

# Automated Feedback for “Fill in the Gap” Programming Exercises

**Nghi Truong, Paul Roe and Peter Bancroft**

Faculty of Information Technology  
Queensland University of Technology  
GPO Box 2434, Brisbane QLD 4001, Australia

nk.truong@student.qut.edu.au, {p.roe, p.bancroft}@qut.edu.au

## Abstract

Timely feedback is a vital component in the learning process. It is especially important for beginner students in Information Technology since many have not yet formed an effective internal model of a computer that they can use to construct viable knowledge. Research has shown that learning efficiency is increased if immediate feedback is provided for students. Automatic analysis of student programs has the potential to provide immediate feedback for students and to assist teaching staff in the marking process. This paper describes a “fill in the gap” programming analysis framework which tests students’ solutions and gives feedback on their correctness, detects logic errors and provides hints on how to fix these errors. Currently, the framework is being used with the Environment for Learning to Programming (ELP) system at Queensland University of Technology (QUT); however, the framework can be integrated into any existing online learning environment or programming Integrated Development Environment (IDE).

*Keywords:* automated testing, Java, C#, dynamic analysis, fill in the gap, XML, black box, white box, instant feedback.

## 1 Introduction

Learning to program is a difficult process. Programming is not a single skill but a multi-layered hierarchy of skills, many layers of which need to be active at the same time. Programming demands a great deal of implicit knowledge which is difficult for lecturers to make explicit and which cannot easily be transmitted directly to students. In order to gain knowledge and become competent in the domain, students need to go beyond explicit information to construct experiential implicit knowledge (Affleck and Smith, 1999). Programming cannot be learnt without doing a lot of practice.

When learning to program, it is essential that students are given the opportunity to practice in an environment where they can receive constructive and corrective feedback (Ben-Ari, 2001). Feedback is acknowledged as an important factor in the learning process especially when

available on request. However, with large class sizes, it is difficult for teaching staff to synchronise their heavy schedules to provide additional help when the students need it.

According to Affleck and Smith (1999), one of the main difficulties for beginning programmers is to access prior knowledge and able to apply this knowledge to new situations. Research has shown that the use of “fill-in the gap” programming exercises is one of the best ways to overcome the above problems (Lieberman, 1986). The supplied skeleton code in the exercises is generally based on known concepts; the missing code may contain new concepts. Students need to understand the given code in order to complete an exercise. They incorporate the knowledge provided in the exercise skeleton with the new knowledge to complete the exercise providing a new solution. Thus, “fill-in the gap” programming exercises help to close the gap between existing knowledge and new knowledge (Van Merriënboer and Paas, 1990). Furthermore, gap filling exercises reduce the complexity of writing programs. The majority of novice programmers have difficulty in starting their programs because they are not used to thinking in an abstract way; they cannot convert the text of a question into pseudo-code or code. If a partial solution to a programming problem is provided for the students, they get a better understanding of the exercise requirements and have a better chance of supplying the correct answer (Norcio, 1980). According to the chunking hypothesis, the first element of a chunk provides the key to the contents of the entire unit (Miller, 1956). Thus the fill in the gap approach builds up the students’ confidence, improves motivation and engages them more actively in the learning process.

The contribution of this paper is to describe a dynamic component of a program analysis framework intended for beginning students’ Java and C# programs. Analysing “fill-in the gap” exercises is a novel aspect of the framework which makes it distinctive from previous related work. The framework can be used for both tutoring and semi-automatic marking purposes. It carries out both black box and white box tests to provide feedback about the correctness of students’ solutions as well as identifying logic and runtime errors and their possible causes. The key characteristics of the dynamic analysis framework are its extensibility and configurability. Although the framework can be used as a separate tool, it is currently integrated into the ELP system which provides web based “fill in the gap” exercises.

The framework affords benefits to both students and teaching staff. While it is not able to completely replace

instructors or tutors, it helps students to learn in an environment where formative feedback and correct solutions can be obtained immediately. Students are able to access as much tuition as they need at their own pace; they are not limited to standard working hours or a particular location by having to come to university to consult teaching staff about their tutorial work. Students' programs are assessed by executing them against a set of test inputs, comparing the outputs with the expected outputs and giving feedback. With the analysed result provided by the framework, instructors only need to add comments which focus on specific aspects of the student's work; thus the marking task is less time consuming and laborious.

This paper is organized into seven sections. In Section 2, the paper presents an overview of the ELP system and the program analysis framework. In the following sections, it describes the design, current implementation status and limitations of the dynamic analysis framework with reference to some concrete examples. Related work is mentioned in Section 6. Future extensions and improvements are briefly described in Section 7.

## 2 Background: The ELP

The ELP is a web-based programming environment. It currently supports Java, C# and C programming exercises. Students are presented with program template exercises as web pages. They complete an exercise and submit their answer to the server for compilation. The executable format of the exercise is downloaded and run on the student's own machine. With Java programming exercises, the resulting .class file of the exercise is packed together with other necessary libraries in a Java Archive (JAR) file; .exe files are used for C and C# exercises. Figure 1 illustrates the integration between the ELP system and the program analysis framework.

