nExaminer: A Semi-automated Assignment Assessment Framework for Moodle.

Zheng Cheng

Research Thesis 2011

M.Sc. in Computer Science (Software Engineering)

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Department of Computer Science

National University of Ireland, Maynooth

County Kildare, Ireland

**A thesis submitted in partial fulfillment
of the requirements for the
M.Sc. in Computer Science (Software Engineering)**

Supervisors: Dr. Rosemary Monahan and Dr. Aidan Mooney

January 2011

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master of Science in Software Engineering, is entirely my own work and has not been taken from the work of the others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: ______________________________ Date: _______________________________

Acknowledgement

I would like to thank Dr. Rosemary Monahan and Dr. Aidan Mooney at National University of Ireland, Maynooth for their support and input on the project.

Abstract

In this paper, we present the nExaminer framework for semi-automated assignment assessment in an open source virtual learning system, namely, Moodle. The motivation for developing the framework is established on the observation of a major problem associated with traditional assignment assessment in Moodle - managing an effective relationship between the instructor and the students is difficult. Thus, the design and implementation of the nExaminer framework is discussed, along with the advantages it provides to both students and instructors. Experimental results with the nExaminer framework have been encouraging. It shows that the framework provides instructors with the ability to semi-automatically generate subjective feedback and automate the process of objective assessment within Moodle in an efficient manner. Moreover, the results also manifest that students are motivated by the usage of automated objective assessment in the nExaminer framework.

Table of contents

1. Introduction

Programming is an essential but difficult skill that students are expected to learn [1]. As constructivism theory suggests, learning is an interactive process between experience and understanding [3]: when a student submits an assignment, the instructor should act as a counselor or a collaborator, providing positive advice and motivation that help student to understand the course material. Moodle [24], a virtual learning environment (VLE)[1], contributes to the constructivism theory by providing a platform that enables instructors and students to exchange knowledge. Thus, assignment details, results and feedback can be managed and viewed effectively by both instructors and students.

However, generating assignment results and feedback is still a manual process for the instructor. This work can be error-prone, time-consuming and difficult, e.g.:

- I/O based assignment requires the instructor to manually type input every time that it is executed, and input might or might not reflect the potential problem of the assignment.
- If students fail to submit a compiled program, tracing and debugging are unavoidable and tend to be labor intensive.
- Effectiveness aspects (e.g. algorithms) are usually missing from the feedback as identifying and documenting are not an easy task.

As a result, it is difficult for human graders to objectively and effectively evaluate assignments within the time limit, which renders large-scale courses impractical.

To manage assignment activities with a reasonable amount of resources in Moodle, we propose the nExaminer framework, a framework that integrates with Moodle, to automatically test objective aspects[2] of assignment and facilitate reviewing of subjective aspects[3]. Thus, students can get immediate objective feedback that helps when learning how to program and instructors can save resources in reviewing assignments which in turns allow resources to be distributed to other tasks.

The rest of the paper is organized as follows. We give the background of the problem in Section 2. Section 3 designs our solutions to the problem. Our implementation is detailed in Section 4. Section 5 evaluates our work using an experiment. Finally, Section 6 provides a critical analysis, presents our conclusions and makes suggestions for future work.

[1] Virtual learning environments are online software systems that mange and administrate teaching recourses
[2] Also known as functional aspects, which are expected functionalities in the assignment description.
[3] Also known as non-functional aspects, such as structure, effectiveness and style of the solution

2. Background

This section will describe all the background details for understanding why we propose semi-automated assignment assessment. It will describe the traditional approach of assessing assignments in Moodle. It will analyze the problems of traditional assignment assessment in Moodle. It will illustrate the solutions presented from previous work, using automated assignment assessment. It will also describe why semi-automated assignment assessment is our proposal. Finally, a brief description of the tools used in the project will be given.

2.1. Traditional assignment assessment in the Moodle

Moodle provides a portal that allows assignments to be uploaded and downloaded. When all students submit their assignments, the instructor can retrieve solutions from Moodle and begin evaluation. Although consensus on the best way to evaluate students has not been established, previous research [5] takes the position that two types of criteria are applied in the assessing procedure:

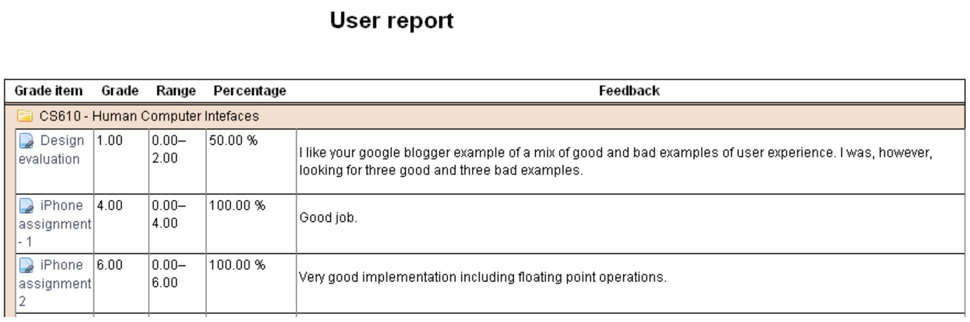
- Objective aspects (required functionalities).
- Subjective aspects (e.g. structure, style, efficiency).

Traditionally, objective aspects are evaluated by applying black box testing [40] (BBT) on the submitted solution – the assignment under test (AUT) is manually tested with pre-defined test data that ensure required functionalities are met. BBT produces a boolean answer that suggest whether the solution works or not and guides the instructor to grade the objective aspects.

The evaluation of subjective aspects of a student's work is usually based on the instructor's experience [5] - by examining the implementation of submitted assignment in detail, the instructor will provide their opinion of the student's work using his/her experience and knowledge.

It is worthwhile pointing out that subjective and objective assessment might not be entirely independent [8], for example, the code that remains unexecuted under the objective assessment intuitively suggest a strong motivation of evaluating the solution's structure.

Finally, the instructor combines objective and subjective results into a final report and updates them on Moodle (as shown in Figure 2.1).


User report

Grade item	Grade	Range	Percentage	Feedback
CS610 - Human Computer Intefaces				
Design evaluation	1.00	0.00–2.00	50.00 %	I like your google blogger example of a mix of good and bad examples of user experience. I was, however, looking for three good and three bad examples.
iPhone assignment - 1	4.00	0.00–4.00	100.00 %	Good job.
iPhone assignment 2	6.00	0.00–6.00	100.00 %	Very good implementation including floating point operations.

Figure 2.1: Traditional assignment assessment report in the Moodle

2.2. Problem analysis

Traditional assignment assessment approaches described in Section 2.1 are intrinsically hard to perform because of series of problems. We have cataloged these problems through interviewing and surveying literature, and will illustrate them in this section.

2.2.1. Objective assessment

The major problem of objective assessment is that students' solutions prevent the instructors from applying black box testing automatically on assignments, which results in instructors having to evaluate assignments manually [39]. In our opinion, it is not that students do it on purpose, but because the instructors do not explicitly state testability in the assignment description. As a result, students:

- Decorate outputs or generate irregular outputs.
- Do not provide consistent signatures[4] for compulsory methods of the AUT.

Thus, assignments cannot be automatically tested by the same set of tests.

2.2.2. Subjective assessment

In our experience, the assignment assessment becomes complicated when subjection is involved. Moreover, some tasks of subjective assessment might have particular problems. We discuss these problems in the sections that follow.

2.2.2.1. Structure assessment

Structure assessing is a sub-area of subjective assessment, which usually takes place after objective assessment [8]. After interviewed the instructors at National University of Ireland, Maynooth (NUIM), we think it should evaluate at least:

- The general structure of the submitted solution, e.g. are design patterns or anti-patterns applied? are method calls arranged in a reasonable sequence?
- The internal structure of the method, e.g. Is there any code that remains unexecuted after the objective assessment and is this code necessary?

However, our experience is that when reviewing the structure of an assignment, the hierarchy and the relationships between classes are not often clear at first glance, especially when an

[4] The Java language specification specifies that method signature includes the method name and the types and order of its parameters

assignment involves multiple classes. If we reach the common understanding that the code is raw data, previous research [17] finds that raw data without organization can make it very difficult for the recipient to understand their meaning.

In addition, there could be unexecuted code or unexecuted branches where new behaviors of the AUT can be hidden under objective testing. Thus, the instructor must generate inputs that increases the code coverage (through statement coverage [40], or branch coverage [40], or both), so that new behaviors of unexecuted code can be observed and validated [40]. But to our experience, increasing code coverage manually can be difficult. Because:

- There is no intuitive way to tell whether code coverage can be increased, e.g. unreachable code.
- Increasing code coverage manually requires tracking the internal work flow. This manual check can be error-prone and time-consuming.

All in all, structure assessing is difficult to perform efficiently.

2.2.2.2. Style assessment

Generally speaking, style refers to the programming conventions which help people to read and understand the code [38], e.g. the code should indent consistently.

In traditional assignment assessment, instructors often complain about style inconsistency among assignments, as the inconsistency causes great difficulty in reviewing solutions and style aspects grading, for example:

- Mutual acceptance of good style has not been established [35], thus the fair grading of assignments is difficult.
- Some bad styles prevent reviewing in an effective manner [35], e.g. curly brackets appear irregularly.

Sometimes, the instructor specifies what styles are expected, but the problem remains. Thus, we conclude that the instructor must express the expected style more concretely to allow students to remember, apply and verify it.

2.2.3. Revising the solution

We define revising as the activity of restructuring [19] or modifying code, to construct better solutions in terms of different metrics (e.g. code coverage [9] and code functionality [5, 6, 7]). In our opinion, revising will contribute to the evaluating and analyzing level in the cognition domain of Bloom's taxonomy (explained in appendix A). This will lead to students remembering the material that has been taught.

However, when we interviewed 29 students of a programming course at NUIM, about 25% (7/29) students claimed that they have not revised their own work before submission, and over 80% (24/29) students claimed that they have not revised in any form after submission.

We analyze the reasons behind these facts are compound:

- Previous research [34] finds that anxiousness tends to narrow the thought process so that we will directly concentrate on problems that are directly relevant. Thus, students do not revise their solutions before submission maybe because they are too anxious about their grade when they are constructing their solutions.
- Students do not revise their solutions after submission partially because students are intimidated by the poor structure of their own work. Or more likely, feedback is not provided (or not given on time), so students are not motivated to revise [2].

2.3. Previous work

Automated assignment assessment is an active research area [5, 6, 7, 8, 9]. Over a dozen mature and experimental systems (or frameworks) have been proposed in recent decades to solve some of the problems described in Section 2.2. They are mostly based on testing approaches where the student's assignment is executed with different representative data which automatically evaluates the reliability of submitted solution.

Textual comparison is the simplest technique used in automated assessment: real outputs of the AUT are compared with the expected output, using strict textual-based comparison – if any letter in the real output is different from the corresponding position in the expected output, the comparison result treats them as two distinct entities [42]. This technique can be found in the systems TRY [6] and ASSYST [7].

TRY is a standalone system that allows students to verify the correctness of their programs automatically. When TRY is launched, it uses predefined test data to test against submitted assignments. The testing mechanism is based on strict textual comparison. Finally, the student is shown results that inform of the soundness of their program. The student can keep using TRY to verify their solution, but will be penalized for failures exceeding a certain number of times. This penalty ensures that students inspect their coding behavior thoroughly before the next evaluation.

ASSYST is another system that uses textual comparison to automatically evaluate assignments. It introduces a scheme that assesses the submission across a number of criteria that are not restricted to functionality. It also checks whether the submission makes efficient use of the CPU, whether a programming style is followed and so on (only the functionality aspects are assessed automatically though). This system contributes to the recognition that automated assessment systems have the potential to become grading systems as they allow weightings to be distributed among different program metrics (e.g. efficiency, style).

Furthermore, testing tools and testing frameworks are introduced in automated assignment assessment. They provide facilities like assertions [41] to determine whether the AUT behaves as expected, e.g. assertion allows the instructor to check whether an object is in a particular state. Thus, the definition of a correct AUT can be broader than the similarity to the expected output.

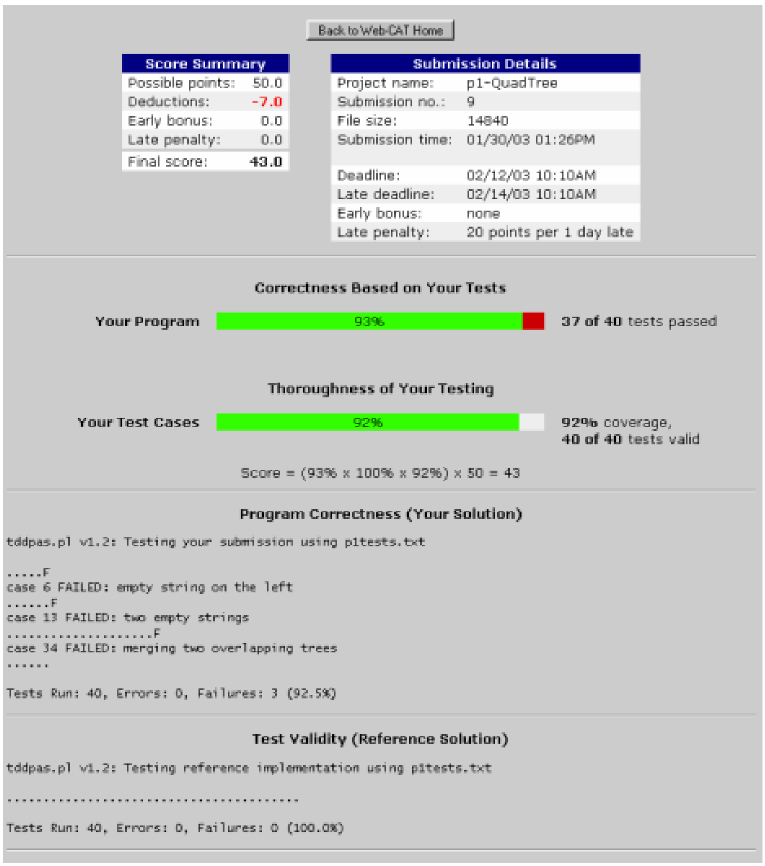
Autograder [5] is a framework developed for a course on data structures which is taught using the Java programming language. This framework is assisted by the testing framework – JUnit [29]. Since data structures are often abstracted to common interfaces, the assignment can be presented to students as a task of implementing the provided abstract interface, which will make each student's code testable by the same set of tests. This framework also requires students to be cautioned about programming style while coding their solutions. Thus, the style checking tool is invited in the assignment. If the student makes a style violation while programming, the style checking tool will complain simultaneously.

