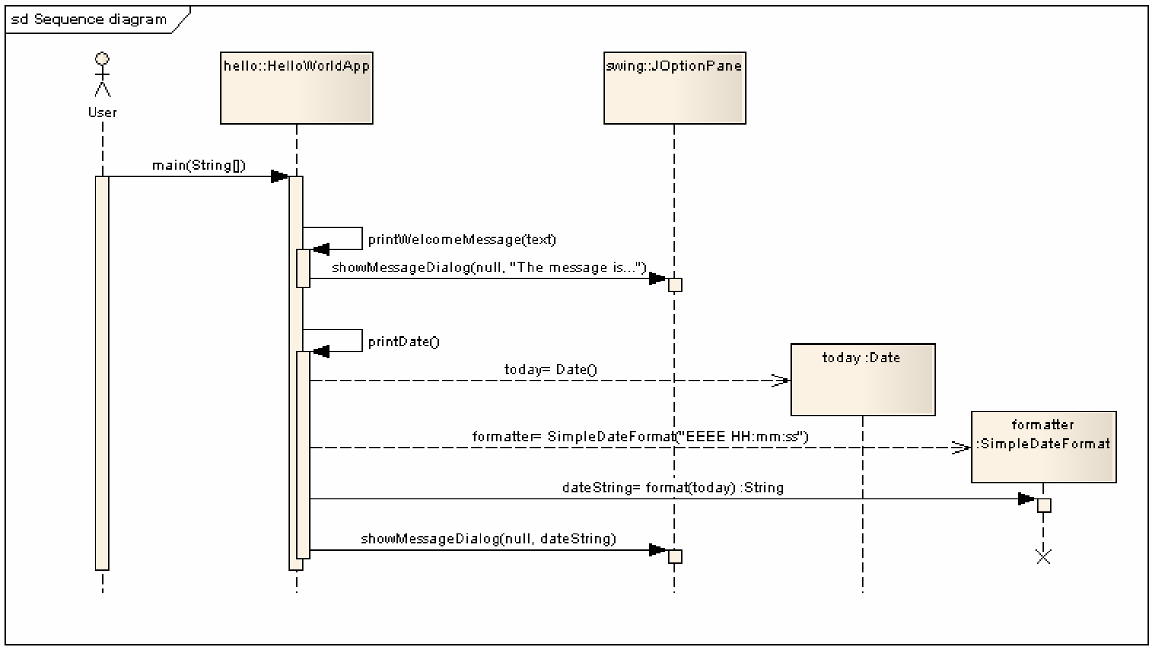
Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**1.63**
68	**59**	**9**	**85**	**81**	**4**	

Table 4: Results for Sample Program #2

B.3 Program #3: Hello World Printer with UML Sequence Diagram

http://wiki.msoe.us/doku.php?id=se1020labs:umlsequence
Retrieved on December 19, 2007.

B.3.1 UML Diagram



sd Sequence diagram
User
hello::HelloWorldApp
swing::JOptionPane
main(String[])
printWelcomeMessage(text)
showMessageDialog(null, "The message is...")
printDate()
today= Date()
today :Date
formatter= SimpleDateFormat("EEEE HH:mm:ss")
formatter :SimpleDateFormat
dateString= format(today) :String
showMessageDialog(null, dateString)

Figure 15: Program Sample #3 Sequence Diagram

B.3.2 Converted Sequence Diagram

```
hello.HelloWorldApp main -> hello.HelloWorldApp setCheck(int)
hello.HelloWorldApp main -> hello.Printer printWelcomeMessage(java.lang.String)
hello.HelloWorldApp main -> hello.Printer printDate()
```

B.3.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**2.5**
75	**50**	**25**	**87**	**81**	**6**	

Table 5: Results for Sample Program #3

B.4 Program #4: An ATM Simulation – Start Up

http://www.cs.gordon.edu/courses/cs211/ATMExample/index.html
Retrieved on February 15, 2008. Modified February 17, 2008.

B.4.1 UML Diagram

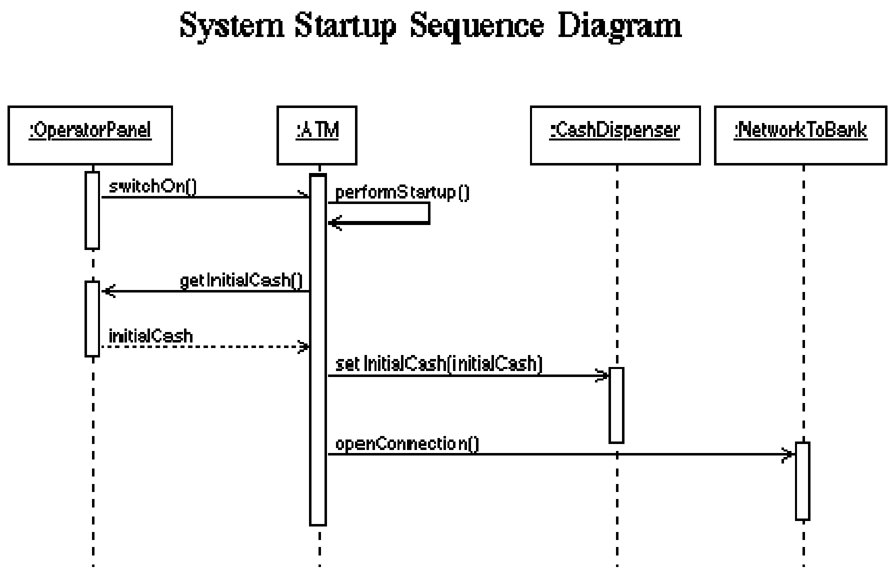


Figure 16: Program Sample #4 Sequence Diagram

B.4.2 Converted Sequence Diagram

```
program4.OperatorPanel main -> program4.ATM switchOn()
program4.ATM switchOn -> program4.ATM performStartup()
program4.ATM switchOn -> program4.OperatorPanel getInitialCash() returns cash
program4.ATM switchOn -> program4.CashDispenser setInitialCash(int)
program4.ATM switchOn -> program4.NetworkToBank openConnection()
```

B.4.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**1.88**
57	**33**	**24**	**82**	**73**	**9**	

Table 6: Results for Sample Program #4

B.5 Program #5: Java and UML Interaction Diagrams (Modified)

http://www.informit.com/articles/article.aspx?p=29582&seqNum=2&rl=1

Retrieved on November 27, 2007. Modified January 28, 2008 to increase code complexity.