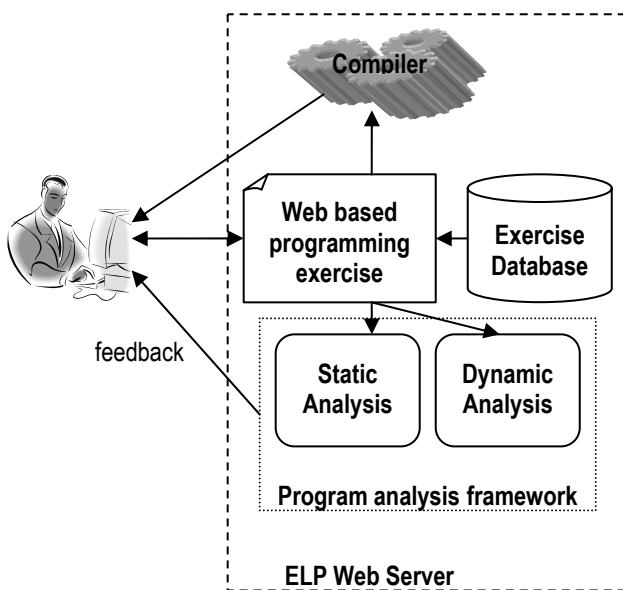


Figure 1: The ELP and the program analysis framework integration

The ELP provides “fill-in the gap” exercises to help beginning programmers to successfully write their

programs at an early stage of the learning process. Gaps in the ELP system can be as small as an expression or as big as a complete class. An example of a “fill-in the gap” exercise is shown in Figure 2 in which the gap code is shaded; only this gap code can be edited.

```

import TerminalIO.*;

public class DisplayName {
    KeyboardReader reader = new
        KeyboardReader();
    ScreenWriter writer = new ScreenWriter();
    public void run() {
        writer.println("Hi, my name is Mary");
    }
    public static void main (String [] args)
    {
        DisplayName tpo = new DisplayName();
        tpo.run();
    }
}
  
```

Figure 2: An example of a fill in the gap exercise

The program analysis framework consists of two separate components: static analysis and dynamic analysis. Static analysis is the process of analysing source code without executing it whereas dynamic analysis involves executing a student's solution through a set of test data. The two analyses are orthogonal to each other to provide more helpful feedback for students and teaching staff. The static analysis of the framework is used to check on the quality of students' solutions while the dynamic analysis is used to test the correctness of their solutions. The static analysis can be carried out both for gaps and whole programming exercises. Neither of these analyses can be used to diagnose a program which has syntax errors; programs must compile first.

The key features of the programming analysis framework are its configurability and extensibility. Analyses are provided as a set of functions so that instructors can specify which analyses should be carried out for each gap in an exercise. Additional analyses can be easily plugged-in at runtime. Unlike other systems, the framework provides both positive and negative feedback to students as we believe that positive feedback plays an important role in engaging students in the learning process.

Figure 3 gives an overview of the program analysis framework. The main focus of this paper is to describe in detail the design and implementation of the dynamic analysis component of the framework. The static analysis framework is discussed in (Truong et al., 2004).

## 3 The Dynamic Analysis Design

This section first describes how students can perform dynamic testing of their programs on the ELP system. We then consider the different types of beginning students programming exercises which played a major role in designing the framework. The design of both black box and white box tests are described.

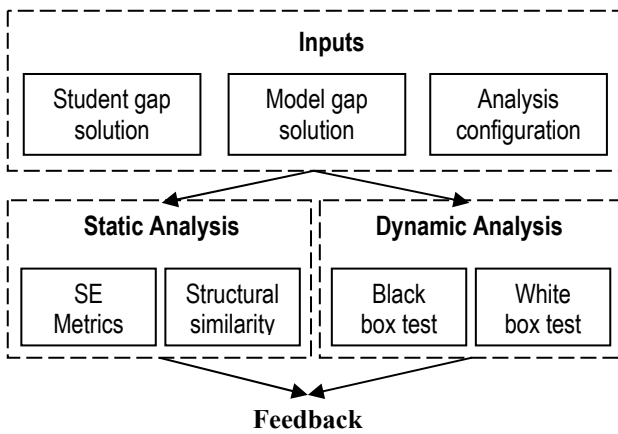


Figure 3: An overview of the program analysis framework

### 3.1 Overview

When a Java ELP exercise is completed and compiled, students submit the exercise to the ELP web server for testing. The resulting .class file of the exercise is packaged together with test drivers, test configuration and test inputs in a JAR file and subsequently downloaded and run on the student's own machine. When all tests have successfully run, outputs are sent back to the server for comparison with those generated by executing the exercise's model solution with the same set of inputs.

The framework incorporates black box and white box tests. Black box testing is the process of testing a program against its specification with testers having no knowledge of the implementation of the program. This knowledge is however required in white box testing. Black box testing is carried out by executing students' programs through a set of test data; gap outputs are captured and sent back to the server to check for correctness. White box testing is carried out on a student's solution, one gap at a time. Outputs of each gap are compared with the outputs produced by the corresponding gap solution. The feedback to the student indicates any lack of functionality in their solution, whether they have obtained the correct results and logic errors and their possible causes. More importantly, the framework also provides hints on how to fix those problems.

### 3.2 Beginning Students' Programming Exercises

To ensure that the framework is useful for students, we collected student programming exercises in three introductory subjects in four consecutive semesters. These exercises can be grouped into three categories: console, object oriented and Graphical User Interface (GUI) exercises. These categories and their subtypes are shown in Table 1.