Web technologies allow testing tools and testing frameworks to be deployed on the internet. This allows both the instructor and student to invoke automated assignment assessment. Web-CAT [9] and FrontDesk [8] are developed in this way.

Web-CAT is a system that grades student's solution and test data as a whole. It emphasizes that students should be given the responsibility of providing test data to fully test their own code. Thus, three criteria are automatically measured on a student's solution:

- Effectiveness of submitted tests - applying submitted tests on the solution provided by the instructor.
- Thoroughness of submitted tests - calculating code coverage for submitted tests on the solution provided by the instructor.
- Correctness of the submitted solution – applying submitted tests on submitted solution.

A sample report of Web-CAT is shown in Figure 2.2.



Score Summary	
Possible points:	50.0
Deductions:	-7.0
Early bonus:	0.0
Late penalty:	0.0
Final score:	43.0

Submission Details	
Project name:	p1-QuadTree
Submission no.:	9
File size:	14840
Submission time:	01/30/03 01:26PM
Deadline:	02/12/03 10:10AM
Late deadline:	02/14/03 10:10AM
Early bonus:	none
Late penalty:	20 points per 1 day late

Correctness Based on Your Tests

Your Program 93% **37 of 40** tests passed

Thoroughness of Your Testing

Your Test Cases 92% **92%** coverage, **40 of 40** tests valid

Score = (93% x 100% x 92%) x 50 = 43

Program Correctness (Your Solution)

```
tddpas.pl v1.2: Testing your submission using p1tests.txt

.....F
case 6 FAILED: empty string on the left
......F
case 13 FAILED: two empty strings
....................F
case 34 FAILED: merging two overlapping trees
......

Tests Run: 40, Errors: 0, Failures: 3 (92.5%)
```

Test Validity (Reference Solution)

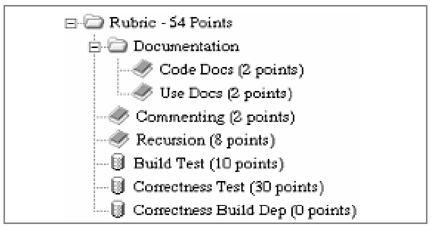
```
tddpas.pl v1.2: Testing reference implementation using p1tests.txt

........................................

Tests Run: 40, Errors: 0, Failures: 0 (100.0%)
```

Figure 2.2: Assignment report of Web-CAT

FrontDesk is an enterprise class web-based assignment assessment system that is designed for managing Computer Science courses. Functionalities are automatically verified through the JUnit. Each runtime exception raised by the JUnit assertion clause is a requirement violation. It transforms into XML format to be consumed by other components of FrontDesk for grading the AUT. FrontDesk also provides a portal for the instructor to generate feedback in a consistent format (as shown in Figure 2.3).



Rubric - 54 Points
Documentation
Code Docs (2 points)
Use Docs (2 points)
Commenting (2 points)
Recursion (8 points)
Build Test (10 points)
Correctness Test (30 points)
Correctness Build Dep (0 points)

Figure 2.3: Portal of FrontDesk for generating consistent feedback

2.4. Our proposal

The previous work described in Section 2.3 is based on automated assignment evaluation, which uses assistance from the computer to aid grading and generating feedback. This technique has three major educational advantages:

- Automated assessment is essentially interactive - students can receive immediate feedbacks after submitting their solutions. Thus, we believe this interactive nature will contribute to the evaluation level in cognition domain of Bloom's taxonomy.
- Automatic assessment allows more assignments to be developed by students [36]. In our opinion, it is important that the students write and test enough programs in order to understand and remember certain concepts. However, if students cannot evaluate their works before starting a new task, we feel that they will very likely stagnate at create level of the cognition domain. Automatic assessment solves this dilemma by shortening the time in waiting for feedback, and thus multiple assignments can perform effectively.
- Automatic assessment can save resources and allow them to be distributed to the teaching activities that cannot be or are difficult to automate [36], e.g. personal guidance and assignment design.

However, we also feel there are opinions and knowledge that come from experience, and this experience cannot easily be automated into software or systems. Therefore, instead of constructing a system in which the entire assessment is performed automatically by the computer, we propose the design of a semi-automatic assignment assessment framework where the instructor participates in the assessment but the computer simplifies the procedure. In this way the characteristics of automatic assessment can be preserved while keeping the important role of the instructor as counselor or collaborator in the VLE.

We have seen previous work using computer to simplify specific assessment procedure (e.g. objective assessment [5,6,7,8,9], style assessment [5]). Thus, we have confidence in improving them and discovering more in our semi-automatic assignment assessment framework.

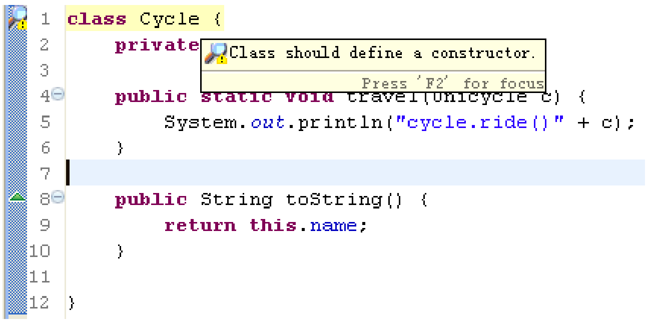
To make our proposal more concrete, we present the tools that will be used in this project in Section 2.5.

2.5. Tools used

In this section, we will give a brief description of the tools that are used in the implementation of our semi-automatic assignment assessment framework.

2.5.1. Checkstyle

Checkstyle [26] is a style checking tool that integrates with the development environment (e.g. Eclipse) to check code against sets of programming style rules. It is a static checker, which means that running Checkstyle does not require execution of the code - if the user makes a style violation while programming, the style checking tool will complain simultaneously (as shown in Figure 2.4).



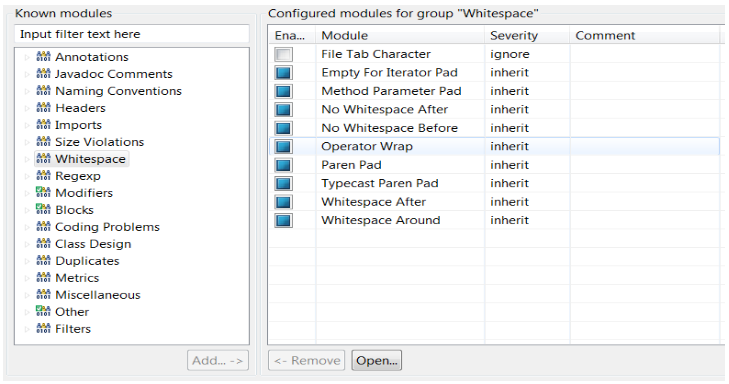
```
class Cycle {
    private
                               Class should define a constructor.
                                                Press 'F2' for focus
    public static void travel(Unicycle c) {
        System.out.println("cycle.ride()" + c);
    }

    public String toString() {
        return this.name;
    }

}
```

Figure 2.4: Example of Checkstyle

Checkstyle focuses on conventions checking, but also has the ability to locate bad practice and potential bugs. Rules template in Checkstyle is highly configurable in terms of the personal preference, allowing the tutor to choose a selection of rules to be checked (as shown in Figure 2.5) and providing a default template if none are specified.



Known modules

Input filter text here

- Annotations
- Javadoc Comments
- Naming Conventions
- Headers
- Imports
- Size Violations
- Whitespace
- Regexp
- Modifiers
- Blocks
- Coding Problems
- Class Design
- Duplicates
- Metrics
- Miscellaneous
- Other
- Filters

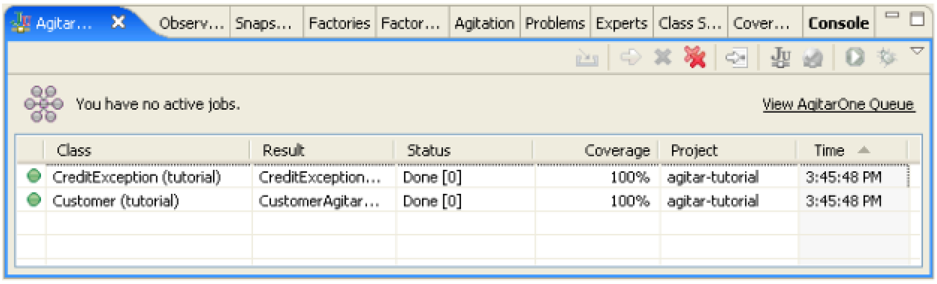
Configured modules for group "Whitespace"

Ena...	Module	Severity	Comment
☐	File Tab Character	ignore	
☑	Empty For Iterator Pad	inherit	
☑	Method Parameter Pad	inherit	
☑	No Whitespace After	inherit	
☑	No Whitespace Before	inherit	
☑	Operator Wrap	inherit	
☑	Paren Pad	inherit	
☑	Typecast Paren Pad	inherit	
☑	Whitespace After	inherit	
☑	Whitespace Around	inherit	

Figure 2.5: Rules template configuration of Checkstyle

2.5.2. AgitarOne

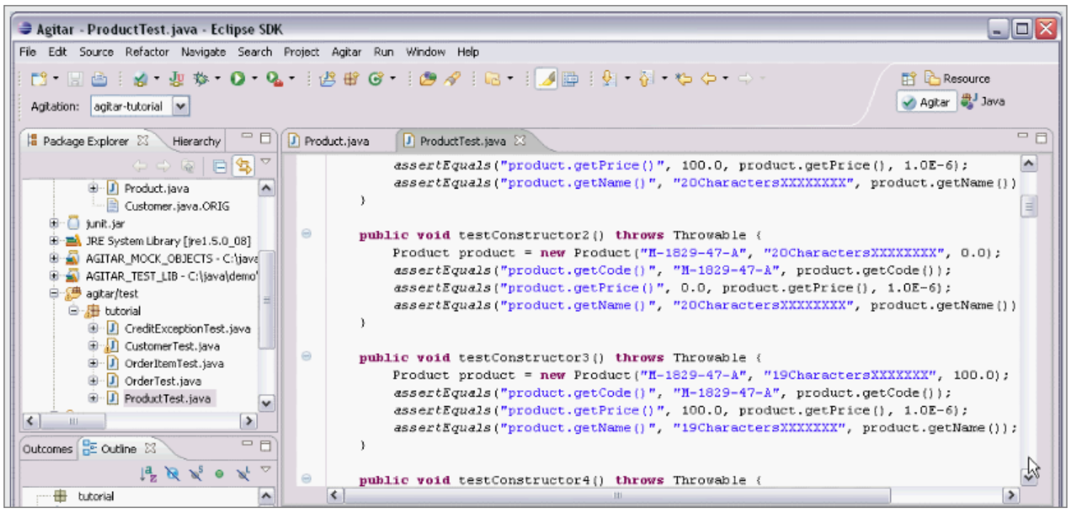
AgitarOne is a commercial tool that automates test case generation for reaching high code coverage [14]. It is a client/server solution - once AgitarOne is deployed and configured on the client side, the test generation request can be sent to the AgitarOne master server. Then, the test case generation proceeds on the server and sends back results to the client. The results not only calculate code coverage for generated test cases (as shown in Figure 2.6), but also include generated test cases to the project's directory of client (as shown in Figure 2.7)



You have no active jobs.

	Class	Result	Status	Coverage	Project	Time
🟢	CreditException (tutorial)	CreditException…	Done [0]	100%	agitar-tutorial	3:45:48 PM
🟢	Customer (tutorial)	CustomerAgitar…	Done [0]	100%	agitar-tutorial	3:45:48 PM

Figure 2.6: Code coverage results for generated test cases using AgitarOne



```
        assertEquals("product.getPrice()", 100.0, product.getPrice(), 1.0E-6);
        assertEquals("product.getName()", "20CharactersXXXXXXXX", product.getName())
    }

    public void testConstructor2() throws Throwable {
        Product product = new Product("M-1829-47-A", "20CharactersXXXXXXXX", 0.0);
        assertEquals("product.getCode()", "M-1829-47-A", product.getCode());
        assertEquals("product.getPrice()", 0.0, product.getPrice(), 1.0E-6);
        assertEquals("product.getName()", "20CharactersXXXXXXXX", product.getName())
    }

    public void testConstructor3() throws Throwable {
        Product product = new Product("M-1829-47-A", "19CharactersXXXXXXX", 100.0);
        assertEquals("product.getCode()", "M-1829-47-A", product.getCode());
        assertEquals("product.getPrice()", 100.0, product.getPrice(), 1.0E-6);
        assertEquals("product.getName()", "19CharactersXXXXXXX", product.getName());
    }

    public void testConstructor4() throws Throwable {
```

Figure 2.7: Generated test cases using AgitarOne

2.5.3. Ant, JUnit and Emma

Ant [28] is a Java library and command-line tool that can automatically drive tasks described in the Ant script (also called build file).

JUnit is a testing framework for developing unit tests[5] for Java programming language. It provides a base class called "TestCase" that can be extended to create a series of unit tests for the unit under test, an assertion library used for evaluating the results of individual tests, and several applications that run the created unit tests.

Emma [30] is an open-source tool for measuring and reporting code coverage of Java programs. It supports four types of code coverage calculation (class, method, line, basic block) and three types of output format (plain text, HTML, XML). It also has the capability of linking to the source code.

Both JUnit and Emma can be written as a task in the Ant script so that they can be driven by the Ant tool (the details are presented in Section 4.3.2).

[5] A unit test is a self-contained program that checks an aspect of the unit under test. A unit is the smallest testable part of the program.

2.5.4. Japa

Japa [33] is a Java language parser with abstract syntax tree (AST) generation and visitor support. The AST records the source code structure. It is also possible to change the AST nodes or create new ones to modify the source code using the provided visitor (the details are presented in Section 3.2 and Section 4.1.4).

In Section 2, we presented all the background details for understanding why we propose the semi-automated assignment assessment. We illustrated the traditional assignment assessment approach in Moodle, and then analyzed the problems within it. We also surveyed literature in order to have better insight of what have been done in the area of automated assignment assessment. Then, we proposed semi-automated assignment assessment, deduced from the literature. At the end of Section 2, we gave a brief introduction to the tools used in the project. In Section 3, we will illustrate the design of a framework called nExaminer to support our proposal.

3. Framework design

In this section, we will design the nExaminer framework so that it solves the problems of traditional assignment assessment in Moodle. This will be achieved as follows:

- In Section 3.2, a sample code transformer will be designed to allow the instructor explicitly convey testability to the student.
- In Section 3.3, a style pre-checking unit will be designed to ensure style consistency among submitted solutions.
- In Section 3.4, automated objective testing will be illustrated for its capability to help students revise solutions and an extended upload feature will be designed to prevent students from abusing this automated nature.
- In Section 3.5, automating increased code coverage also will be presented to help instructor identifying hidden new behaviors of the AUT quickly.