B.5.1 UML Diagram

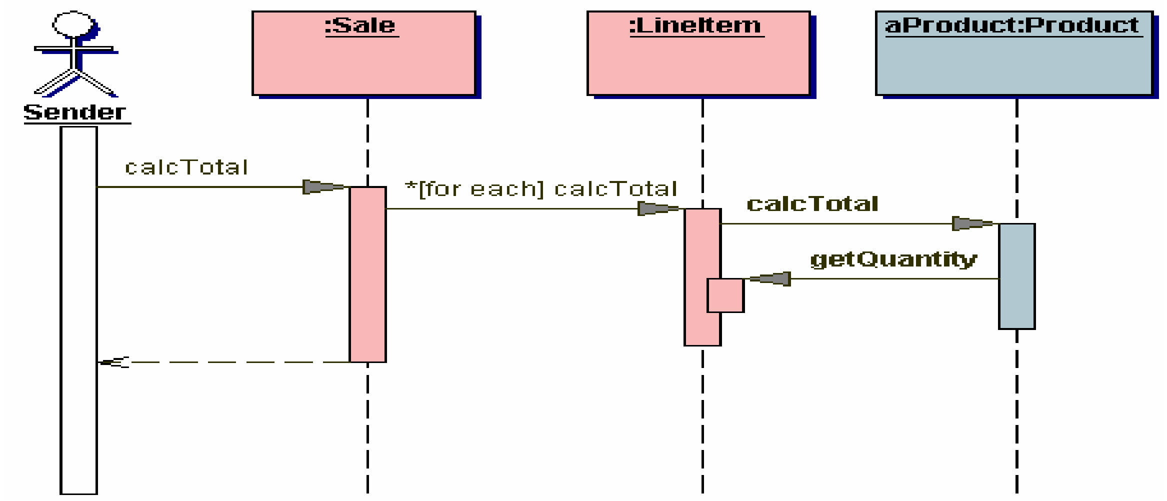


Figure 17: Program Sample #5 Sequence Diagram

B.5.2 Converted Sequence Diagram

```
program5.Sender main -> program5.Sale setSale(int)
program5.Sale calcSale -> program5.LineItem calcTotal(boolean)
program5.LineItem calcTotal -> program5.Product calcTotal(program1.LineItem)
```

B.5.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**1.89**
65	**35**	**30**	**85**	**76**	**9**	

Table 7: Results for Sample Program #5

B.6 Program #6: An ATM Simulation: Shutdown

http://www.cs.gordon.edu/courses/cs211/ATMExample/index.html
Retrieved on February 15, 2008. Modified February 17, 2008.

B.6.1 UML Diagram

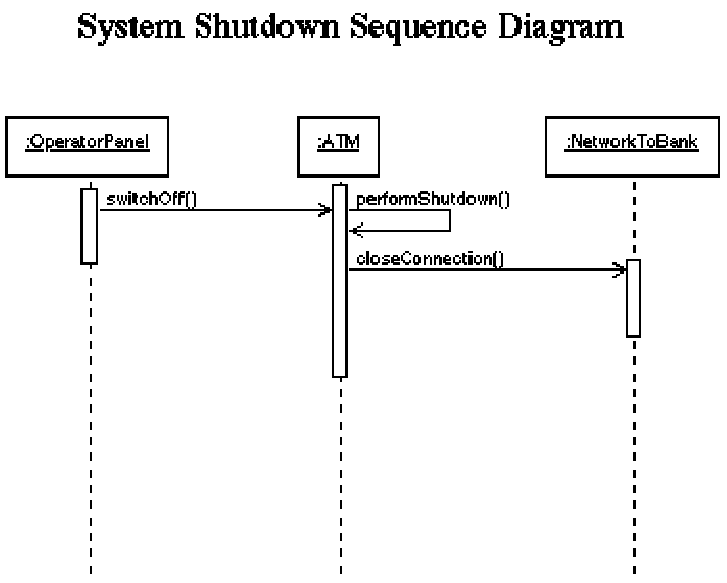


Figure 18: Program Sample #6 Sequence Diagram

B.6.2 Converted Sequence Diagram

```
program6.OperatorPanel main -> program6.ATM switchOff()
program6.ATM switchOff -> program6.ATM performShutdown()
program6.ATM switchOff -> program6.NetworkToBank closeConnection()
```

B.6.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**2.05**
65	**44**	**21**	**84**	**76**	**8**	

Table 8: Results for Sample Program #6

B.7 Program #7: An ATM Simulation: Session Sequence

http://www.cs.gordon.edu/courses/cs211/ATMExample/index.html
Retrieved on February 15, 2008.