As mentioned earlier, analysing "fill in the gap" exercises is the novel aspect of the framework and therefore it is crucial for us to have an understanding of all possible gap types. Analysis of the collected exercises revealed five major gap types which vary from as small as an expression to as big as multiple methods. Gap types

include expression gaps, declaration gaps, statement gaps, block of statements gaps and gaps which are a complete method.

Exercise Categories
<b>Console</b> <ul style="list-style-type: none"> <li>only modify the format of the output.</li> <li>only perform arithmetic or string literal operations and display the result.</li> <li>have loop and conditional statements.</li> <li>make use of arrays.</li> <li>perform file input and output.</li> </ul>
<b>Object Oriented</b> <ul style="list-style-type: none"> <li>concerned with access modifiers</li> <li>with user defined types</li> </ul>
<b>Graphical User Interface</b> <ul style="list-style-type: none"> <li>a user interface is provided.</li> <li>a user interface is not provided.</li> </ul>

Table 1: Exercise types used in introductory programming courses

### 3.3 Black Box Testing

The black box testing of the dynamic analysis framework is designed to check if a student's program behaves correctly, as specified in the exercise requirements. It is carried out by executing the student's completed program and the model solution with the same test inputs. The outputs are then compared with each other. As skeleton code is provided for all fill in the gap exercises, it is only necessary to compare the output produced by the gaps. Special print statements are inserted at the start and end of each gap to mark the outputs so that the individual gap outputs can be extracted and analysed separately. The feedback to the student indicates which tests do not yield a correct result and suggests possible causes. Differences in formatting of the student program outputs and the model solution outputs are also reported. Figure 4 gives an overview of black box testing.

According to Jackson (1991), complete automation of the program grading process needs to have five important properties. Firstly, there must be a way to distinguish between important items that are relevant to the program correctness and those less important. Secondly, the system should be able to scope insignificant differences in style. Thirdly, the system should perform range checks for numerical items. Fourthly, the system should be able to cope with important items in the program output being in a different in order. Finally, the correctness of a program should be independent of its output format.

To satisfy the above properties, a set of filters and normalizers have been designed to process the outputs before the comparison process. These filters and normalizers make use of dynamic class loading so that additional filters and normalizers can be added in the future; thus the framework is extensible. Filters can be used to extract all the important keywords or values that are related to correctness and insert them into a set data structure. The student set and the model solution set may then be matched, taking order into account or not.

Normalizers allow the framework to compare the outputs of a student's solution with those of the model solutions while ignoring insignificant differences such as spaces or tabs. By doing that the framework has provided a mechanism to distinguish between important items that are relevant to the correctness of the solution and those that are irrelevant.

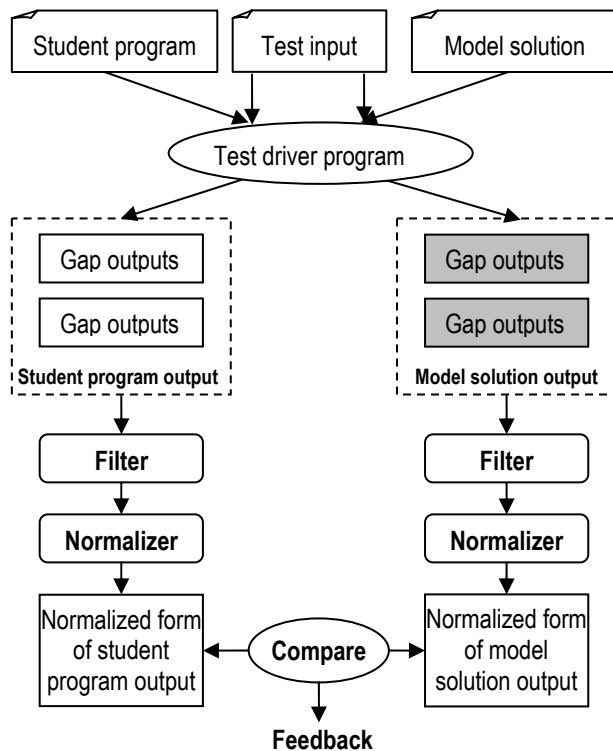


Figure 4: An overview of black box test

For example, the comparison process for exercises which involve arithmetic operations and displaying the output consists of two analysis steps: checking the results of the calculations and examining the output message. In order to first evaluate the correctness of the arithmetic, the student's program outputs and the model solution outputs are filtered against a set of rules to extract all the key values of the calculations. Then the two resulting sets of values are sent to the matcher to compare. Checking the format of the output can be as simple as exact match to as complex as normalizing space, filtering keywords and matching those keywords with a set of expected keywords.

The process of comparing test output is fully configurable by teaching staff which allows various correctness levels of a programming problem rather than just right or wrong. This also means that the framework is able to provide more detailed feedback for students. The feedback identifies which particular sections students have done correctly and which are incorrect so that they can easily locate errors in their program. Preston and Shackelford's (1999) research reveals a major reason for the lack of success for most existing automated grading systems - the lack of flexibility. Teaching staff are not able to adjust the marking criteria and use their own feedback which may better focus on a student's work rather than on the work being assessed. Making the

framework fully configurable gives teaching staff a great level of control over the feedback and the assessment process.