3.1. nExaminer framework overview

To get a quick overview of what nExaminer framework can do, we present the high level abstraction of the nExaminer framework and the use case diagram in this section.

3.1.1. Framework overview

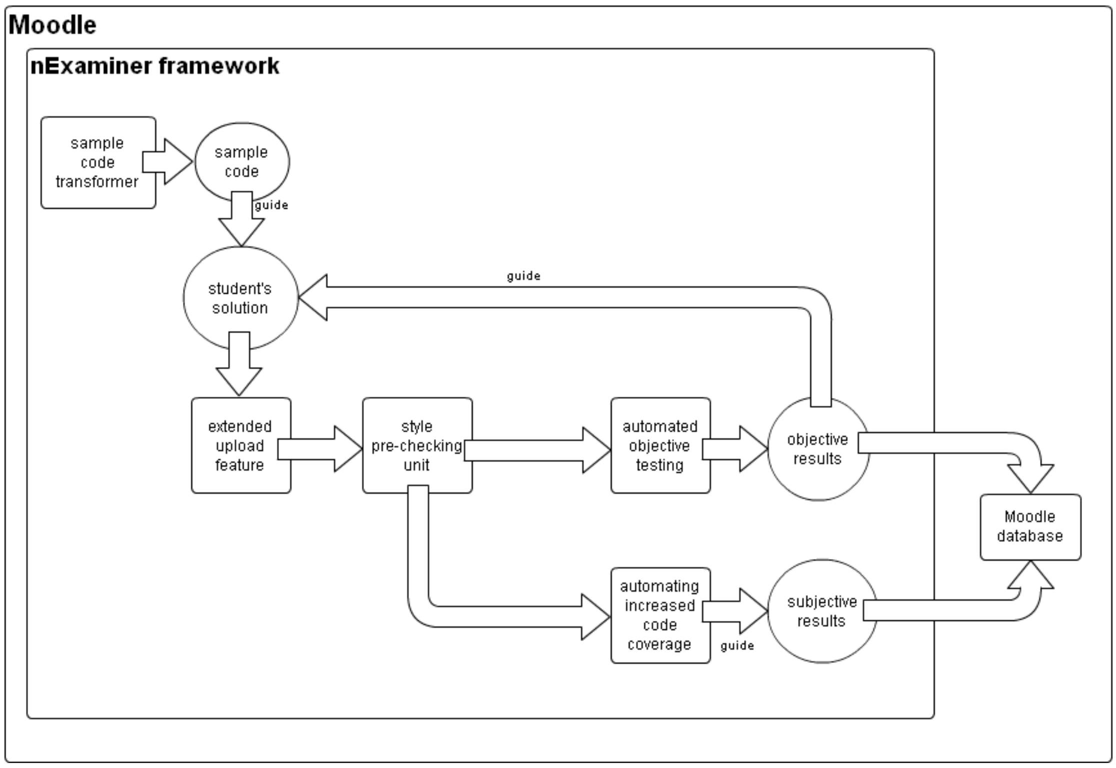


Figure 3.1: nExaminer framework

As we can see from Figure 3.1, the instructor will use the sample code transformer to generate sample code which will guide students to build testable solutions. When the students are finished coding, they can submit their solutions through the extended upload feature which checks that certain rules hold when uploading. Then the style pre-checking unit will be used to examine if the required programming styles are followed by the students. After that, the students' solutions will be uploaded onto Moodle triggering automated objective testing. The objective testing results will then be stored in the Moodle database and displayed to the students immediately.

Students can draw on the objective results to revise their work, and keep their attempts in the nExaminer framework. We refer to this as the learning mode. In this mode, students learn from previous experience and challenge themselves to construct improved solutions. This is repeated until either the students satisfy their work or are restricted by the framework. The final submitted solution will be examined by the instructor for subjective review. We refer this as the evaluation mode. In the evaluation model, the student's solution is subjectively reviewed by the instructor, facilitated by automating increased code coverage to find and validate new behaviors of the AUT. Finally, the instructor will generate feedback and upload it to the Moodle database.

3.1.2. Use case diagram

A use case diagram is shown in Figure 3.2 to manifest what instructors and students can do in the nExaminer framework.

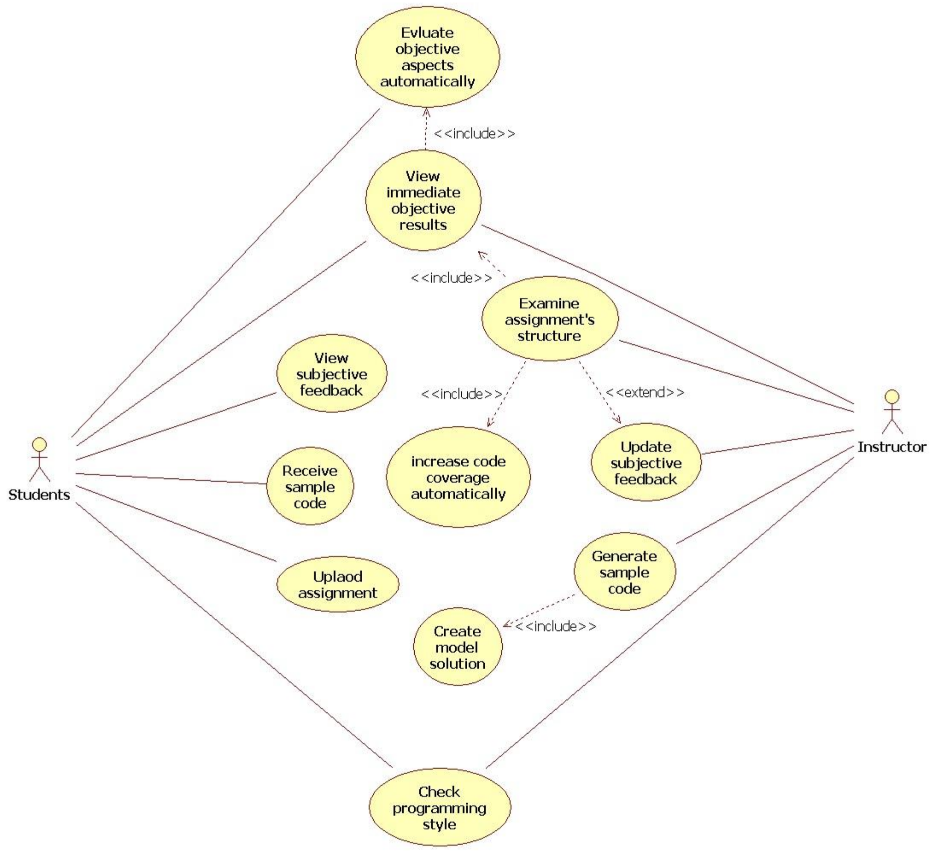


Figure 3.2: Use case diagram of nExaminer framework

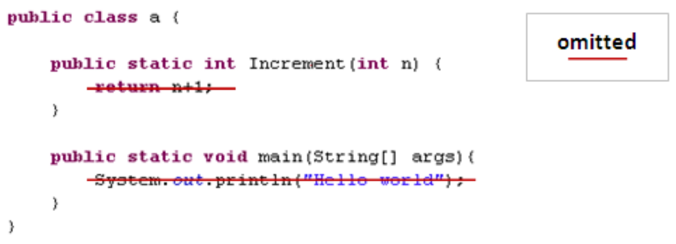
We now present the design of the nExaminer framework as described in the introduction of Section 3.

3.2. Sample code transformer

As discussed in Section 2.2.1, objective assessment is difficult to perform automatically since the instructor does not convey testability clearly to students in the first place. To allow an instructor to illustrate testability to students, the common interface is applied to the assignment design in data structure courses [5]. This is a legacy of object oriented programming and usually introduced to students ahead of advance computer science courses (e.g. data structures course). Thus, we doubt this abstract concept can be easily understood by students from backgrounds other than computer science.

Therefore, we propose a more general solution to state testability to students – the sample code. It is common to the interface in the way of asking students to implement compulsory classes and methods, while still using the students' creativity to discover the relationships between classes, design helper methods and so on. However, the sample code does not require students to have any preliminary knowledge - we intend to deliver an abstract concept without alerting the students, so that by the time students realize they are using an "interface" in development, they have experience about what "interface" can do for them and they are likely to keep using it. Consequently, using sample code to enhance testability can be applied to courses other than data structure courses and solves the problem of conveying testability as presented in Section 2.2.1.

To facilitate the provision of sample code for programming assignments, we design a transformer that omits certain programming constructs within a provided model solution (as shown in Figure 3.3). This allows us to construct sample code in a concise manner.



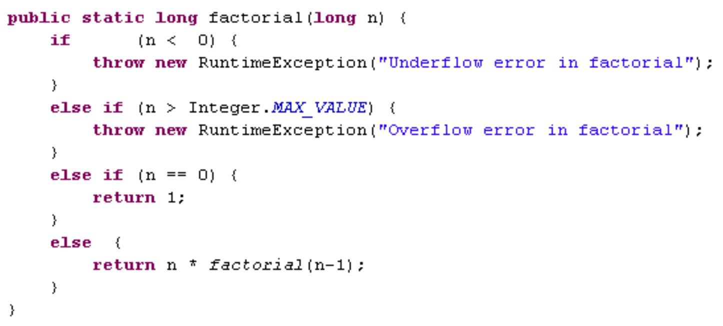
```
public class a {

    public static int Increment(int n) {
        ~~return n+1;~~
    }

    public static void main(String[] args){
        ~~System.out.println("Hello world");~~
    }
}
```

Figure 3.3: Omitting programming constructs in model solution

A model solution is a possible way to solve an assignment, provided by the instructor. It is consistent with assignment requirements, e.g. a model solution for factorial is shown in Figure 3.4. Of course we do not want to provide the full solution to the students. Instead we only want to provide a template. We can use the sample code to provide this template.

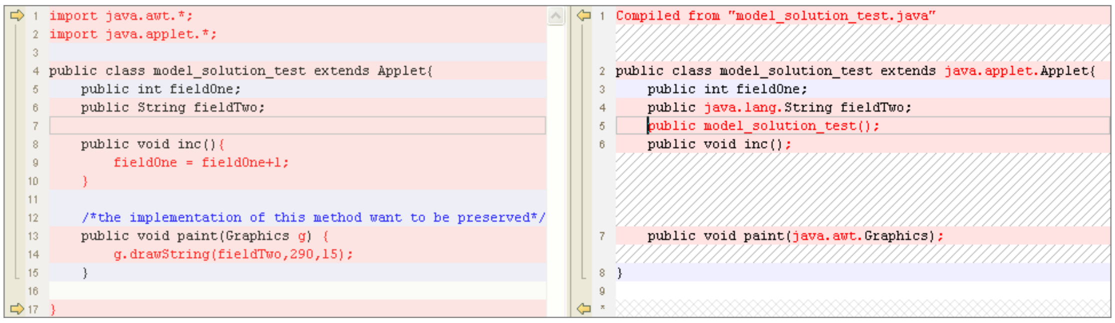


```
public static long factorial(long n) {
    if      (n <  0) {
        throw new RuntimeException("Underflow error in factorial");
    }
    else if (n > Integer.MAX_VALUE) {
        throw new RuntimeException("Overflow error in factorial");
    }
    else if (n == 0) {
        return 1;
    }
    else  {
        return n * factorial(n-1);
    }
}
```

Figure 3.4: The model solution for factorial (written in Java)

For example, the sample code transformer can be simply implemented with tools like the Java class disassembler [25] - by passing in the compiled Java file (.class), the disassembler will print out the fields and methods information within the class. Thus, students can have good sample code to reference (as shown in Figure 3.5):

- The class name is preserved.
- The fields of the class are preserved.
- The implementation within method body is omitted.
- The method signatures, return type and modifiers are preserved.



```
import java.awt.*;
import java.applet.*;

public class model_solution_test extends Applet{
    public int fieldOne;
    public String fieldTwo;

    public void inc(){
        fieldOne = fieldOne+1;
    }

    /*the implementation of this method want to be preserved*/
    public void paint(Graphics g) {
        g.drawString(fieldTwo,290,15);
    }

}
```

```
Compiled from "model_solution_test.java"
public class model_solution_test extends java.applet.Applet{
    public int fieldOne;
    public java.lang.String fieldTwo;
    public model_solution_test();
    public void inc();
    public void paint(java.awt.Graphics);
}
```

Figure 3.5: The left side is the source file and corresponding Java class disassembler output is on the right

However, we can also see from Figure 3.5, the output of Java class disassembler might be difficult for students to understand:

- Object's type is appended with full package name.
- Import clauses are eliminated.
- Cannot show any implementation even if it is intended.
- Comments in the source code are eliminated.

In our solution, we design a sample code transformer that works as follows:

1. Initially, the instructor implements the model solution and annotates the methods with the desired hiding level (as shown in Figure 3.6) and sends to the sample code transformer.

```
public class Model_solution {
    @hidden("method_body")
    public int test(int n) {
        n = 1;
        return n;
    }
}
```

Figure 3.6: Annotating method within model solution

2. Next, each source code will be syntactically parsed to build the corresponding AST (the AST for the code of Figure 3.6 is shown in Figure 3.7), where each node of the tree denotes a programming construct occurring in the source code. The reason why we prefer not to directly manipulate the source code is because relationships among programming constructs are not clear in the source code [18]. Thus, it will be appropriate to build an AST first to ensure code transforming is proceeded on top of specific-structured programming constructs instead of entire-unstructured source code.

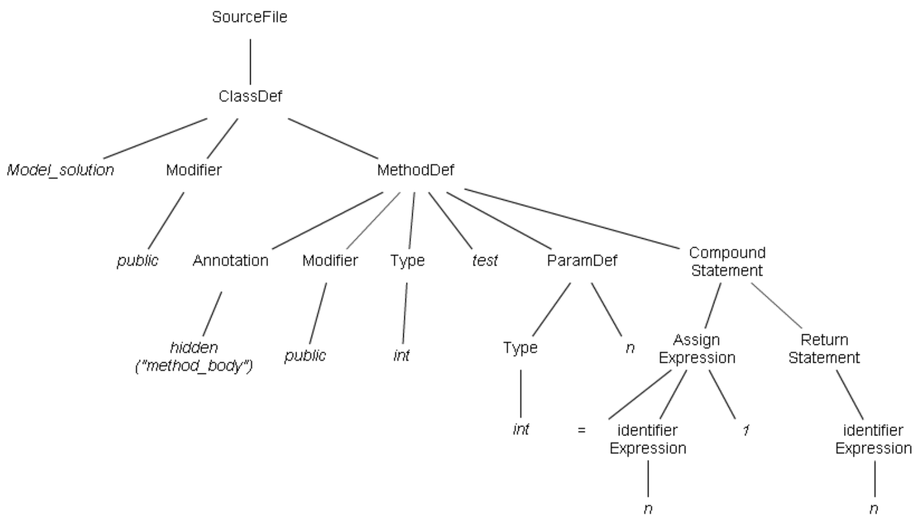


Figure 3.7: Build AST

3. The third step is to traverse the entire AST to find method nodes that are being annotated, and then process the method nodes with the corresponding behavior.