B.7.1 UML Diagram

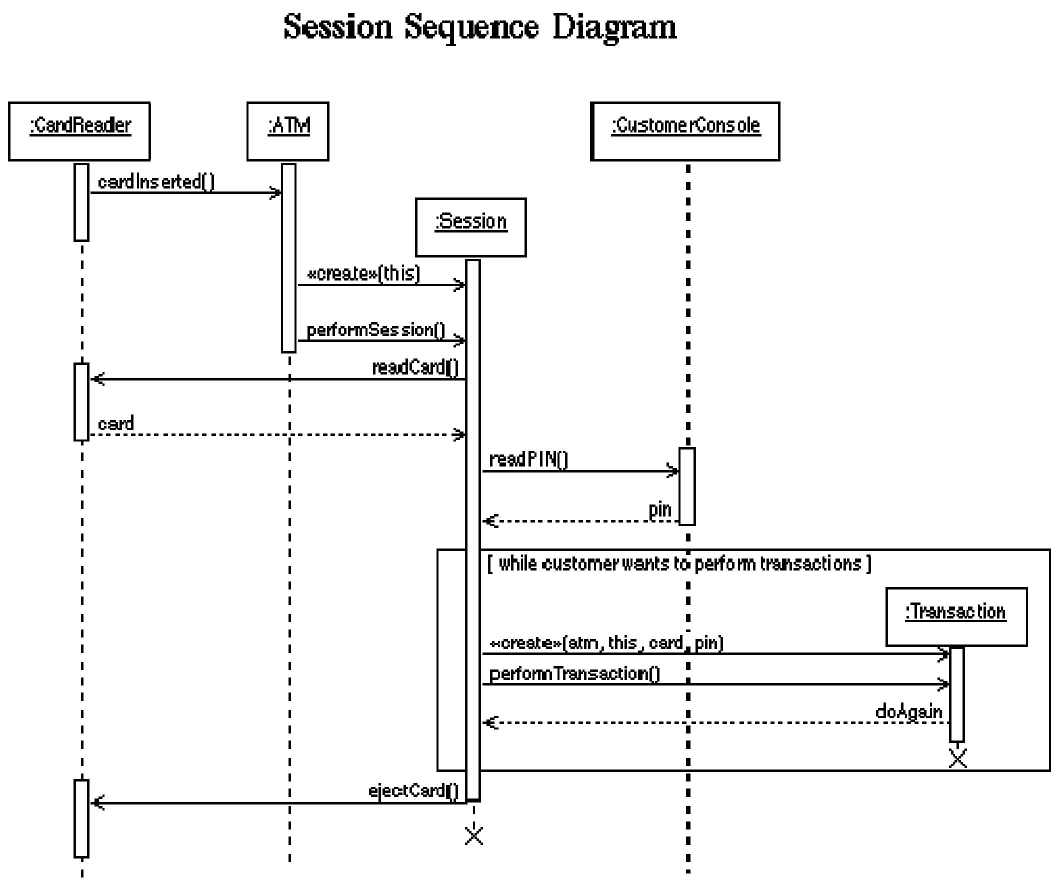


Figure 19: Program Sample #7 Sequence Diagram

B.7.2 Converted Sequence Diagram

```
program7.CardReader main -> program7.ATM cardInserted()
program7.ATM cardInserted -> program7.Session performSession()
program7.Session performSession -> program7.CardReader readCard() returns card
program7.Session performSession -> program7.CustomerConsole readPIN() returns pin
program7.Session performSession -> program7.Transaction performTransaction()
program7.Session performSession -> program7.CardReader ejectCard()
```

B.7.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**2.1**
85	**76**	**9**	**66**	**45**	**19**	

Table 9: Results for Sample Program #7

B.8 *Program #8: An Address Book: Sort Entries By Name*

http://www.math-cs.gordon.edu/courses/cs211/AddressBookExample/index.html
Retrieved on November 27, 2007.

B.8.1 UML Diagram

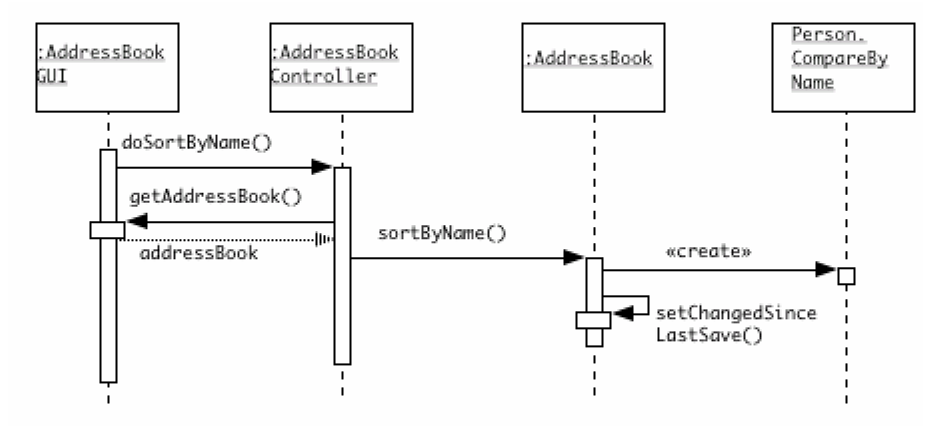


Figure 20: Program Sample #8 Sequence Diagram

B.8.2 Converted Sequence Diagram

```
program8.AddressBookGUI main -> program8.AddressBookController doSortByName()
program8.AddressBookController doSortByName -> program8.AddressBookGUI getAddressBook()
       returns book
program8.AddressBookController doSortByName -> program8.AddressBook sortByName()
program8.AddressBook sortByName -> program8.AddressBook setChangedSinceLastSave()
```

B.8.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**1.63**
64	**50**	**14**	**84**	**74**	**10**	

Table 10: Results for Sample Program #8

B.9 *Program #9: An Address Book: Sort Entries By ZIP*

http://www.math-cs.gordon.edu/courses/cs211/AddressBookExample/index.html
Retrieved on November 27, 2007.