In order to enable black box testing, lecturers need to provide a good test plan which consists of test classes and test cases. They also need to configure filter, normalizer and matching rules in the comparison process. Additionally, they can set customized feedback for each test class. Test classes are normally used to partition the input domain of a program so that the programmer can assume that a particular test of a representative value of the class is equivalent to a test of any other values in the same class. Test cases of a test class are representative input values for that particular class. A good test case is one which has a high probability of making a faulty program fail. The main benefit of providing a good test plan is that these test cases can be re-used in the white box testing. Furthermore, it enables the testing process to be carried out in an adaptive manner. For example, if a student's program fails in a test class, certain white box tests will be carried out.

### 3.4 White Box Testing

The main reasons for white box testing are to discover any possible logic errors which are not revealed in black box testing or to detect gaps that have logic errors which lead to the incorrect outputs in black box testing. White box testing is carried out by inserting a student's gap solutions, one at a time into a test harness program. The outputs of each gap are compared with the outputs produced by the gap solution. This method allows the framework to be able to indicate which gaps are responsible for failed black box tests. This will provide beginner programmers with better guidance in debugging their programs.

In white box testing, gaps are considered as units of a program which cannot stand alone. Each unit is inserted into a test harness program so that it can be run. A test harness is a program skeleton which is used to test a unit that is dependent on it. The framework supports two different types of test harness program: a default test harness which is the exercise model solution and a customized one which is provided by teaching staff. If the default test harness program is used, white box testing makes use of a regression testing technique in which the student's gap solution is considered as a new change for the program. Unit testing is adopted when a customized test harness program is preferred.

White box testing requires that each gap be wrapped in adaptor code to provide more information. We identified two types of gap behaviour: gaps in which the state of variables are changed and gaps which only modify the output of the program. In order to accommodate these gap types, the framework provides three different types of adapter code each providing a mechanism to carry out white box testing. These mechanisms include: variable state dumps, program assertions and **print**. Variable state dumps and program assertions can be used to check the state of variables before and after gap execution while the **print** method is used to check those programs whose

gaps only modify the output. The state dump mechanism provides a set of functions to record the type and value of a variable at runtime. The program assertion mechanism can be used to test certain conditions that need to be met either before or after gap execution. The **print** mechanism provides markers to distinguish the output produced by each gap from the program outputs. The gap outputs are then extracted and analysed for changes.

Figure 5 illustrates steps in white box testing when the default test harness program is used. Each gap solution with its adaptor code (which is provided by the lecturer) is independently inserted into the exercise model solution to produce a mixed program. The mixed program is then compiled and executed as is the exercise model solution, using the same set of inputs. The output of the student's tested gap is compared with the outputs of the model solution gap to provide feedback to the student. The feedback incorporates the lecturer's customized feedback and the test results. This will report on the differences in the states of variables in the student's solution and gap model solution for gaps that have variables changing state at runtime. For gaps which only modify the program output, the feedback advises students on how similar their gap outputs are to the gap model solution outputs.

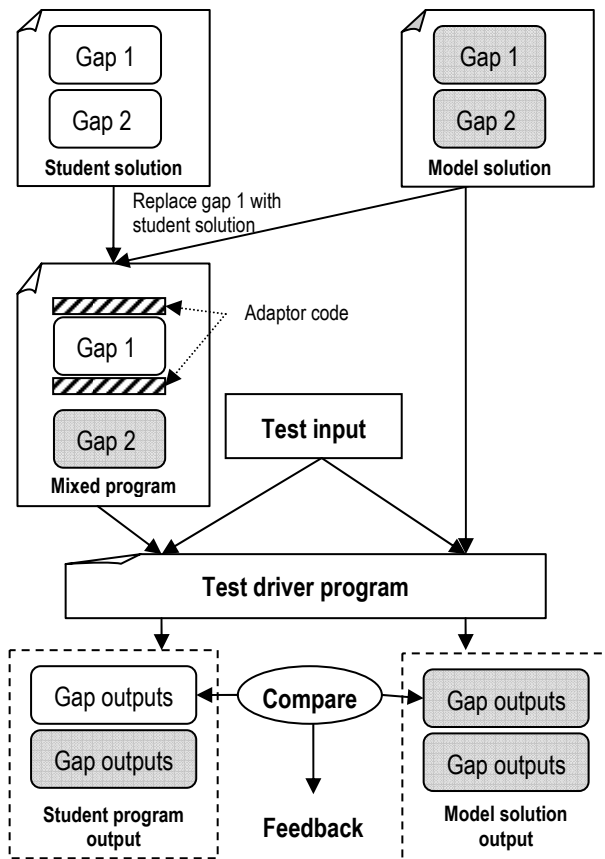


Figure 5: An overview of white box testing

White box testing is slightly different when a customized test harness program is used. Instead of producing the mixed program, two test harness programs are generated: one contains the student's gap solution and the other contains the gap model solution. These test harness programs are then run by the test driver.

The strength of this approach is that it provides the ability to isolate bugs in a particular gap. Unfortunately, it does not work well if students declare their own identifiers. In this case, gaps and code must be tested together.

To enable white box testing, lecturers need to provide adaptor code for variable state dumps, program assertions or **prints**. They also need to set up the matching rules for the comparison process. A majority of test inputs are re-used from black box testing; however additional test inputs can be applied. Optionally, lecturers can provide customized test harness programs when the default is not sufficient. Customized feedback can be applied to white box testing as well.

## 4 Implementation

This section reports on the technologies and techniques which are used to implement the framework and its current status.