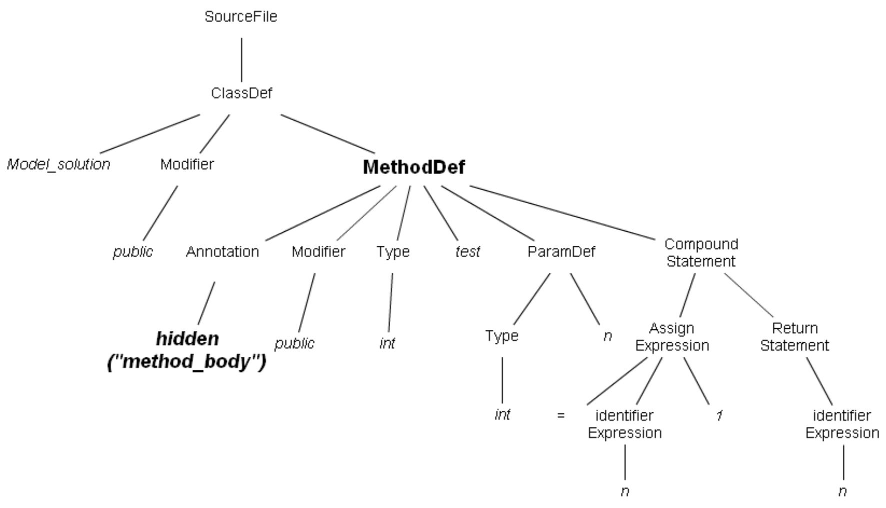


Figure 3.8: Finding method nodes that being annotated

We can see from Figure 3.8, the annotation of 'hidden("method_body")' is detected, so the compound statement node (which represents the method body) and the annotation node (which is not relevant to students) are eliminated in the sample code as shown in Figure 3.9.

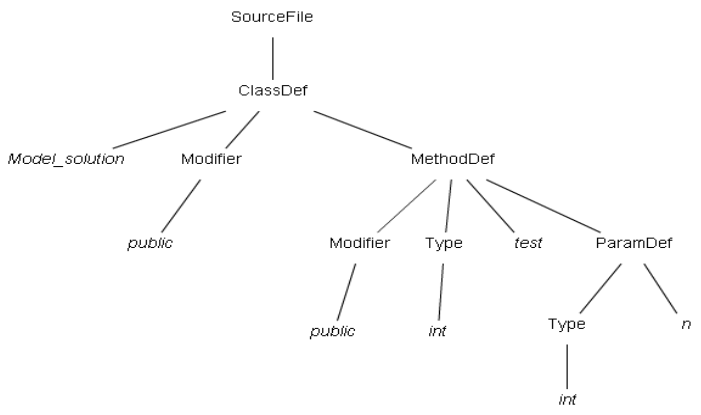
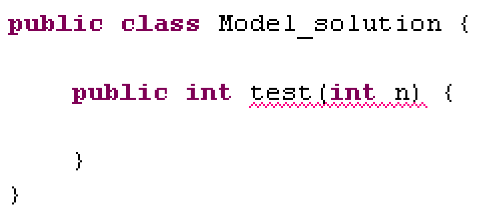


Figure 3.9: Process the AST tree

4. Finally, we output the modified code (s), as shown in Figure 3.10.



```
public class Model_solution {

    public int test(int n) {

    }
}
```

Figure 3.10: Output of sample code transformer

The work flow of the sample code transformer can be abstracted as in Figure 3.11.

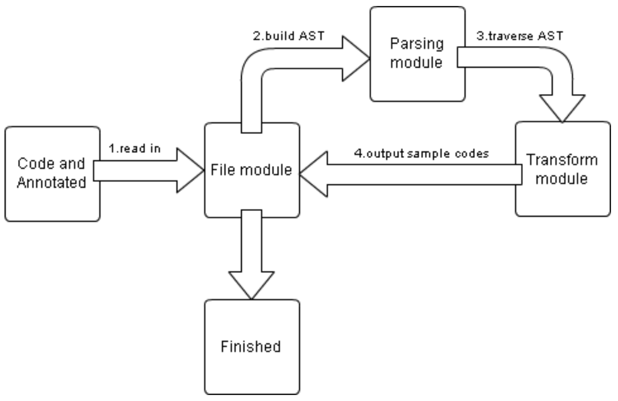
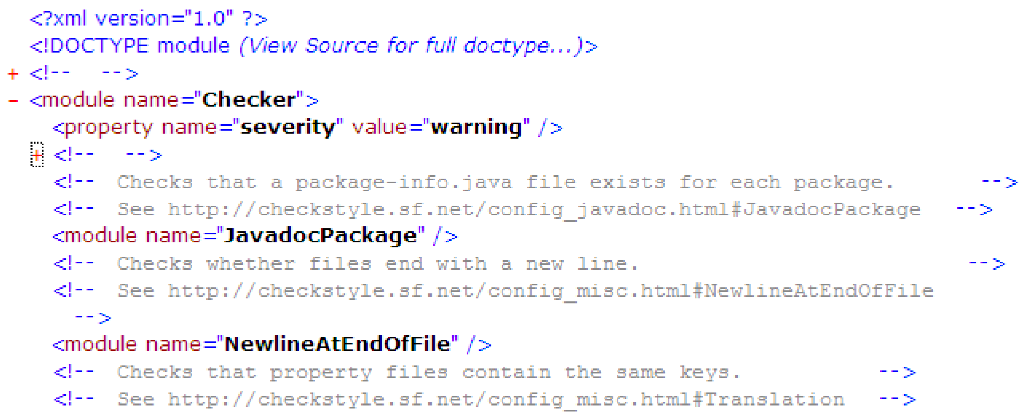


Figure 3.11: Workflow of sample code transformer

3.3. Style pre-checking

In Section 2.2.2.2, we discussed the problem that style inconsistency causes the instructor great difficulty in subjective assessment. To ensure that the expected style is concretely conveyed to students, previous work [5] suggests applying style checking tools such as Checkstyle [26]. These tools are able to plug into an integrated development environment (IDE) and will complain about style violation immediately when a user is coding. This is achieved using a rules template which is configurable and exportable (as shown in Figure 3.12).



```
   <?xml version="1.0" ?>
   <!DOCTYPE module (View Source for full doctype...)>
 + <!--  -->
 - <module name="Checker">
     <property name="severity" value="warning" />
   + <!--  -->
     <!-- Checks that a package-info.java file exists for each package.      -->
     <!-- See http://checkstyle.sf.net/config_javadoc.html#JavadocPackage   -->
     <module name="JavadocPackage" />
     <!-- Checks whether files end with a new line.                          -->
     <!-- See http://checkstyle.sf.net/config_misc.html#NewlineAtEndOfFile
       -->
     <module name="NewlineAtEndOfFile" />
     <!-- Checks that property files contain the same keys.         -->
     <!-- See http://checkstyle.sf.net/config_misc.html#Translation  -->
```

Figure 3.12: Rules template of Checkstyle

However, style checking tools can be turned off while coding, rising uncertainty about style consistency of submitted assignments. We suggest that it would be better to provide the instructor with confidence that students indeed conform to the specified style. Thus, besides applying style checking tools (nExaminer is integrated with Checkstyle) while programming the solutions, we propose pre-checking style consistency when students upload their solutions on Moodle to ensure that submissions with style violation are rejected. Moreover, rejected submissions should be given style violation details to help the students correct their solution before resubmission.

As we can see from the prototype of a style pre-checking unit shown in Figure 3.13, the files that have style violations will be displayed in tab fashion. The content of each tab are divided into left and right panel, so that each line in the source file will have its line number displayed in the left panel and corresponding source code in the right panel. Furthermore, the lines that have no style violation will be displayed normally and the style-violated line will be highlighted by its line number. When a student hovers onto the line number, the violated message will be shown.

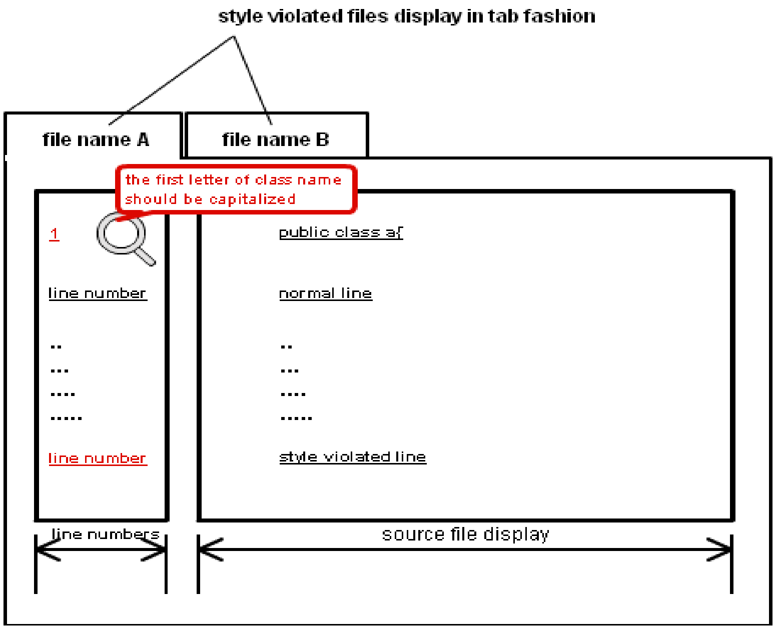


Figure 3.13: Prototype of style pre-checking unit

As a result, students are more likely to be concerned with the readability of their code, which in turn reduced the stress of reviewing assignment for the instructor. The problem of style inconsistency among submitted assignment (as discussed in Section 2.2.2.2) can be solved.

3.4. Automated objective testing

To allow students to have a quick idea about whether their code performs as they expect, automated objective testing has being continually experimented and applied in the area of automated assignment assessment [5,6,7,8,9]. Consequently, the interactive nature of automated objective testing motivates students to revise their work more frequently with the assistance of immediate functionality results [6, 20]. We build our automated objective testing tool using Ant, JUnit and Emma to assess Java assignments (general description is shown in Section 4.3.2), so that functionality results of submitted solution can be generated automatically as in other earlier work [5,8]. In this paper, we focus on integrating automated objective testing with Moodle and how to prevent abusing its interactive nature.

3.4.1. Extended upload feature

The student who wants to repeat the assignment evaluation process is motivated by improving grades on successive attempts, and interested in feedback [4]. However, immediate resubmission after a previous attempt may yield an undesired outcome: students tend to make small changes to their solutions without thinking thoroughly, and keep submitting to see the feedback changing [4]. Thus, we believe this resubmission procedure will provide little progress in learning.

Previous research proposes penalizing students for sending failed solutions more than a specified number of times [6]. We worry that students may be intimated by this restriction and stop revising once penalized. Therefore, we propose adding upload limitations to Moodle in order to let students inspect their coding behavior thoroughly before their next submission. Specifically, we design the extended upload feature on Moodle to check that the following rules hold when students submit:

- There should have a time interval between two continuously submissions.
- Total submission times should not exceed a certain times.
- The submitted date should be before the specified deadline of the assignment.

If students can successfully submit their solutions on Moodle, we assume these solutions are the results of thorough thinking.

3.4.2. Integration with Moodle

There are four assignment types in Moodle by default (shown in Figure 3.14). We want to add a new type that will draw on the style pre-checking unit, the extended upload feature and the automated objective testing tool, so that automated objective testing can be integrated with Moodle.

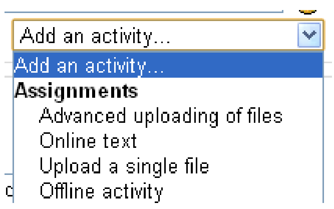


Figure 3.14: Default assignment type in Moodle

More specifically, this new assignment type is different to the other assignment types in its upload behavior: when student uploads a solution for this new assignment type, the solution will be sent to the extended upload feature to check against a set of upload rules, if any rules are violated, the upload procedure will terminate. Otherwise, the submission is permitted for upload onto Moodle, and then assessed by the style pre-checking unit. If style violations are found, the upload procedure will terminate, otherwise, the solution can be evaluated by the automated objective testing tool. Then, the corresponding objective results will be generated by the automated objective testing tool and stored in the Moodle database. By the end of the upload procedure, the student can see whether their submitted solution behaves as expected. The whole upload procedure is shown in Figure 3.15.

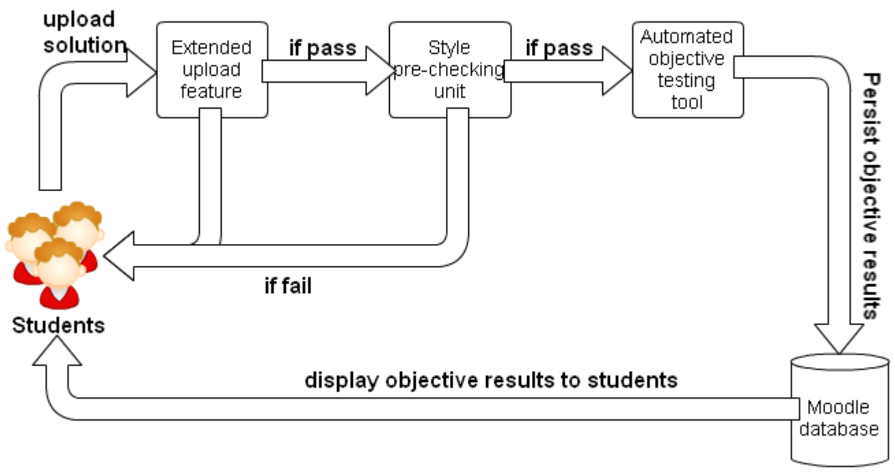


Figure 3.15: Work flow of automated objective testing

By following the above procedure, automated objective testing is integrated with Moodle through this new assignment type. Thus, students can receive immediate functionality results while still using Moodle as a platform to share knowledge.