B.9.1 UML Diagram

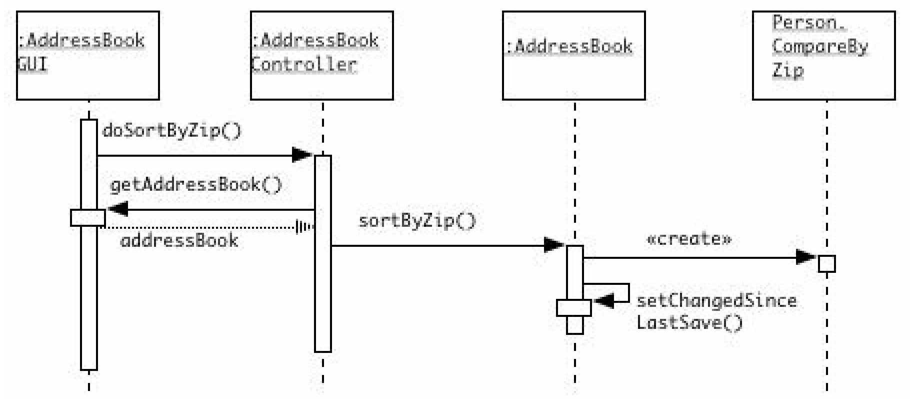


Figure 21: Program Sample #9 Sequence Diagram

B.9.2 Converted Sequence Diagram

```
program9.AddressBookGUI main -> program9.AddressBookController doSortByZip()
program9.AddressBookController doSortByZip -> program9.AddressBookGUI getAddressBook()
       returns book
program9.AddressBookController doSortByZip -> program9.AddressBook sortByZip()
program9.AddressBook sortByZip -> program9.AddressBook setChangedSinceLastSave()
```

B.9.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**1.65**
73	**50**	**23**	**90**	**76**	**14**	

Table 11: Results for Sample Program #9

B.10 Program #10: An ATM Simulation – Transaction Sequence

http://www.cs.gordon.edu/courses/cs211/ATMExample/index.html
Retrieved on February 15, 2008. Modified February 17, 2008.

B.10.1 UML Diagram

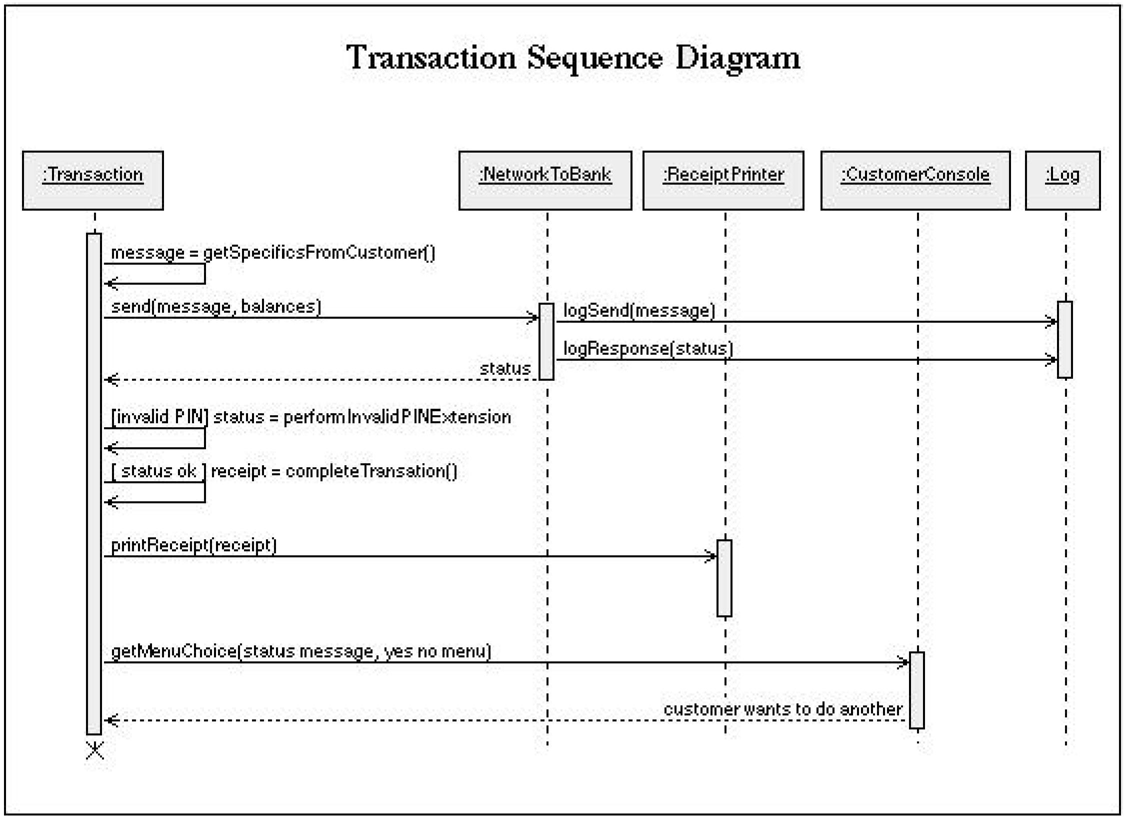

Transaction Sequence Diagram
:Transaction
:NetworkToBank
:ReceiptPrinter
:CustomerConsole
:Log
message = getSpecificsFromCustomer()
send(message, balances)
logSend(message)
logResponse(status)
status
[invalid PIN] status = performInvalidPINExtension
[status ok] receipt = completeTransation()
printReceipt(receipt)
getMenuChoice(status message, yes no menu)
customer wants to do another

Figure 22: Program Sample #10 Sequence Diagram

B.10.2 Converted Sequence Diagram

```
program10.Transaction main -> program10.Transaction getSpecificsFromCustomer()
       returns message
program10.Transaction main -> program10.NetworkToBank send(String, double) returns status
program10.NetworkToBank send -> program10.Log logSend(String)
program10.NetworkToBank send -> program10.Log logResponse(String)
program10.Transaction main -> program10.Transaction performInvalidPINExtension()
program10.Transaction main -> program10.Transaction completeTransaction()
program10.Transaction main -> program10.ReceiptPrinter printReceipt(String)
program10.Transaction main -> program10.CustomerConsole getMenuChoice(String, boolean)
       returns boolean
```

B.10.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**2.0**
71	**51**	**20**	**88**	**80**	**8**	

Table 12: Results for Sample Program #10

B.11 Program #11: An Address Book: Edit A Person

http://www.math-cs.gordon.edu/courses/cs211/AddressBookExample/index.html
Retrieved on November 27, 2007.