### 4.1 The Client-Server Communication Mechanism

The dynamic analysis framework uses request and response messages to communicate between client and server. Request and response messages are marked up with XML and sent from client to server as a serialized object. Figure 6 shows the communication between the client and server of the dynamic analysis. Request and response messages are numbered to represent the creation sequence.

A **TestRequest** is sent to the server when a student submits their work for testing. The server checks which test configuration is set for the exercise; it packs all the necessary test driver classes, testing inputs and the .class file of the student's solution into a JAR which is subsequently sent back to the client as the payload of the **TestResponse** message.

When a JAR is executed, a driver program runs each separate black and white box test. After all tests are run, the TCP/IP server sends the results back to the ELP web server in an **TestOutputRequest** message then terminates. The driver uses a timeout to guard against infinite looping tests.

A **GetTestFeedbackRequest** is sent to the server when a student requests the result from the ELP exercise editor applet. The server replies with a **GetTestFeedbackResponse**. If the feedback is available, it is set as content in the response from the server which is then displayed on the analysis tab in the applet; otherwise a try again later message is displayed.

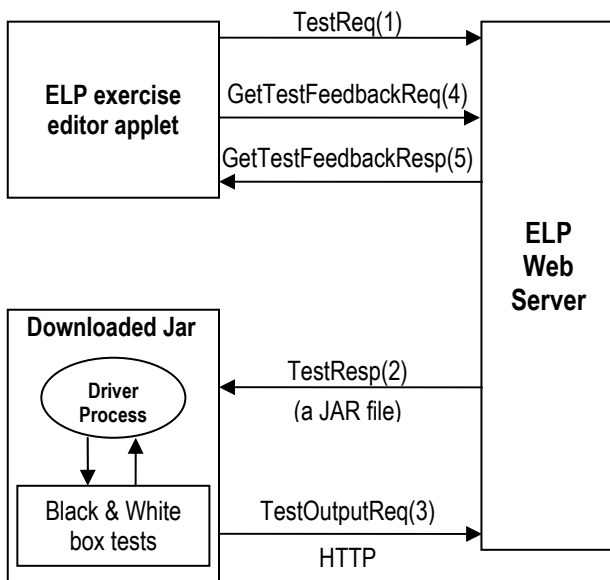


Figure 6: The communication between ELP and the dynamic analysis testing framework

## 4.2 The Black Box Test Implementation

In this test, a student program solution and the exercise model solution are invoked with the same test inputs. Gap outputs from the student solution are extracted to compare with the gap model solution outputs for correctness. Special `print` statements are inserted before and after gap code to mark the outputs produced by the gap. However, the end gap `print` statement might not always be displayed in student solution outputs for gaps which have unexpected `return`, `exit`, `break` and `continue` statements. In these cases, output from the start gap marker to the next start gap marker is compared. The delimiter is generated from the hash code of a gap solution, gap identifier and the current system time. This will ensure a different delimiter is produced not only for each gap in a program but also for gaps which have the same solutions as the constructed times are different. These delimiters have XML-like format. An open tag style delimiter is inserted at the beginning of the gap and the close tag style is inserted at the end of the gap.

A test driver program is used to invoke the student program through a sequence of tests according to the test plan specified by lecturers. At the beginning of each test, the standard input of the test driver is set to read from a text file which contains all the inputs for the test. Its standard output is forwarded to a customized stream which stores all students' program outputs for that test. This stream is sent to the TCP/IP server when the test finishes and finally all test outputs are sent back to the ELP web server for analysis through a HTTP connection.

When test outputs are received, individual gap outputs are first extracted. These outputs and gap model solution outputs are applied through a set of filter and normalizer rules to extract all important keywords or values; these keywords or values are then compared with each other to provide feedback for students. The output is filtered and normalized. The feedback states which tests do not yield

a correct result and which tests have a correct result but have incorrect format. In order to reduce analysis time, gap model solution outputs are filtered and normalized when lecturers configure tests; the result is stored on the server.

Currently there are two filters, two matchers and two normalizers to compare students' programs outputs with the model solution output. Table 2 summarises all filters, matchers and normalizers together with their sub-options. Dynamic class loading is used for the filters, matchers and normalizers; this makes the framework extensible as additional filters and matchers can be easily plugged in.

<b>Filters</b>
<ul style="list-style-type: none"> <li>• Number filter</li> <li>• Keyword filter               <ul style="list-style-type: none"> <li>○ Case sensitive</li> </ul> </li> </ul>
<b>Matchers</b>
<ul style="list-style-type: none"> <li>• Exact match</li> <li>• Match disregard the order</li> </ul>
<b>Normalizer</b>
<ul style="list-style-type: none"> <li>• Space or tab               <ul style="list-style-type: none"> <li>○ Leading and trailing space</li> <li>○ Space or tab between words</li> </ul> </li> <li>• New line character               <ul style="list-style-type: none"> <li>○ Leading and trailing new line character</li> </ul> </li> </ul>

Table 2: Functions provided to check the result of black box testing

The framework makes use of XML extensively. All testing configuration and analysis feedback is stored in XML on the server. The use of XML has brought several advantages to the framework including: easy to understand and manipulate, extensible, widely supported and human readable (Mamas and Kontogiannis, 2000).