As we discussed in Section 2.2.3, anxiety and slow feedback prevents students revising their works. The automated objective testing provides immediate feedback that allows students to have an extent of confidence and motivation to revise their solutions. We expect that automatic objective testing can be effectively and wisely used on Moodle to contribute to the learning activities.

3.5. Automating increased code coverage

As presented in Section 2.2.2.1, increasing code coverage manually causes the instructor great difficulty when performing subjective reviewing. Various techniques have been proposed to automate the process of increasing code coverage to observe new behaviors of the software under test (SUT) [10, 11, 12, 13]. Among these, dynamic symbolic execution (DSE) is especially suited in our work because of its capability to quickly generate effective sets of test cases that reach high code coverage of the SUT [12].

Initially, the DSE needs to select a type of code coverage (e.g. branch coverage, statement coverage) as the ultimate goal. When the selected code coverage reaches the maximum, the DSE should stop. Next, it executes the SUT for some random inputs and collects encountered predicates in branch along the execution. The conjunction of these predicates is constructed as a conjunction equation which is modified by flipping[6] one predicate at a time to generate a new conjunction equation. Solving the new equation will generate further test inputs that contribute to increase code coverage. If the new conjunction equation is infeasible to be solved, it will be discarded and restarting with new random inputs [13]. This process is repeated until the selected code coverage reaches the maximum.

[6] Flip means negating the predicate, e.g. predicate x!=90 is the negating result of x=90

In the educational environment, the SUT we have just discussed is the AUT which is being assessed. Consequently, the task of increasing code coverage is simplified by applying DSE, allowing the instructor to dedicate more time to observe and validate the new behaviors of AUT through generated test cases (the details are discussed in Section 4.4).

In Section 3, we designed components in the nExaminer framework in order to semi-automate assignment assessment in Moodle. We designed sample code transformer to help the instructor state testability to the students in Section 3.2. We presented the design of style pre-checking unit to ensure style consistency among the students' submissions in Section 3.3. We also illustrated how we will automate object assessment in Moodle and designed extended upload feature to prevent students from abusing this automated nature in Section 3.4. Finally, we demonstrated the concept of automating increased code coverage to help instructor identifying hidden new behaviors of AUT quickly in Section 3.5. Next, we will present the implementation of each design in Section 4.

4. Implementation

In this section we present the implementation of the nExaminer framework. We will take the design discussed in Section 3.2 and show how a concrete sample code transformer can be built with the assistance from Japa [33]. We will draw on Checkstyle [26] to construct style pre-checking unit in order to fulfill the design presented in Section 3.3. We will also show how the extended upload feature can be built using the design from Section 3.4.1. In addition, we will use the design of Section 3.4.2 and present the implementation of a new assignment type in the Moodle that can allow students and instructors to automate objective assessment by using Ant [28], JUnit [29] and Emma [30]. Finally, we will show how to automate the increased code coverage as described in Section 3.5 using AgitarOne [14].

In Figure 4.1, we can see a component diagram that shows the tools that we integrate and develop within the framework which we call nExaminer.

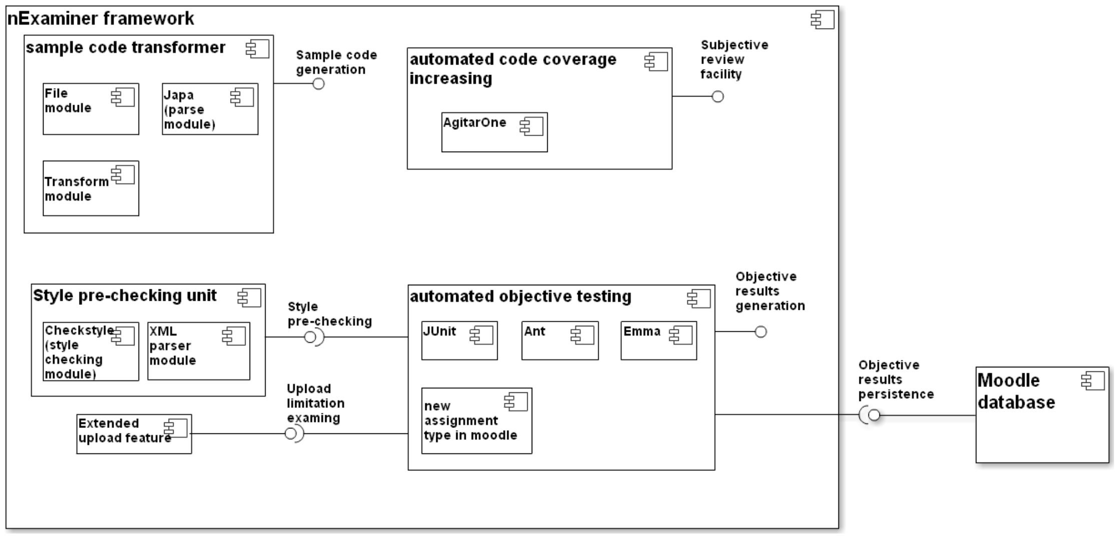

nExaminer framework
sample code transformer
File module
Japa (parse module)
Transform module
Sample code generation
automated code coverage increasing
AgitarOne
Subjective review facility
Style pre-checking unit
Checkstyle (style checking module)
XML parser module
Style pre-checking
Extended upload feature
Upload limitation examing
automated objective testing
JUnit
Ant
Emma
new assignment type in moodle
Objective results generation
Objective results persistence
Moodle database

Figure 4.1: Component diagram of the nExaminer framework

We discuss our implementation of this framework in the sections that follow.

4.1. Sample code transformer

In Section 3.2, we designed the sample code transformer to enhance the testability of assignments. We illustrate how it is developed in this section. The class diagram shown in Figure 4.2 presents the abstraction to the implementation of our sample code transformer.

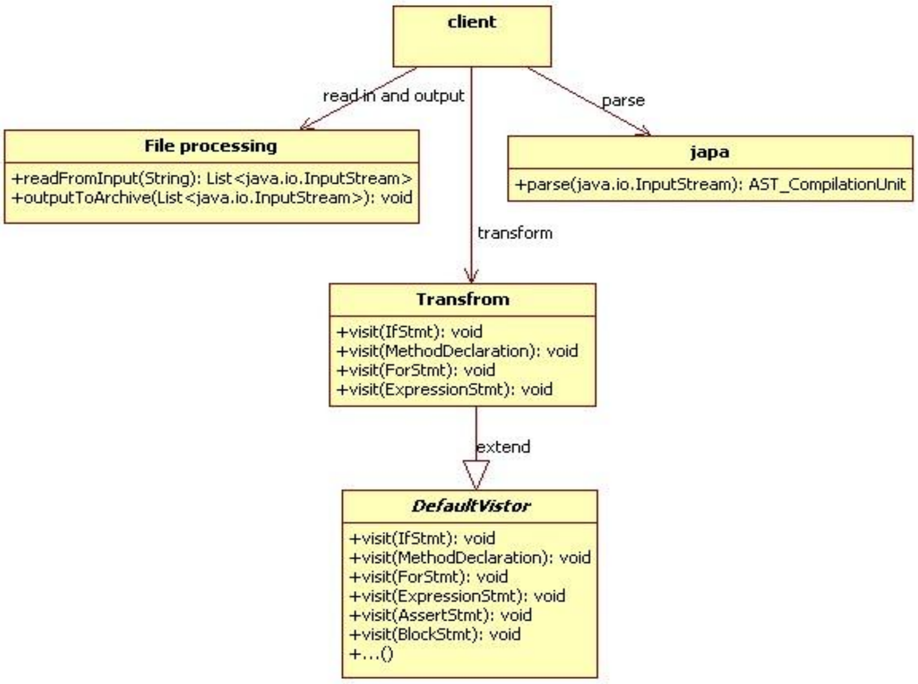

client
read in and output
parse
File processing
+readFromInput(String): List<java.io.InputStream>
+outputToArchive(List<java.io.InputStream>): void
japa
+parse(java.io.InputStream): AST_CompilationUnit
transform
Transfrom
+visit(IfStmt): void
+visit(MethodDeclaration): void
+visit(ForStmt): void
+visit(ExpressionStmt): void
extend
DefaultVistor
+visit(IfStmt): void
+visit(MethodDeclaration): void
+visit(ForStmt): void
+visit(ExpressionStmt): void
+visit(AssertStmt): void
+visit(BlockStmt): void
+...()

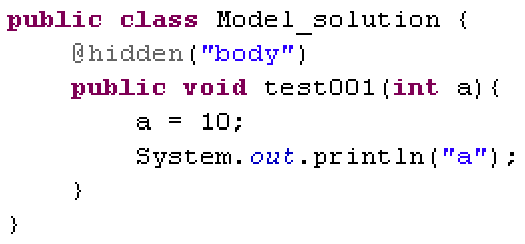
Figure 4.2: Class diagram of sample code transformer

4.1.1. Defining annotation

In order to provide instructors with the privilege of choosing their desired levels of hiding, we define three types of annotation that can be put on the model solution (as shown in Figure 4.3):

- @hidden (“body”) - for hiding the entire method body.
- @hidden (“if”) - for hiding only the “if blocks” within an annotated method.
- @hidden (“for”) - for hiding only the “for blocks” within an annotated method.

Since these annotations are only identifiers that need to be recognized by the sample code transformer while parsing and have no actual functionality, we are not required to add new classes for them.



```java
public class Model_solution {
    @hidden("body")
    public void test001(int a){
        a = 10;
        System.out.println("a");
    }
}
```

Figure 4.3: Annotated method within model solution

4.1.2. File processing module

In order to read in source code from the input model solution and output to the corresponding sample code, we implement a file utilities module that helps the input and output processes.

The reading process analyzes the format of the input: If the input is a single Java source file, then we return a list that contains the file stream of the input to the caller who invokes the reading process. If the input is an archive that contains multiple Java source files, then we un-archive the input first, and then return a list that contains the file stream of each un-archived Java source file to the caller.

The sample code will be generated through the transform module (described in Section 4.1.4), as a series of raw file streams. Then, the file processing module reads each file stream and storing it as a Java source file. Finally, once all sample codes are stored in the Java source files, they will be archived into a zip file for the convenience when uploading to Moodle.

4.1.3. Parsing module

We use Japa for the parsing task. By sending in a file stream of Java source file, Japa returns a corresponding AST compilation unit that can be manipulated with.

4.1.4. Transform module

The Japa tool provides a default visitor object that uses the visitor pattern to traverse the AST - the AST can have a corresponding visitor object that defines how to handle each visited nodes in the AST while traversing. We extend the default visitor object to develop the transform module.

The default visitor object handles each encountered node by printing it out. However, method nodes, annotation nodes, if statement nodes and for statement nodes require special treatment in the transform module. Thus, we implement a new visitor object that inherits its behaviors from the default visitor object to treat the normal nodes, and overwrite the behaviors for the nodes that require to be treated differently. Specifically, when a method node is encountered while traversing the AST, we get its annotation node first. If hidden annotation is detected, special treatment will performed accordingly:

- For the method annotated @hidden (“body”) - all compound statement sub-nodes under the method nodes are replaced with the sentence “your codes here”.
- For the method annotated @hidden (“if”) - only “if” statement sub-nodes and corresponding else statement sub nodes under the method nodes are replaced with the sentence “your codes here”.
- For the method annotated @hidden (“for”) - only “for” statement sub-nodes under the method nodes are replaced with the sentence “your codes here”.

When the transform module completes its task, a file stream of sample code is generated, which needs to be stored in a Java source file (as shown in Figure 4.4) by using the file processing module.

```
public class Model_solution {
    public void test001(int a){
        /* your codes here*/
    }
}
```

Figure 4.4: Result of sample code transformer

4.1.5. Additional implementation

There could be circumstances where the instructor accidentally annotated methods with reduplicated annotations (e.g. annotate hidden "method body" on the same method twice) or added multiple annotations that conflict with each other (e.g. hidden "if statements" while re-specifying to hide the entire method body). Therefore, we add an extra module to solve the confliction by applying the following rules:

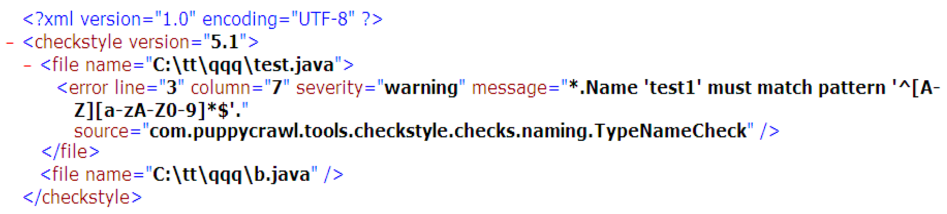
- Eliminate all reduplicated annotations.
- Choose higher hidden scope types (body > if, for) for a method.

In conclusion, we implement the sample code transformer as we designed in Section 3.2. Thus, when students receive the sample code, their solution will be constructed on top of a predefined structure which establishes the foundation of automated objective testing. The transform module is flexible for modification and extension - If instructors want to extend or modify the behaviors of the node processing, all they need to do is inheriting from the default visitor object of the Japa tool or the transform module that we designed, and define new behaviors for processing the specified nodes. In addition, we implement a new functionality to solve the annotation conflict which makes the sample code transformer more robust.

4.2. Style pre-checking unit

As discussed in Section 3.3, the nExaminer framework uses Checkstyle to check style violation while students are programming their solutions. The instructor should use Checkstyle first to generate a style template and hand it over to the students. After students successfully install Checkstyle and import the style template, they can be informed immediately when a specified style rule is violated. However, there is no guarantee that students indeed followed the instructions to apply Checkstyle. Thus, style pre-checking unit is integrated into Moodle to double-check that the specified style is followed by students.

The first task is style checking. When a student submits a solution, the server side Checkstyle plug-in will be invoked automatically. This plug-in is the same one we discussed about in Section 3.3, but activated in command-line mode. We use command-line mode because it is easy to be invoked from the script that we develop. The server side Checkstyle plug-in expects a style template (placed on server in advance) and target files (the submitted assignment) to perform style checking, the output can be in text, XML or HTML format. We prefer XML format for its structured nature. Sample XML results are shown in Figure 4.5.