B.11.1 UML Diagram

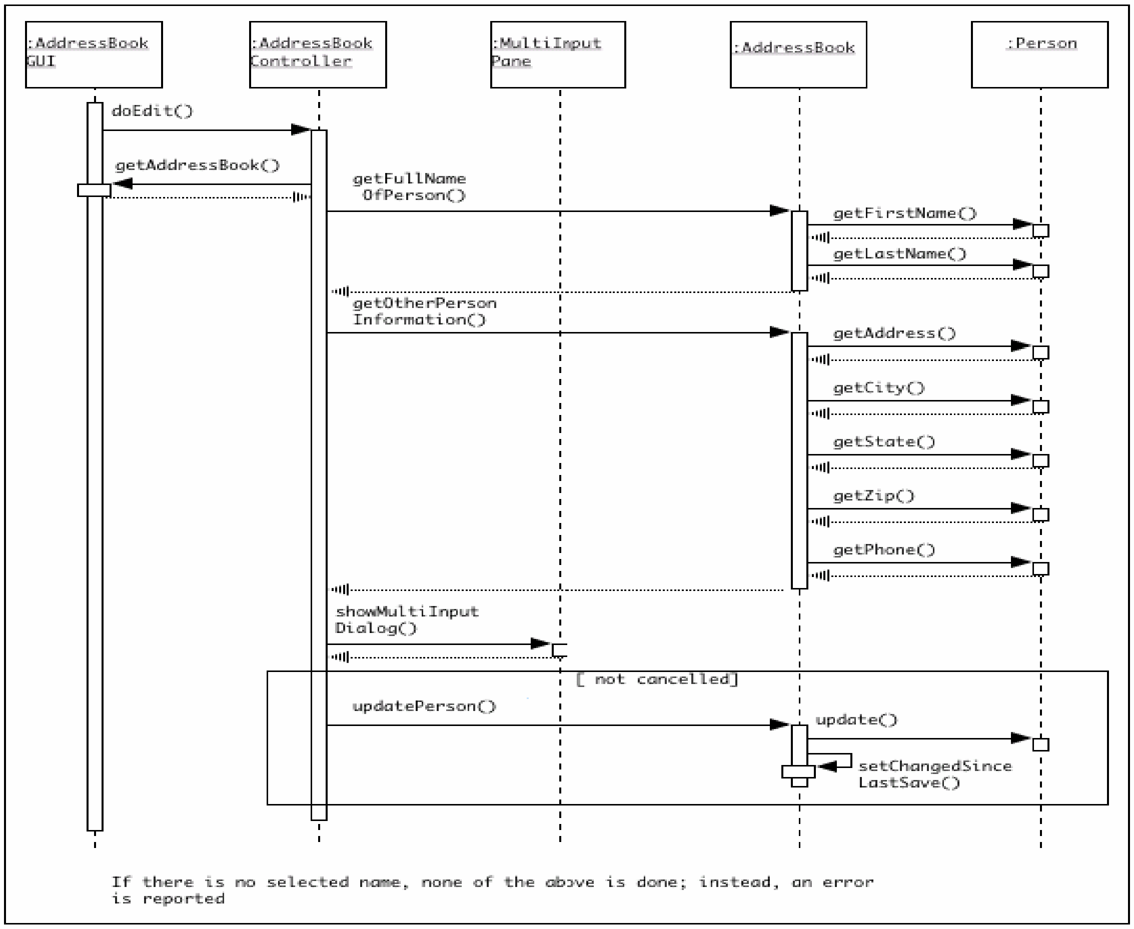


Figure 23: Program Sample #11 Sequence Diagram

B.11.2 Converted Sequence Diagram

```
program11.AddressBookGUI main -> program11.AddressBookController doEdit()
program11.AddressBookController doEdit -> program11.AddressBookGUI getAddressBook()
        returns book
program11.AddressBookController doEdit -> program11.AddressBook getFullNameOfPerson()
        returns fullName
program11.AddressBook getFullNameOfPerson -> program11.Person getFirstName()
        returns firstName
program11.AddressBook getFullNameOfPerson -> program11.Person getLastName()
        returns lastName
program11.AddressBookController doEdit
        -> program11.AddressBook getOtherPersonInformation() returns otherPersonInfo
program11.AddressBook getOtherPersonInformation -> program11.Person getAddress()
        returns address
program11.AddressBook getOtherPersonInformation -> program11.Person getCity()
        returns city
program11.AddressBook getOtherPersonInformation -> program11.Person getState()
        returns state
program11.AddressBook getOtherPersonInformation -> program11.Person getZip()
        returns zip
program11.AddressBook getOtherPersonInformation -> program11.Person getPhone()
        returns phone
program11.AddressBookController doEdit -> program11.MultiInputPane showMultiInputDialog()
        returns pane
program11.AddressBookController doEdit -> program11.AddressBook updatePerson()
program11.AddressBook updatePerson -> program11.Person update()
program11.AddressBook updatePerson -> program11.AddressBook setChangedSinceLastSave()
```

B.11.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**1.58**
56	**41**	**15**	**81**	**65**	**16**	

Table 13: Results for Sample Program #11

B.12 Program #12: Proxy Design Pattern – Java World

http://www.javaworld.com/javaworld/jw-02-2002/jw-0222-designpatterns.html
Retrieved on November 27, 2007.