## 4.3 The White Box Testing Implementation

White box testing is carried out by inserting the student's gap solutions and gap model solutions one at a time into test harness programs. The test harness program which contains a student's gap solution and the one which contains the gap model solution are then compiled and executed with the same set of test inputs. The student's gap outputs and the model gap outputs are compared.

The framework supports two types of test harness program: the default test harness program which is the exercise model solution and a customized one which is provided by teaching staff. If the default test harness program is used, multiple programs are generated with the same class name because a Java class needs to be stored in a file with the same name as the class name. The .class files of these test harness programs are packed into a JAR file in a directory structure to overcome the existence of multiple files with the same name in the JAR root directory and to maintain the hierarchical structure of the exercise. Although customized test harness programs have different names, the .class files of these programs are also stored in a directory structure so that the hierarchical structure of the exercise is maintained. Figure 7 gives an example of a directory structure in a JAR for

an exercise which has two gaps in a class when there is only one test carried out for each gap.

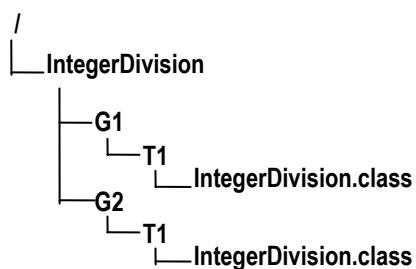


Figure 7: An example of the directory structure of a JAR

There are three different mechanisms which can be used to perform white box testing: state dumps, program assertions and **print**. State dumps and assertions are used to track the change of variables in gaps while the **print** approach is used to separate gap outputs from the test harness program outputs. There will be no additional outputs apart from the gap outputs with a customized test harness. In contrast, where the exercise solution is used as a test harness, test output and program output will be mixed. For this reason, the **print** mechanism is used to differentiate the outputs. The output of the gap is extracted and compared with the output of the model gap solution using the same matching mechanism as for black box testing. As with black box tests, the **print** statement which marks the end of gap output might be missing when students make unexpected use of `return`, `exit`, `break` and `continue` statements in their solution. This problem is overcome by comparing the output from the start gap marker to the end of the outputs.

A **StateDump** class is implemented to provide a set of overloaded **dump** methods which dump out the state of different variables at run time. This class also provides a set of overloaded **dumpExpression** methods to dump out the values of different expressions. Java assertions are used in the assertion mechanism to test certain conditions that need to be met either before or after gap execution.

States of variables in a program need to be serialized XML string to be sent from the client to the server. This allows the framework to perform dynamic analysis for any programming language. There are four possible types of variables that can appear in a program: primitive type, reference type, array type (either primitive or user defined object) and user defined type. Reflection is used to record information of variables. With a user defined type, the recorded information includes: object type, classes and interfaces that the user defined type extends and implements and all fields in the object's class. Modifiers, type and the name and value of a field are recorded. Teaching staff can choose to set a deep serialization which will recursively serialize a user defined class or array of user defined class which is referenced in the current class. Each element of an array of user defined type is serialized.

Comparing the value of serialized objects is flexible and extensible. Teaching staff are able to check on the type, format and range of a numerical value. With an array type

variable, it is possible to compare only values in the array regardless of an array elements' position. With variables of user defined type, teaching staff can choose to check only fields whose values would be changed after gap execution.

#### 4.4 Framework Limitations

Currently the framework has two main limitations. Firstly, it is not able to analyse GUI interface exercises. Secondly, exercises which perform file input and output operation can not be checked for correctness.

### 5 Examples

The following example shows how to configure black box and white box tests for an exercise. The exercise has two gaps and the states of the variables are changed after gap two is executed thus the state dump mechanism is used to carry out white box testing. Figure 8 shows the exercise skeleton with the solution in the gaps.

Figure 9 gives the black box test configuration. The exercise requires students to perform two arithmetic operations thus the comparison process of black box testing comprises of two steps: check the calculation result and check the format of the output. Since all the prompt messages in this exercise are provided by teaching staff, only the values of the calculations for the student program are compared with the model solution. All the numbers from gap outputs are first filtered; these numbers are then compared with a list of numbers extracted from the gap solution without regard to order but they need to be the same value and format.

In this example, the "in-line" attribute is set to true in the white box test configuration which means the default test harness program is used; otherwise an external program is used. Two **IntegerDivision** classes are constructed, each containing a student's solution for gap one and gap two. These classes are stored in a directory structure as shown in Figure 7. The "type" attribute is used to determine where the adaptor code is inserted in relation to the gap by setting a value of pre, post or pre-post. In this example the type set to post and the adaptor code is inserted at the end of the gap. Each test is associated with one or more sets of input and output data. One or more gaps can be set as dependent on other gaps, which means they will need to be successfully tested before the parent can be tested. If one of the dependent tests fails, the parent gap is marked as failing the white box test. This helps to locate errors in student's codes more accurately. An example of a white box test configuration is shown in Figure 10.