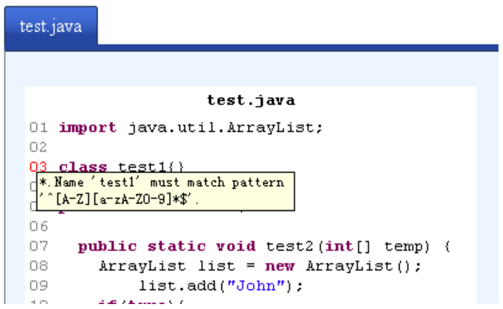


```
<?xml version="1.0" encoding="UTF-8" ?>
- <checkstyle version="5.1">
  - <file name="C:\tt\qqq\test.java">
      <error line="3" column="7" severity="warning" message="*.Name 'test1' must match pattern '^[A-
        Z][a-zA-Z0-9]*$'."
        source="com.puppycrawl.tools.checkstyle.checks.naming.TypeNameCheck" />
    </file>
    <file name="C:\tt\qqq\b.java" />
  </checkstyle>
```

Figure 4.5: XML output of invoking Checkstyle from command-line mode

Next, XML output of Checkstyle will be analyzed by the Java XML parser. As shown in Figure 4.5, the XML hierarchy is straightforward: each error tag manifests an error under the file tag. We are interested in line and message attributes of the error tag, so that style violation message can be positioned to the source file, guided by the line number.

Finally, after related style violation messages have been inserted at the appropriate position, source files that violate style are displayed in a tab-fashion webpage (as shown in Figure 4.6) and shown to students. If no style violation is found, the style pre-checking unit will return a signal (number of value one) as the style pre-checking result.



```
test.java
import java.util.ArrayList;

class test1{}
*.Name 'test1' must match pattern
'^[A-Z][a-zA-Z0-9]*$'.

  public static void test2(int[] temp) {
    ArrayList list = new ArrayList();
      list.add("John");
```

Figure 4.6: Sample of style pre-checking results.

To summarize, the implementation of the style pre-checking unit uses Checkstyle to verify style consistency of submitted assignments and displays style violation in an efficient manner. As a result, the students can take advantage of immediate and concise feedback to revise their work and will be more cautious about style consistency.

4.3. Automated objective testing

In this section, we will illustrate the implementation of the extended upload feature. Then, we will give a brief implementation of an automated objective testing tool and show how to integrate it with Moodle.

4.3.1. Extended upload feature

In Section 3.4.1, we illustrated the extended upload feature to help students inspect coding behaviors. We are going to describe how to implement this feature in this section.

The first extension is to add a rule that examines time interval between consecutive submissions. Database manipulation language (DML) is a family of pre-defined functions used in moodle to manipulate the Moodle database [27]. When students submit, we use DML to retrieve the date of the last successful submission from the Moodle database, compared with the current submission time. If the comparison result is less than the predefined time interval, our assumption is that students might not think thoroughly. Such activity should be discouraged by rejecting the submission.

The second extension is to add a rule that restricts the total number of permitted submissions. When a student submits, we use the DML to count all previous successful submissions. If the results are equal to maximum submission times, the upload attempt will be rejected.

The last extension verifies the rule of whether the assignment deadline has been met. We retrieve deadline information from the Moodle database using the DML, and compare that deadline information with the current submission time. The submission successes if the deadline is not reached.

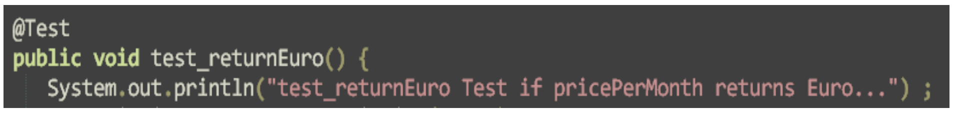
To conclude, the extended upload feature allows students to inspect their coding behaviors thoroughly before next submission through the introduced upload extensions. Its implementation is based on the DML. The DML has been professionally developed and is fully tested. Thus, we can achieve some important features for free, e.g. input filtering. Furthermore, the programming style of extended upload feature is consistent with other components within Moodle if the DML is used; thereby ensuring the implementation can be easily identified by other seasoned Moodle developers. However, we also acknowledge a limitation in the implementation of our extended upload feature - the properties such as time interval and maximum submission times are fixed in the implementation of the extended upload feature, which might be hard to configure and adjust in the long run. Thus we will describe how we implement the new assignment type in Moodle to manage these properties in Section 4.3.3.2.

4.3.2. Automated objective testing tool

Although building an automated objective testing tool is not our major focus, we give a general description about how we construct the tool to automatically test the objective aspects of the AUT.

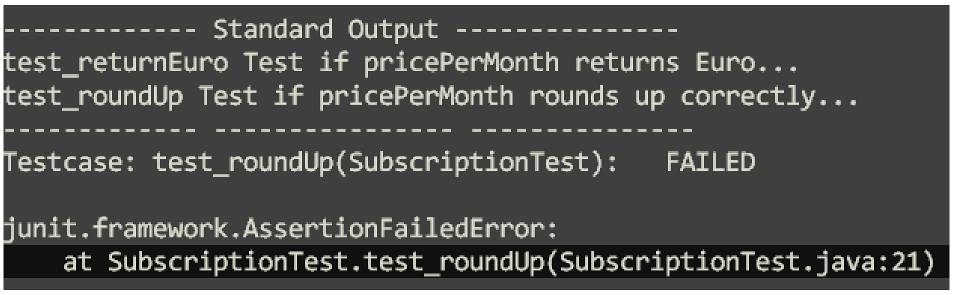
We assign the compilation task, the objective testing task (JUnit) and code coverage calculation task (Emma) in the Ant script (a default Ant script can be found in Appendix B). The compilation task is for compiling the AUT. The objective testing task uses predefined test cases to drive functional testing on the compiled AUT. The code coverage calculation task can then generate a report to show whether there are lines (and branches) that do not execute under predefined test cases, which guide the instructor to perform subjective reviewing. The location of a submitted solution must be sent as additional information to the Ant script, so that the compiler, JUnit and Emma will know the place where the AUT can be referenced and then perform the task.

We want to explicitly stress that the instructor should document the intention of each test case (an example is shown in Figure 4.7) so that while the JUnit performs functional testing, the intention of each test will be output to the JUnit report, along with debug information of each failed test case (as shown in Figure 4.8). As a result, we can match each failed test case with its intention by scanning the JUnit report.



```
@Test
public void test_returnEuro() {
    System.out.println("test_returnEuro Test if pricePerMonth returns Euro...") ;
```

Figure 4.7: Possible way of documenting the test intention in the test case

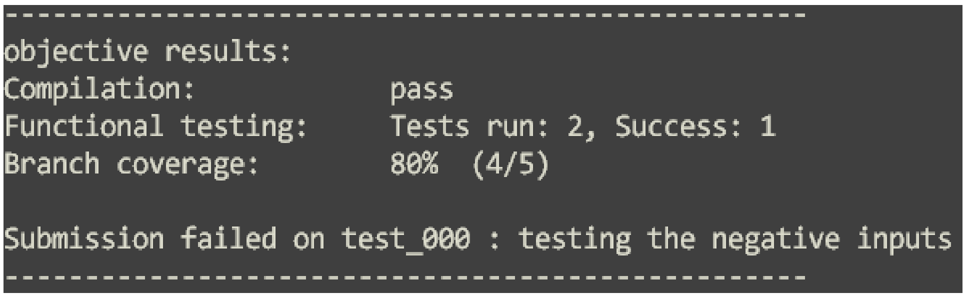


```
------------- Standard Output ---------------
test_returnEuro Test if pricePerMonth returns Euro...
test_roundUp Test if pricePerMonth rounds up correctly...
------------- ---------------- ---------------
Testcase: test_roundUp(SubscriptionTest):   FAILED

junit.framework.AssertionFailedError:
    at SubscriptionTest.test_roundUp(SubscriptionTest.java:21)
```

Figure 4.8: The JUnit report

There are three pieces of reports generated by running Ant script. They correspond to the complier, JUnit and Emma respectively. We use regular expressions to find interesting data from each report and organize them into an objective result (a sample is shown in Figure 4.9).



```
-------------------------------------------------
objective results:
Compilation:            pass
Functional testing:     Tests run: 2, Success: 1
Branch coverage:        80%  (4/5)

Submission failed on test_000 : testing the negative inputs
-------------------------------------------------
```

Figure 4.9: Objective results sample

As we can see the objective result is straightforward and informative for students:

- The compilation result shows if the AUT is compiled on the automated objective testing tool.
- The functional testing result shows how many tests are executed and passed.
- The branch coverage result shows how many branches are in the AUT and how many branches are actually executed.
- The intention of failed test cases is displayed.

Finally, the objective result should be stored in the Moodle database.

4.3.3. Adding a new assignment type in Moodle

In this section, we will take the design of Section 3.4.2 and show how to implement a new assignment type in Moodle.

4.3.3.1. Prerequisites for new assignment type

According to the Moodle assignment type development guidelines [37], a new assignment type should have a corresponding directory[7] under the Moodle system. All of the files for this new assignment type need to go in the directory. The most important file is the assignment type definition class. It will contain the main code for the new assignment type.

[7] <Moodleroot>/mod/assignment/type/PLUGINNAME

There is an assignment base class in Moodle (as can be seen in Figure 4.10). It defines a list of functions that can be invoked when specific events are encountered, for instance, when students want to show the assignment description, a function called 'view' will be invoked. The assignment type definition class should be inherited from the assignment base class, so that functions from the base class can be reused in the assignment type definition class. However, in order to implement a unique assignment type, we do not intend to reuse all functions from the base class. There are three functions that need to be overridden to have customized behaviors:

- 'Setup_elements' function, which is invoked when we construct the form for instructor to add assignment details.
- 'Upload' function, which is invoked when the student uploads the assignment.
- 'view_upload_form' function, which is invoked when we construct the upload form for students.

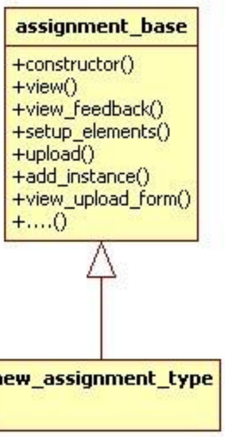
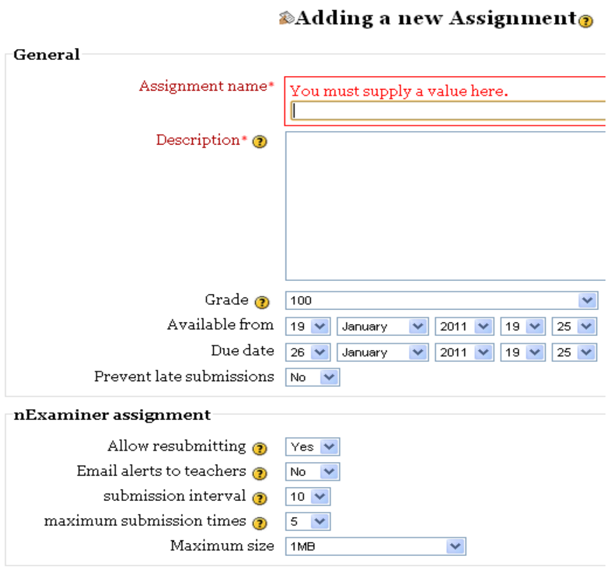


Figure 4.10: Class diagram of assignment base class in Moodle

4.3.3.2. Setup_elements function

In Section 4.3.1, we find that hard-coding the properties (e.g. the required time interval and maximum submission times) into the extended upload feature is a maintenance nightmare in the long run - each assignment might require different property settings in terms of the instructor's preference or educational requirements. Thus, making properties adjustable is our new requirements. We find the best way to achieve this is through the setup_elements function in the new assignment type definition class.

The setup_elements function is able to add extra items to the page that the instructor uses to display assignment details and allows those items to be stored in the Moodle database. Therefore, we add two items for setting a time interval and a maximum number of submission times in the setup_elements function. When the instructor provides new assignment details, they can adjust these properties and then store them in the Moodle database along with other assignment details (as shown in Figure 4.11).



Adding a new Assignment

General

Assignment name*	You must supply a value here.
Description*	
Grade	100
Available from	19 January 2011 19 25
Due date	26 January 2011 19 25
Prevent late submissions	No

nExaminer assignment

Allow resubmitting	Yes
Email alerts to teachers	No
submission interval	10
maximum submission times	5
Maximum size	1MB

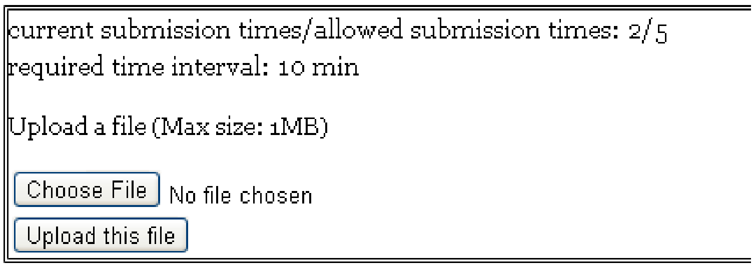
Figure 4.11: Adjusting extra properties when providing assignment details

As a result, the required time interval and the maximum number of submissions are adjustable for every assignment with the new assignment type. And since they will be stored in the database along with other assignment details, they can be referenced by other functions or components (e.g. extended upload feature) through DML at anytime.

4.3.3.3. View_upload_form function

When students intend to upload a solution, the view_upload_form function is called to construct an upload form for students. The upload form we expect is different to the usual upload form. Besides normal functionalities that an upload form should have (e.g. upload field, submit button), we want to convey the information of current submission times, allowed submission times and the required time interval, so that the students can clearly be aware the situations they are in.

Therefore, we use the DML language to query the database to get related information, organize them into a view_upload_form function and build the upload form. The result is shown in Figure 4.12.



current submission times/allowed submission times: 2/5
required time interval: 10 min

Upload a file (Max size: 1MB)

Figure 4.12: Upload form generated by view_upload_form function

4.3.3.4. Upload function

In Section 3.4.2, we presented the upload behavior of the new assignment type. We will describe how to encapsulate the upload behavior into the upload function in this section.