B.12.1 UML Diagram

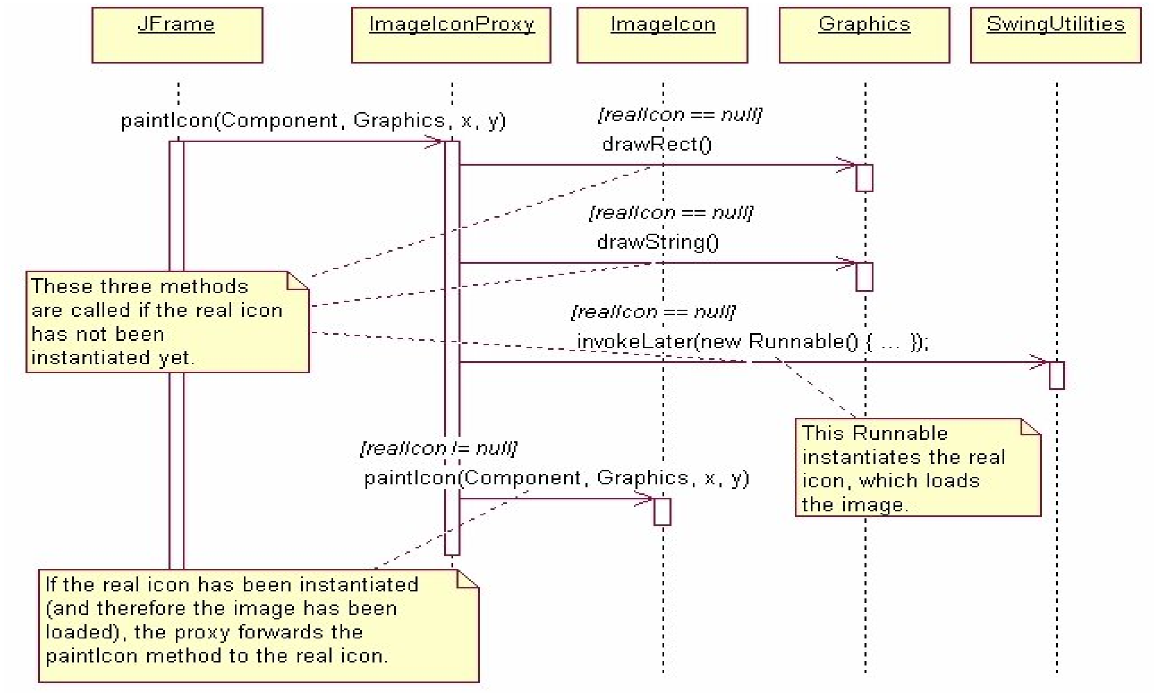

JFrame
ImageIconProxy
ImageIcon
Graphics
SwingUtilities
paintIcon(Component, Graphics, x, y)
[realIcon == null]
drawRect()
[realIcon == null]
drawString()
[realIcon == null]
invokeLater(new Runnable() { ... });
These three methods are called if the real icon has not been instantiated yet.
This Runnable instantiates the real icon, which loads the image.
[realIcon != null]
paintIcon(Component, Graphics, x, y)
If the real icon has been instantiated (and therefore the image has been loaded), the proxy forwards the paintIcon method to the real icon.

Figure 24: Program Sample #12 Sequence Diagram

B.12.2 Converted Sequence Diagram

```
program12.JFrame main
       -> program12.ImageIconProxy paintIcon(Component, Graphics, int, int)
program12.ImageIconProxy paintIcon -> program12.Graphics drawRect()
program12.ImageIconProxy paintIcon -> program12.Graphics drawString()
program12.ImageIconProxy paintIcon
       -> program12.SwingUtilities invokeLater(RealIconRunnable runnable)
program12.ImageIconProxy paintIcon -> paintIcon(Component, Graphics, int, int)
```

B.12.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With	Without	Final %	With	Without	Final %	**1.33**

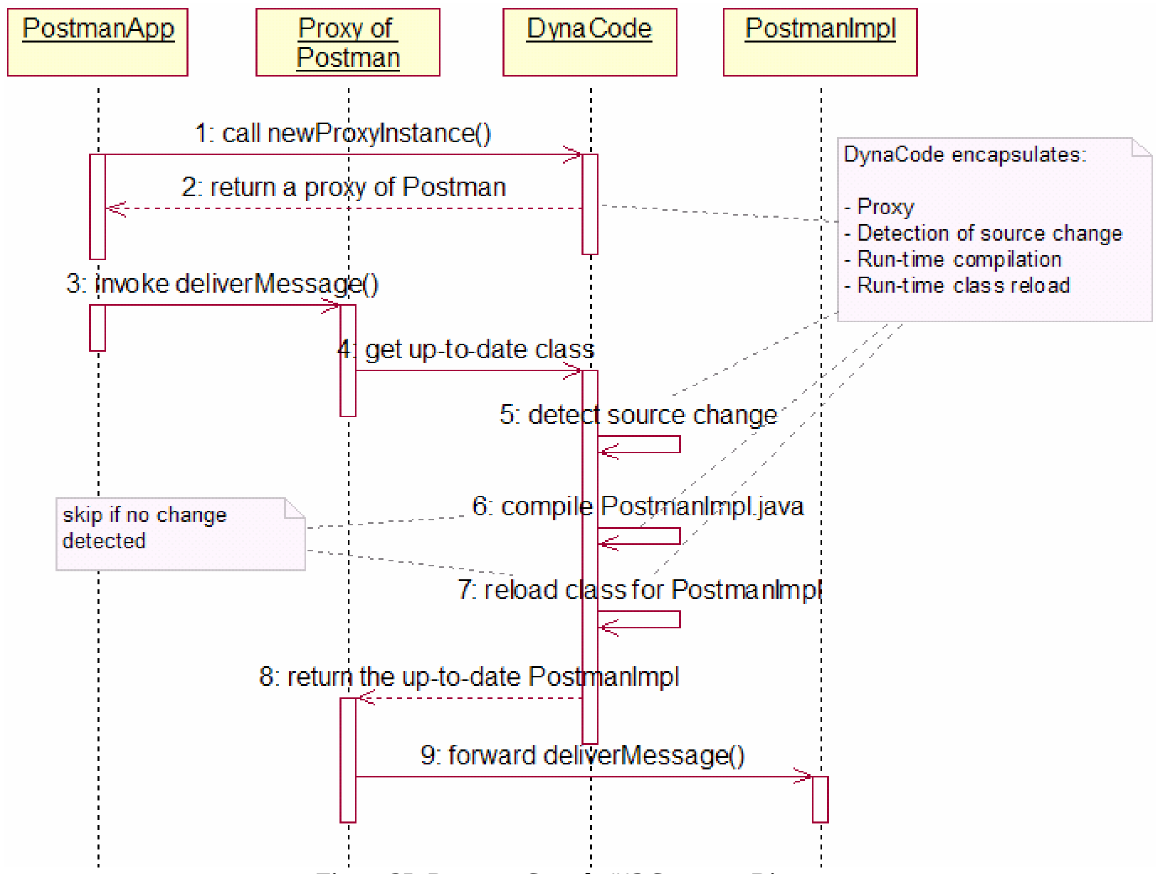
Design Knowledge	Design Knowledge	Coverage Increase	Design Knowledge	Design Knowledge	Coverage Increase	
83	**17**	**66**	**96**	**82**	**14**	