#### Question:

*Write a program that takes two integers as inputs and displays their quotient and remainder. Do not assume that the integers are entered in any particular order, but be sure to divide the larger integer by the smaller integer.*

```

import TerminalIO.*;

public class IntegerDivision {

    KeyboardReader reader = new
        KeyboardReader();
    ScreenWriter writer = new ScreenWriter();

    public void run() {

        int value1, value2;
        int largest, smallest;
        int quotient, remainder;

        // get user inputs
        value1 = reader.readInt("Enter the
            first integer: ");
        value2 = reader.readInt("Enter the
            second integer: ");

        // find the largest and smallest
        if (value1 > value2) {
            largest = value1;
            smallest = value2;
        } else {
            largest = value2;
            smallest = value1;
        }

        // do calculations
        quotient = largest / smallest;
        remainder = largest % smallest;

        // output results
        writer.println(largest + " divided by
            " + smallest + " = " + quotient +
            " remainder " + remainder);
    }

    public static void main (String [] args)
    {
        IntegerDivision tpo = new
        IntegerDivision();
        tpo.run();
    }
}

```

Figure 8: A fill in the gap exercise with a model solution

```

<blackbox>
<testclass id="1" name="Largest value
first">
<testcase id="1.1">
    <input filePath="inblackbox1.1.txt"/>
    <output filePath="outblackbox1.1.txt"/>
    <compare-options>
        <filters><NumberFilter/></filters>
        <matches>
            <matching
                order="disregard">exact</matching>
            </matches>
        </compare-options>
        <feedback/>
    </testcase>
<testcase id="1.2">
    <input filePath="inblackbox1.2.txt"/>
    <output filePath="outblackbox1.2.txt"/>
    <compare-options>
        <filters><NumberFilter/></filters>
        <matches>
            <matching>exact</matching>
        </matches>
        </compare-options>
        <feedback/>
    </testcase>
</testclass>
</blackbox>

```

Figure 9: An example of a black box test configuration

```

<white-box>
<class name="IntegerDivision">
<gap id="1">
<dependency/>
<test name="1"
    description="Check largest and smallest">
    <driver in-line="true" type="post-code">
    <post-code><![CDATA[
        DumpState.dump(largest,"largest.so");
        DumpState.dump(smallest,"smallest.so");
    ]]></post-code>
    </driver>
    <testdata>
    <set id="1">
    <input filePath="inwhitebox1.1.1.txt"/>
    <output filePath="outwhitebox1.1.1.txt"/>
    </set>
    </testdata>
    <feedback/>
    </test>
</gap>
<gap id="2" declarativeGap="no">
<dependents/>
<test name="1"
    description="Check the calculation">
    <driver in-line="true" type="post-code">
    <post-code><![CDATA[
        DumpState.dump(quotient,"quotient.so");
        DumpState.dump(remainder,"remainder.so");
    ]]></post-code>
    </driver>
    <testdata>
    <set id="1">
    <input filePath="inwhitebox1.2.1.txt"/>
    <output filePath="outwhitebox1.2.1.txt"/>
    </set>
    </testdata>
    <feedback/>
    </test>
</gap>
</class>
</white-box>

```

Figure 10: An example of a white box test configuration

```

if (value1 > value2) {
    largest = value1;
    smallest = value2;
} else {
    largest = value2;
    smallest = value1;
}
DumpState.dump(quotient, "largest.so");
DumpState.dump(remainder, "smallest.so");

```

Figure 11: The first gap solution and its adaptor code

The main aim of white box testing for the first gap is to ensure the conditional statement is correct so that the correct largest and smallest values are returned. The states of the largest and smallest are changed from 0 which is the default initialized value of an integer in Java to some values that are different from 0 after the gap is executed. The state dump mechanism is used to record the states of largest and smallest variables in the student's program in **largest.so** and **smallest.so** serialized objects accordingly. These objects are sent back to the server for comparison with the state of largest and smallest variables in the model solution. Figure 11 gives an example of the first gap solution and its adaptor code.

Similarly, the state of the quotient and remainder variables in the student's program are recorded in the **quotient.so** and **remainder.so** serialized objects which are subsequently sent back to the server to check if the calculation is correct.

## 6 Related Work

Automatic grading systems are economical and effective. This kind of system reduces the workload for instructors and improves the student's learning experience by providing instant feedback. Because of these benefits, widespread research has been carried out to develop automatic grading systems. However, few systems support the analysis of "fill in the gap" programming exercises.

CourseMaster (CourseMaster, 2000), WebToTeach (Arnou and Barshay, 1999), the automatic grader (Morris, 2002) and datlab (MacNish, 2000) are systems that have had an impact on the design of the matching mechanism in this program analysis framework.

Course Master is a client server system for delivering course based programming. It provides functions for automatic assessment of students' work, administration of the resulting marks, solutions and course materials. Student can submit their work to the server for an oracle to check the program correctness which involves a number of regular expressions to define the structures it expects to find in the student program's output. For each set of test data, the teacher provides a set of regular expressions to recognize required features of the output.

WebToTeach is a web based automatic homework checking tool for computer science classes. It supports Java, C, C++, Fortran, Ada and Pascal and incorporates various types of exercises including writing a code fragment, writing data for a test suit, writing a complete single source program and writing several source files.

Students are given either a single text area or multiple text areas on the web browser to provide the solution depending on the exercise type. Upon submitting the solution for the exercise, students are told immediately whether it is correct. In the case of failure the student is given information about the cause of failure. The outputs from the student program are compared with a model solution in 3 modes with additional options. The space comparison mode has three options: exact comparison, map sequence to a single space and strip all white space. Exact comparison and eliminate empty lines are available options for the line comparison mode. The student program output can also be compared with the model solution output in case sensitivity mode.