First, we show an activity diagram that illustrates the whole process of upload function in Figure 4.13

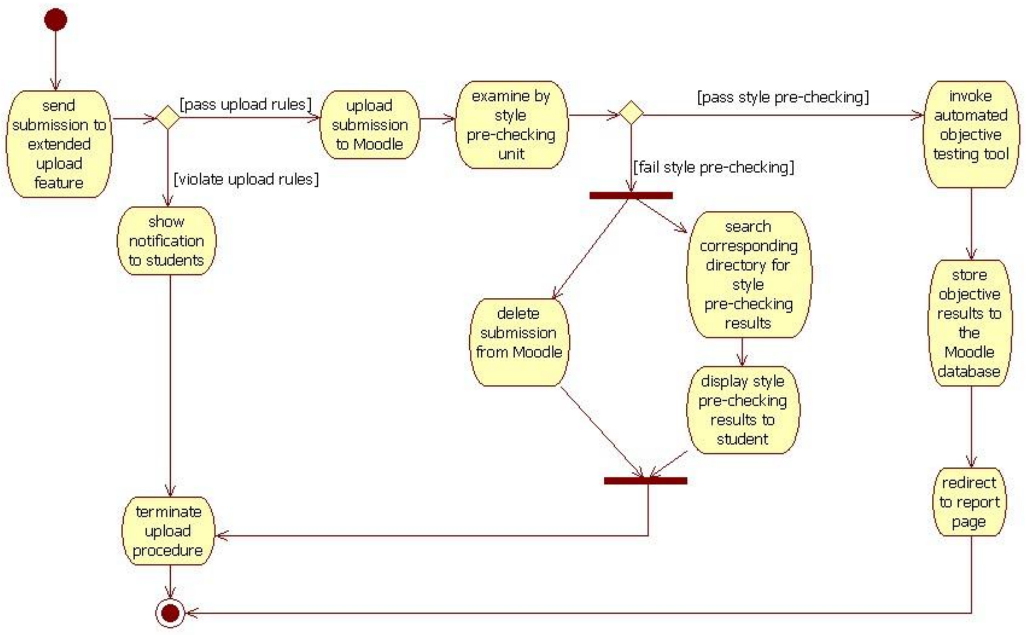
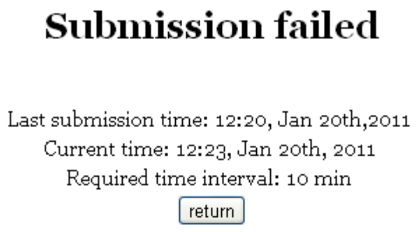

send submission to extended upload feature
[pass upload rules]
[violate upload rules]
upload submission to Moodle
examine by style pre-checking unit
[pass style pre-checking]
[fail style pre-checking]
invoke automated objective testing tool
show notification to students
search corresponding directory for style pre-checking results
delete submission from Moodle
display style pre-checking results to student
store objective results to the Moodle database
redirect to report page
terminate upload procedure

Figure 4.13: Activity diagram of the upload function

When students use the upload form to submit the solution, the upload function will be invoked:

- Initially, the upload function will use the DML to retrieve assignment details from the Moodle database. We are particularly interested in the extra properties set on the assignment (e.g. required time interval), since these properties will be referenced by the extended upload feature to check predefined rules hold(as discussed in Section 4.3.1), if any rule is violated, the upload function will show a notification to students (as shown in Figure 4.14). Then, the upload procedure will be terminated.



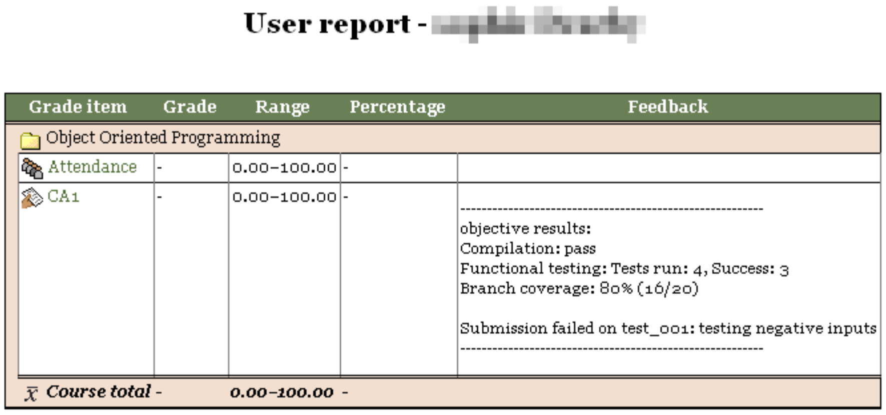
Submission failed

Last submission time: 12:20, Jan 20th,2011
Current time: 12:23, Jan 20th, 2011
Required time interval: 10 min
return

Figure 4.14: Submission notification when time interval violated

- If the submission passes via the extended upload feature, it will be uploaded onto Moodle. The uploaded solution is then examined by style consistency using the style pre-checking unit. The result of the style pre-checking unit is either a webpage that shows style violations or a signal that indicates no style violation is found (as presented in Section 4.2). Thus, we can use the signal to decide whether style checking passes or not. If the signal is not received, which means style violations are found in the submission, we will search the corresponding directory for the style violation displaying page and display it to the student. We will then remove the uploaded solution from Moodle and terminate the upload function. Otherwise, the submission passes the style consistency checking and can be sent to the automated objective testing tool.

- If the submission passes the style pre-checking unit, a request to the automated objective testing tool (presented in Section 4.3.2) will be sent. The request passes the directory information of the submission so that automated objecting tool will know where the AUT can be referenced. After the tool is finished the objective testing, the objective result will be immediately stored in the Moodle database using DML.
- Finally, the upload function will redirect the user to the report page (as shown in Figure 4.15), so that the students can view immediate objective results right after they submit their solution.

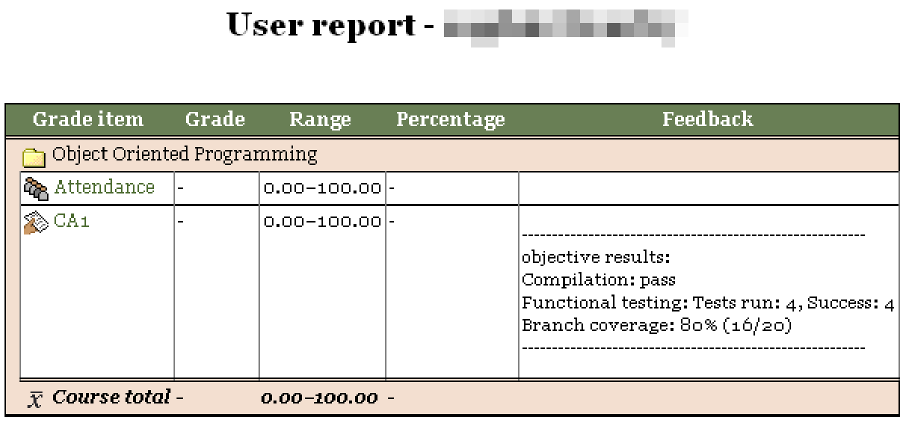


User report - [redacted]

Grade item	Grade	Range	Percentage	Feedback
Object Oriented Programming				
Attendance	-	0.00–100.00	-	
CA1	-	0.00–100.00	-	-- objective results: Compilation: pass Functional testing: Tests run: 4, Success: 3 Branch coverage: 80% (16/20) Submission failed on test_001: testing negative inputs --
Course total -		0.00–100.00	-	

Figure 4.15: Immediate objective feedback on the user report page

The students can keep uploading their work using the upload form until either they satisfy their solutions or they are restricted by the nExaminer framework. The objective result of each submission will replace the last objective result, as shown in Figure 4.16.



User report - [redacted]

Grade item	Grade	Range	Percentage	Feedback
Object Oriented Programming				
Attendance	-	0.00–100.00	-	
CA1	-	0.00–100.00	-	-- objective results: Compilation: pass Functional testing: Tests run: 4, Success: 4 Branch coverage: 80% (16/20) --
Course total -		0.00–100.00	-	

Figure 4.16: Updated objective feedback for the next submission

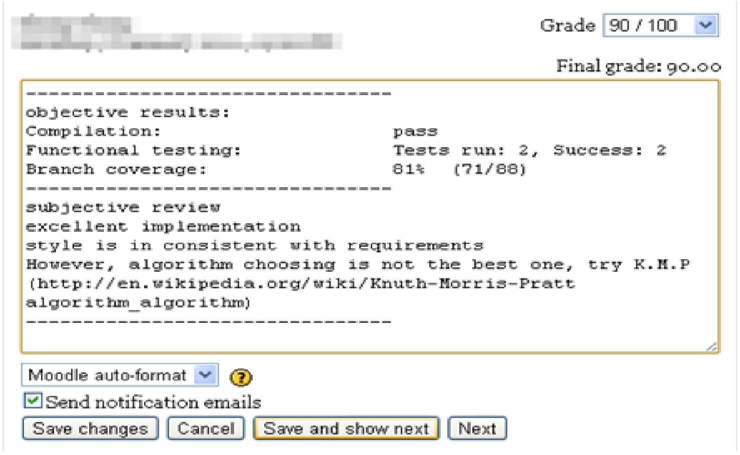
In conclusion, we integrate automated objective testing with Moodle through the new assignment type as we designed in Section 3.4.2. Therefore, students can receive immediate objective feedback without the participation of the instructor while still using Moodle as the platform for sharing knowledge. Besides, the new assignment type also integrates with the extended upload feature and the style pre-checking unit so that their functionalities can be fully drew on by Moodle.

4.4. Automating increased code coverage

In Section 3.5, we propose using DSE to automate code coverage increasing. Due to the complexity of DSE and our time limitation, we were unable to implement this feature on our own. Instead, we use an available tool that implements dynamic symbolic execution in nExaminer framework to automatically increase code coverage. The tool that we use is called AgitarOne [14] (AgitarOne ensures maximum branch coverage and is described in Section 2.5.2).

A set of test cases can be generated with maximum branch coverage for the AUT using AgitarOne. The instructor can then subjectively review each test case to determine its validity or put them under model solution to see how the model solution behaves under such test cases. A test case failure in the model solution can be either a functionality mismatch which has been already identified by objective testing (and should be separate from the subjective review) or a new behavior that the AUT has but the model solution does not. These need more careful review by the instructor. If there is code that remains unexecuted under the generated test cases, the instructor can determine them as unreachable. This could be discouraged through feedback to the student.

Consequently, the task of increasing code coverage is accelerated by the facility provided by the nExaminer framework, as it allows resources to be distributed to other subjective tasks that are difficult to automate. When other subjective tasks have finished, the instructor can update subjective feedback on Moodle and give the final grade (as shown in Figure 4.17).


Grade 90 / 100

Final grade: 90.00

```
---------------------------------
objective results:
Compilation:                      pass
Functional testing:               Tests run: 2, Success: 2
Branch coverage:                  81%  (71/88)
---------------------------------
subjective review
excellent implementation
style is in consistent with requirements
However, algorithm choosing is not the best one, try K.M.P
(http://en.wikipedia.org/wiki/Knuth-Morris-Pratt
algorithm_algorithm)
---------------------------------
```

Moodle auto-format

☑ Send notification emails

Figure 4.17: Update the subjective feedback on Moodle

In Section 4, we presented the implementation of the nExaminer framework to meet our design in Section 3. In Section 4.1, we built the sample code transformer with the assistance from Japa. In Section 4.2, we drew on Checkstyle to implement the style pre-checking unit. In Section 4.3.1, we constructed the extended upload feature using the DML. In Section 4.3.2, we implemented automated objective testing tool using Ant, JUnit and Emma, then integrated it with Moodle through the new assignment type of Moodle as described in Section 4.3.3. In Section 4.4, we also demonstrated how to use AgitarOne to automate increased code coverage. Next, we will perform an experiment to assess the performance of nExaminer framework in Section 5.

5. Results

In order to evaluate the performance of nExaminer framework on Moodle, we use it with a Java programming assignment activity. There are some aspects we want to evaluate in particular:

- If the students' solution can be automatically tested on Moodle.
- The impact of automated objective testing on students.
- The impact of style pre-checking and automating increased code coverage on subjective reviewing.

First, we invited two students with a similar academic background (introductory) and one teacher assistant (TA) to participate in the experiment. The students were both asked to install Checkstyle into their IDE and to follow given style rules template, but only one of them was provided with sample code (described in Section 3.2).

After their assignments were submitted, the student assisted by the sample code was sent to the nExaminer framework to perform automated objective assessment using pre-defined tests. Whereas in control team, the one without was objectively assessed in a traditional way with the TA's participation. The results are shown in table 1.

	nExaminer framework team					Control team
Submitted number of times	5					1
Time Interval from last submission	N/A	16 min	5 min	11 min	70 min	N/A
Objective testing results (identified defects / assessing time)	Rejected	Unable compile	Rejected	2/0.193	0/0.081	1/54
Code coverage (statement)	N/A	N/A	N/A	93% (28/30)	93% (28/30)	100%

Table 1: Objective testing experiment results

The results show that the control team stops doing further attempts after submission. When we asked the reason, the student stated it was because of self-confidence of the solution. And the data proves that this confidence was reasonable and justified - there was only one fault found in the solution (by the TA in 54 seconds).

The student that was assessed by nExaminer framework challenged himself 5 times and eventually constructed a solution that passed pre-defined tests and is style-correct:

- The first solution was rejected at first for style violation.
- After 16 minutes, the student built a solution that passed style pre-checking but was unable to compile on automated objective assessment tool of nExaminer framework.
- The student resubmitted immediately (5 minutes later), which was rejected since the introduced time interval rule was invoked via the extended upload feature.
- The fourth submission was 15 minutes later than the second submission, which passed the time interval rule. 2 faults were found in the solution in 0.193 second and 2 out of 30 lines were unexecuted by pre-defined tests.

- Finally, after 70 minutes, a solution without functionality errors was turned in and evaluated in 0.081 second and unexecuted codes are the same as last submission.

The student in the nExaminer framework team claimed having no difficult to understand the given sample code. The fourth and fifth submission of this student clearly show that student who was given sample code provided solutions that can be automatically tested in less than 1 sec. This saves considerable time in objective assessment (compared to 54 sec in the control team). In addition, the decreasing number of faults in the fourth and fifth submission demonstrates that the student indeed revised their work to be better. Whereas the same code coverage of the last two submissions can be a sign of student's concentration on revising functionalities aspects.

Moreover, style violations were still found in the solution even though Checkstyle was applied. Based on the time spend between first and second submission, it was likely that the student turned the Checkstyle off while programming. Furthermore, subsequent attempts did not show the style violation again. This can be a sign that the student was paying attention to the style consistency.

We also interpret the total of five submissions as an acceptance of the nExaminer framework on Moodle, as this demonstrates its capability to motivate student to revise.

However, as the third submission was rejected and not uploaded onto Moodle, we are unable to determine the solution differences between the second and third submissions. This implies thoroughness of behavior inspection but needs further experiment to do more sampling. However, the transition from not being able to compile to being able to compile implies that the student indeed modified the solution.