Table 14: Results for Sample Program #12

B.13 Program #13: Add Dynamic Java Code

http://www.javaworld.com/javaworld/jw-06-2006/jw-0612-dynamic.html?page=2
Retrieved on November 27, 2007.

B.13.1 UML Diagram



PostmanApp
Proxy of Postman
DynaCode
PostmanImpl
1: call newProxyInstance()
2: return a proxy of Postman
3: invoke deliverMessage()
4: get up-to-date class
5: detect source change
6: compile PostmanImpl.java
7: reload class for PostmanImpl
8: return the up-to-date PostmanImpl
9: forward deliverMessage()
DynaCode encapsulates:
- Proxy
- Detection of source change
- Run-time compilation
- Run-time class reload
skip if no change detected

Figure 25: Program Sample #13 Sequence Diagram

B.13.2 Converted Sequence Diagram

```
program13.PostmanApp main -> program13.DynaCode newProxyInstance() returns postManProxy
program13.PostmanApp main -> program13.PostManProxy deliverMessage(String)
       returns postManImpl
program13.PostManProxy deliverMessage -> program13.DynaCode getUpToDateClass()
program13.DynaCode getUpToDateClass() -> program13.DynaCode detectSourceChange()
program13.DynaCode getUpToDateClass() -> program13.DynaCode compilePostmanImpl()
program13.DynaCode getUpToDateClass() -> program13.DynaCode reloadPostmanImpl()
program13.PostManProxy deliverMessage -> program13.PostmanImpl forwardDeliverMessage()
```

B.13.3 Experiment Results

Branch Coverage % Achieved	Statement Coverage % Achieved	Cyclomatic Complexity

With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**1.25**
100	**100**	**0**	**100**	**100**	**0**	

Table 15: Results for Sample Program #13

B.14 Program #14: Hangman Milestone 2 – Java GUI

http://www.owlnet.rice.edu/~comp202/06-fall/labs/lab02/
Retrieved on November 27, 2007. Second UML Diagram on the page, entitled "Correctly guess a character and win."

B.14.1 UML Diagram

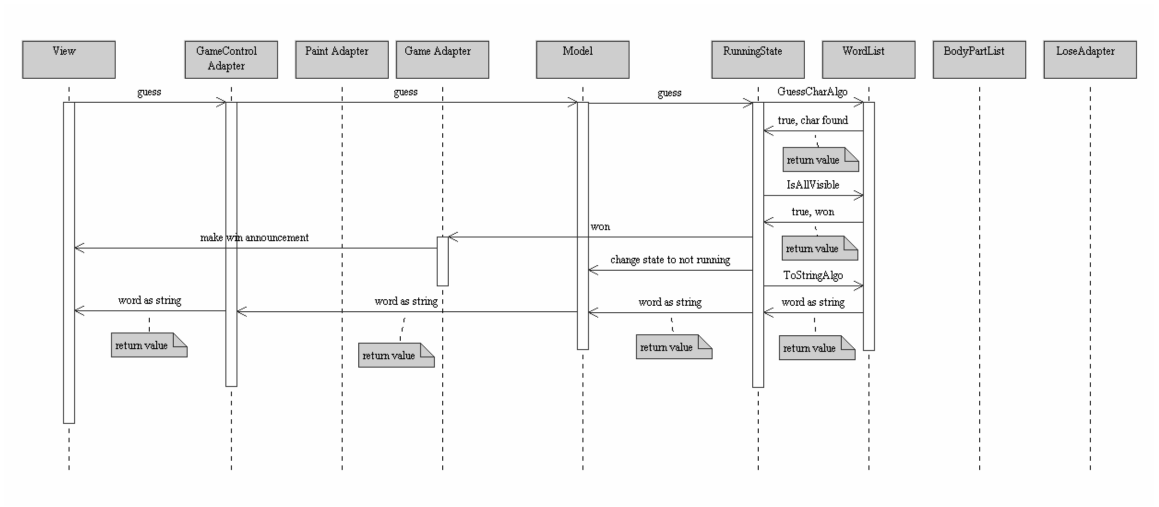


Figure 26: Program Sample #14 Sequence Diagram

B.14.2 Converted Sequence Diagram

```
program14.View main -> program14.GameControlAdapter guess()
program14.GameControlAdapter guess -> program14.Model guess()
program14.Model guess -> program14.RunningState guess()
program14.RunningState guess -> program14.WordList guessCharAlgo() returns bool
program14.RunningState guess -> program14.WordList isAllVisible() returns visibleBool
program14.RunningState guess -> program14.GameAdapter won()
program14.GameAdapter won -> program14.View winAnnouncement()
program14.RunningState guess -> program14.Model stateRunning(boolean)
program14.RunningState guess -> program14.WordList toStringAlgo() returns word
```

B.14.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**1.38**
70	**40**	**30**	**91**	**87**	**4**	

Table 16: Results for Sample Program #14

B.15 Program #15: Core J2EE Patterns – Intercepting Filter

http://java.sun.com/blueprints/corej2eepatterns/Patterns/InterceptingFilter.html
Retrieved on November 27, 2007.