An automatic grading system for Java programming assignments has been developed at Rutgers University to be used in the introductory Computer Science course. The system make uses of Java reflection classes, Java inheritance mechanism and Perl regular expressions. Java Reflection is used to find and execute a student program's methods. The Java inheritance mechanism allows a defective student method to be overridden with a known good method so the evaluation can continue. Perl regular expressions are used to check the program outputs and source code for desirable or undesirable coding patterns.

The datlab system has been developed at University of Western Australia for monitoring student progress in computer science laboratories and providing timely feedback on their work. The datlab system runs on a server and continuously polls for requests at regular intervals. Students submit requests to the system by emailing the lecturer's account with datlab as the subject line. These emails are then filtered and appended to the requests line of the datlab system. A new thread is created to handle each request. When the system finishes analysing the student's work, student records are updated and a report is mailed back. The system makes use of Java technologies to enable the process of running and analysing a student's work. These technologies include: reflection, class loading and the runtime environment. Code analysis relies on syntactic parsing and Java checking mechanisms for compilation, loading and execution. The process of comparing a student solution against a model solution is done by the lecturer.

## 7 Conclusions and Future Work

In summary, the dynamic analysis framework is able to analyse "fill in the gap" Java programming exercises. The framework makes use of client-server communication architecture where the execution of students' programs takes place on the students' own machines while the correctness evaluation is carried out on the server. The framework consists of black box and white box testing. With black box testing, a set of filters, normalizers and matchers are provided to compare the output of students' gap solutions with model solution output. White box testing supports two types of test harness program: a default test harness which is the exercise model solution and a customized one which is provided by teaching staff. Three different mechanisms are provided to carry out white box tests including variable state dumps, program

assertions and **print** to accommodate two types of gap: gaps in which the state of variables are changed and gaps which only modify the output of the program. Variable state dumps and program assertions can be used to check the states of variables before and after gap execution while the **print** method is used to check those programs whose gaps only modify the output. Lecturers can set customized feedback in both black box and white box tests.

The framework currently has two limitations. Firstly, it is not able to test for the correctness of exercises which perform file input and output operations. Secondly, it is not able to analyse GUI interface exercises.

In the future, more filters and matchers will be added. An interface for assisting staff to configure tests for an exercise will also be developed.

Last but not least, an evaluation of the framework in a class of 400 students has been scheduled for first semester 2005 to coincide with the introductory programming course at QUT. Students will be required to complete their first five weeks tutorial exercises on the ELP system. In order to ensure a majority of students will participate in the evaluation, five of these tutorial exercises are hands-in exercises which contribute ten percent of their unit. In week five of the semester, questionnaire forms will be distributed to obtain feedback from students. In addition, the framework is being continuously evaluated by teaching staff in the faculty and consistently receives positive feedback.

## 8 References

Affleck, G. and Smith, T. (1999): Identifying a need for web-based course support. *Proc. Conference of the Australasian Society for Computers in Learning in Tertiary Education*, Brisbane, Australia, Online.

Arnow, D. and Barshay, O. (1999): WebToTeach: An Interactive Focused Programming Exercise System. *Proc. 29th ASEE/IEEE Frontiers in Education Conference, San Juan, Puerto Rico*, 12a9-39, IEEE.

Ben-Ari, M. (2001): Constructivism in Computer Science Education. *Journal of Computers in Mathematics & Science Teaching* **20**(1):24-73.

CourseMaster, School of Computer Science & IT, the University of Nottingham, UK. [http://www.cs.nott.ac.uk/CourseMaster/cm\\_com/index.html](http://www.cs.nott.ac.uk/CourseMaster/cm_com/index.html). Accessed 29 Dec 2002.

Jackson, D. (1991): Using software tools to automate the assessment of student programs. *Journal of Computers & Education* **17**(2):133-143.

Lieberman, H. (1986): An Example Based Environment for beginning programmers. *Journal of Instructional Science* **14**(3):277-292.

MacNish, C. (2000): Java Facilities for Automating Analysis, Feedback and Assessment of Laboratory Work. *Journal of Computer Science Education* **10**(2):147-163.

Mamas, E. and Kontogiannis, K. (2000): Towards Portable Source Code Representations Using XML. *Proc.*

*Seventh Working Conference on Reverse Engineering*, Brisbane, Australia, 172-182, IEEE Press.

Miller, G. A. (1956): The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information. *Psychological Review* **63**:81-97.

Morris, D. S. (2002): Automatically grading Java programming assignments via reflection, inheritance and regular expressions. *Proc. 32nd ASEE/IEEE Frontiers in Education Conference*, Boston, MA., 1: T3G-22, IEEE.

Norcio, A. F. (1980): Human Memory Processes for Comprehending Computer Programs. *Proc. IEEE Systems, Man and Cybernetics Society*, Cambridge, Massachusetts, 974-977, IEEE.

Preston, J. A. and Shackelford, R. (1999): Improving on-line assessment: an investigation of existing marking methodologies. *Proc. The 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and Technology in Computer Science Education*, Cracow, Poland, 29-32, ACM Press.

Truong, N., Roe, P. and Bancroft, P. (2004): Static Analysis of Students' Java Programs. *Proc. Sixth Australasian Computing Education Conference*, Dunedin, New Zealand, **30**: 317-325, Australian Computer Society Inc.

Van Merriënboer, J. J. G. and Paas, F. G. W. C. (1990): Automation and Schema Acquisition in Learning Elementary Computer Programming: Implications for the Design of Practice. *Journal of Computers in Human Behavior* **6**(3): 273-289.