B.15.1 UML Diagram

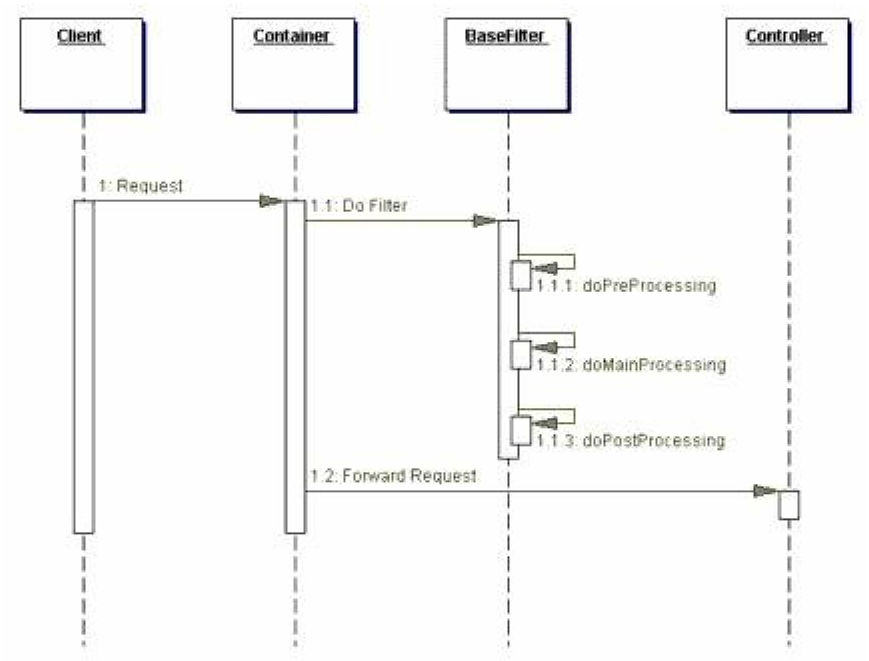


Figure 27: Program Sample #15 Sequence Diagram

B.15.2 Converted Sequence Diagram

```
program15.Client main -> program15.Container request()
program15.Container request -> program15.BaseFilter doFilter()
program15.BaseFilter doFilter -> program15.BaseFilter doPreProcessing()
program15.BaseFilter doFilter -> program15.BaseFilter doMainProcessing()
program15.BaseFilter doFilter -> program15.BaseFilter doPostProcessing()
program15.Container request -> program15.Controller forwardRequest()
```

B.15.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**2.182**
78	**50**	**28**	**98**	**90**	**8**	

Table 17: Results for Sample Program #15

B.16 Program #16: Hangman Milestone 2 – Java GUI

http://www.owlnet.rice.edu/~comp202/06-fall/labs/lab02/
Retrieved on November 27, 2007. Fifth UML Diagram on the page, entitled "The game is lost when the noose is drawn."

B.16.1 UML Diagram

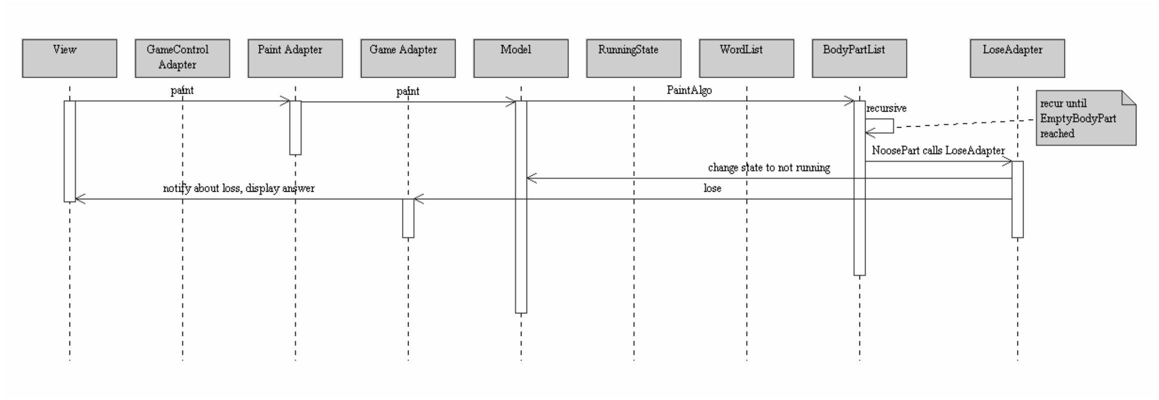


Figure 28: Program Sample #16 Sequence Diagram

B.16.2 Converted Sequence Diagram

```
program16.View main -> program16.PaintAdapter paint()
program16.PaintAdapter paint -> program16.Model paint()
program16.Model paint -> program16.BodyPartList paintAlgo()
program16.BodyPartList paintAlgo -> program16.LoseAdapter lose()
program16.LoseAdapter lose -> program16.Model stateRunning(boolean)
program16.LoseAdapter lose -> program16.GameAdapter lose()
program16.GameAdapter lose -> program16.View displayAnswer()
```

B.16.3 Experiment Results

Branch Coverage % Achieved			Statement Coverage % Achieved			Cyclomatic Complexity
With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	With Design Knowledge	Without Design Knowledge	Final % Coverage Increase	**1.42**
68	**41**	**27**	**91**	**87**	**4**	

Table 18: Results for Sample Program #16