Finally, we sent the final solution from nExaminer framework to the TA to subjectively evaluate it in different environment: traditional (control team) first, and then using the nExaminer framework.

	nExaminer framework team	Control team
Time spend on observing new behavior of the AUT	4 min	21 min
Number of identified new behavior	1	0
Time spend on style assessment	0 min	0 min
Total time of subjective reviewing	15 min	40 min

Table 2: Subjective testing experiment results

We can see from table 2: in the control team, the total time spent on subjective reviewing was 40 minutes and the time spent on observing new behaviors was 21 minutes. Thus, 19 minutes was spent on assessing other subjective aspects. The TA concluded that no new behaviors was found in the AUT and explicitly stated that the style of the AUT was in the expected manner already. This facilitates reviewing other aspects (e.g. identify algorithm) and feedback generating.

When the TA used the nExaminer framework team to subjectively review the same assignment, a new behavior was identified as valid behavior in 4 minutes and report was sent to the instructor to improve the assignment description. 11 minutes later, the TA generated subjective feedback.

The time spending on observing new behaviors of assignment in the control team is more than five times than that spent using the nExaminer framework team. We suggested that the time was consumed mostly by manually increasing code coverage in the control team. The TA confirmed that.

Furthermore, over 60% ((40-15) / 40) of the time was saved on subjective reviewing through using the nExaminer framework. However, we doubt whether if the subjective feedback generated from the control team was reused while reviewing in the nExaminer framework since the same assignment was reviewed in different environment by the same TA. Therefore, we expect more TAs to participate in the experiment. Nevertheless, the results shows that time saved on some aspects of subjective reviewing (style, observe new behaviors) was considerable. If future experiment allow, we would like to see how this time is distributed to other teaching activities.

However, while using AgitarOne to generate high coverage test cases, the TA questioned its capability of maintaining maximum code coverage, which leads to our discussion in future work (Section 6.2.2).

In conclusion, the results of this small-scale experiment are promisingly revealed the performance of nExaminer framework on Moodle. The nExaminer framework not only motivate students to reach higher standard of grading criteria but also semi-automatically help instructor in both subjective and objective reviewing, which saves a great amount of time that has potential to be distributed to activities that cannot be automated (or are difficult to automate). However, we also feel the sampling on such small-scale experiment might not be enough, it is better to arrange a large-scale and long term experiment to fully test the nExaminer framework, which we will describe such an experiment in Section 6.2.1.

6. Conclusion

In this paper, we present nExaminer framework that provides solutions to address problems in traditional programming learning on Moodle. Through its modified automated objective assessing and facilitated subjective assessing, the learning interaction between instructor and students is made easier on Moodle. Moreover, we believe the semi-automated nature of nExaminer framework can be further enhanced and drew on. In the sections that follow, we will critically analyze our work, and then present possible future work.

6.1. Critical analysis and our conclusions

As discussed in Section 2.3, Textual comparison is simple and yet effective in early automated assignment assessment [6, 7]; however, it has predominated problems which prevent its progress - It is overemphasis on formatting and heavily relies on input/output which seems a bad practice for students in the long run.

Then, testing frameworks such as JUnit are applied in assignment evaluation automation [5]. It allows students to relax their emphasis on either output formatting or making Input/output work properly. Alternatively, students can concentrate on assignment specification and improve non-functional aspects. But the testing framework is only controlled by the instructor who should wait until all students submit their assignment and then perform evaluation in a batch mode. Thus, students have to wait for their peers and cannot make progress at their own pace.

The prevalence of web technology results testing framework can be accessed by both students and instructors [8, 9]. Among these, the most similar work to ours is Web-CAT [9]. We both deploy the testing framework on the web. But Web-CAT emphasizes that students should be given the responsibility of providing test data to fully test their own code. This ensures high code coverage of submitted solutions. Their assumption is that students have a better understanding of solutions they built than the instructor. This is indeed a valid point; however, we feel this test-driven development enforced by Web-CAT will only provide feedback that is of limited help for students to debug their solutions. To find out more about errors in their own programs, it will be necessary for students to write new test cases which seem an endless practice. We believe students do their best work when they concentrate on one thing. Thus, we narrow students' responsibility to focus on meeting specification, while using automatically generated objective results and facilitating subjective review to allow instructor to perceive behaviors of the AUT efficiently. As our experiments shown, the nExaminer framework we designed motivates students to meet the specification through continuously revising. Meanwhile, the performance of observing new behaviors in solutions is enhanced through provided facility, and has potential to be more accurate, which we will discuss in Section 6.2.2.

Furthermore, to effectively draw on web technology, we integrate the nExaminer framework with the Moodle VLE to add new capabilities into Moodle: automated objective assessment and semi-automate subjective assessment (subjective aspects like style and observe new behaviors require the subjective opinion of the instructor). As a result, the new capabilities of Moodle assist the cognition domain of Bloom's taxonomy (illustrated in Section 2.4) while the instructor and students can still use Moodle as the platform that contributes to constructivism theory (described in Section 1). In addition, the results show multiple submissions in the nExaminer framework, which are a good proof of its acceptance.

The work of TRY [6] motivates our design. Its designers suggest penalizing students for failures after a threshold for submissions in order to help students inspect their coding behavior thoroughly before the next evaluation. To achieve the same goal, we take an alternative solution that adds extra extensions when students upload assignments. However, the available time prevents us from holding a comprehensive experiment that reveals its implication. Further experiments are expected (discussed in 6.2.1).

We also extend the work of Autograder [5], which is a framework developed for Java data structure course. Since data structures are often abstracted to common interfaces, the assignments can be presented to students as a task of implementing provided abstract interface, which will make each student's code testable by the same set of tests using JUnit. However, we suspect that such abstract concept cannot be understood by students from all academic backgrounds. Therefore, we propose a more general solution – sample code, which achieves the same effects but does not emphasize on the concept of interface. We also provide facility in the framework to guide students and instructors on how to generate such sample code. The results prove that this sample code cause no difficulty of understanding for students but also allow solutions having testability.

In [5], the author also requires students to be cautioned about programming style while coding their assignment and apply style checking tool while students program their solutions. However, students might be annoyed by the style violation warning and turn the tool off. Therefore, we design the style pre-checking unit to double check style violation. If style violation is found, students will be informed explicitly and are advised to upload later with a revised solution. After the style pre-checking unit is applied, we found style violations are gradually dropped in submitted solutions.

FrontDesk [8] is another web-based automated assessment system. It provides a portal for instructors to construct feedback in a consistent format. This feature will be discussed in Section 6.2.3.

Finally, we admit that we trade the time spent on assignment design (test case design, model solution design and etc.) in exchange of semi-automated assignment assessment. However, the implication can be far reaching – not only students and instructors can benefit from the nature of semi-automated evaluation, a well designed assignment might also contribute to the modern pedagogy that benefits one generation.

6.2. Future work

In this section, we will discuss future work that would improve the performance of nExaminer framework.

6.2.1. Large-scale and long-term experiment

Aspects described below are difficult to evaluate in such short-term and small-scale experiment that describes in Section 5:

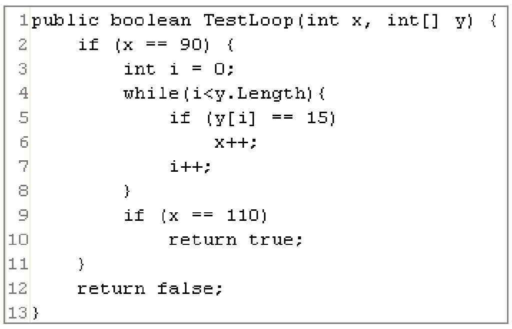
- Students' satisfaction of subjective feedback.
- Students' altitude towards style pre-checking.
- Students' altitude towards sample code.
- To what extent internal structure of test cases is shown in feedback, so that students will not hard coding solutions while being inspired and motivated to revise their work.

Thus, we expect the opportunity of applying nExaminer framework on the long-term experiments of a large-scale introductory programming course.

6.2.2. Accuracy of dynamic symbolic execution

We illustrated in Section 5 that the TA questioned whether DSE can provide maximum code coverage of given code. If we cannot ensure that, we can guarantee neither if there are new behaviors in the AUT nor whether unexecuted code is reachable or not. This causes inaccuracy of the subjective review.

After careful background research, we find that the doubt highlighted by the TA is reasonable. Traditional DSE is insensitive to the indirect relationship between programming constructs. For example, consider the program shown in Figure 6.1. Executing Line 10 needs line 6 to be executed at least 20 times, which infers the array y should contains at least 20 elements of value 15. Such subtle indirect relationships between programming constructs is difficult to bridge using traditional DSE whose first instinct is to try new random inputs when test case generation fails [13]. This random nature leads to uncertainty of DSE to complete its task of reaching maximum code coverage eventually.



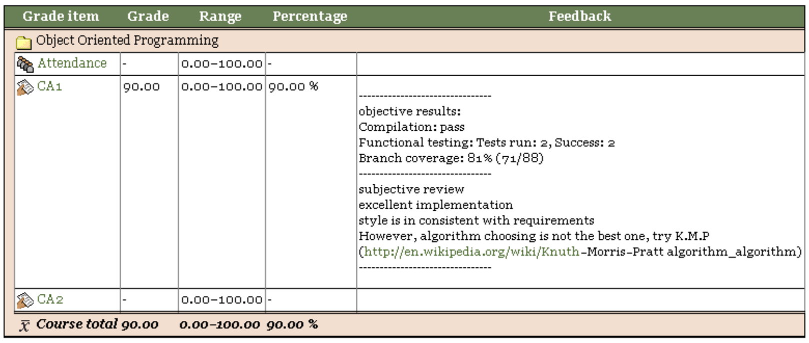
```
1 public boolean TestLoop(int x, int[] y) {
2     if (x == 90) {
3         int i = 0;
4         while(i<y.Length){
5             if (y[i] == 15)
6                 x++;
7             i++;
8         }
9         if (x == 110)
10            return true;
11    }
12    return false;
13 }
```

Figure 6.1: Code snippets

Various techniques are proposed [15, 16] and we would like to assist to increase the accuracy of DSE.

6.2.3. Consistent feedback

In this paper, we draw on Moodle to decrease complexity of managing resources. However, because of time constrain and original layout of Moodle, we found subjective and objective feedbacks are eventually twisted together (shown in Figure 6.2). Moreover, subjective review does not have a consistent format to intuitively tell what has been evaluated. Consequently, students might have difficult to understand generated feedback.



Grade item	Grade	Range	Percentage	Feedback
Object Oriented Programming				
Attendance	-	0.00–100.00	-	
CA1	90.00	0.00–100.00	90.00 %	-------------------------------- objective results: Compilation: pass Functional testing: Tests run: 2, Success: 2 Branch coverage: 81% (71/88) -------------------------------- subjective review excellent implementation style is in consistent with requirements However, algorithm choosing is not the best one, try K.M.P (http://en.wikipedia.org/wiki/Knuth–Morris–Pratt algorithm_algorithm) --------------------------------
CA2	-	0.00–100.00	-	
Course total	*90.00*	*0.00–100.00*	*90.00 %*	

Figure 6.2: User report generated by the nExaminer framework in Moodle

We have seen previous work in generating consistent feedback [8], and would like to tailor Moodle reporting page to have similar or better effects in the near future.

6.2.4. Code visualization

As discussed in Section 2.2.2.1, when reviewing the structure of an assignment, the hierarchy and the relationships between classes are difficult to understand immediately. We believe code visualization is a good start point. Visualization has many good qualities that are specific to our goal:

- Inspiring to encourage the creativity of the recipient.
- Intuition to supports analysis and reasoning.

There are tools for code visualization [21, 22, 31]. However, they are either too complex for our work or not sufficient in the way of maintaining accurate relationship of two classes [23]. Thus, we decide to leave it for future work.

6.2.5. Handling database discrepancy among VLEs

To allow nExaminer framework integrating with VLEs other than Moodle in the future, we believe handling database discrepancy is inevitable since the database of each VLE might be different from one to the other and nExaminer framework requires intensive interaction between its components and the database. Thus, we propose the database manipulation module. It can have four components (the corresponding class diagram is shown in Figure 6.3):

- Configuration component that describes essential properties for database context component to connect database.
- Database context component that defines a list of database operations that are visible to the users and dispatches database operation requesting to database instance component.
- Storable interface that bridge database context and database instance component, defines how should the context component talk to the instance component, and guide how a

database instance component should be built.

- Database instance component that actually performs database operations.

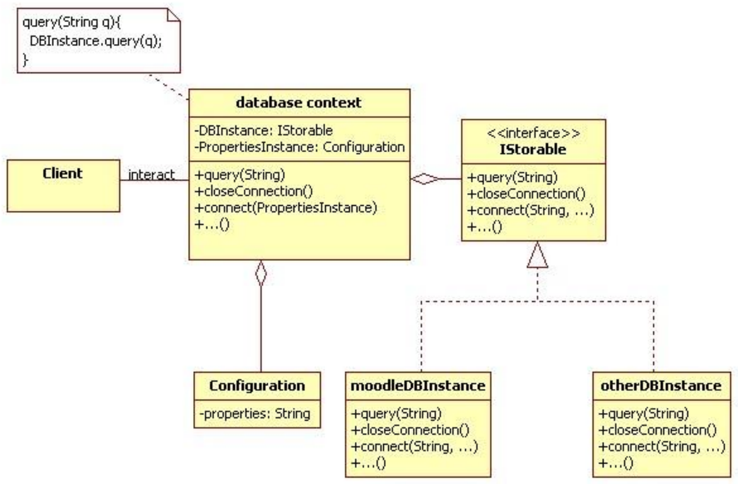


Figure 6.3: Class diagram of database manipulation module

By doing so, database discrepancies are encapsulated into the individual class. Thus, users can select appropriate class to interact with database without disclosing to the internal complexity when VLE changes.

We would like to see the suggestions presented in Section 6.2 are implemented in the nExaminer framework in the near future.

Reference

[1] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y.B. Kolikant, C. Laxer, L. Thomas, I. Utting, T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. SIGCSE Bulletin, Volume 33 Issue 4, 2001.

[2] H. Yeh. The use of instructor's feedback and grading in enhancing students' participation in asynchronous online discussion. Advanced Learning Technologies (ICALT ‘05), 2005.

[3] M. Ben-Ari. Constructivism in computer science education. ACM SIGCSE Bulletin, Vol. 30, 1998.

[4] G.A. Dafoulas. The role of feedback in online learning communities. Advanced Learning Technologies (ICALT’ 05), 2005.

[5] M.T. Helmick. Interface-based programming assignments and automatic grading of Java programs. Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE ’07), 2007.

[6] K.A. Reek. The TRY system -or- how to avoid testing student programs. Proceedings of the twentieth SIGCSE technical symposium on Computer science education (SIGCSE ‘89), Vol. 21, No. 1, 1989.

[7] D. Jackson. Grading student programs using ASSYST. Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education (SIGCSE ’97), Vol. 29, 1997.

[8] M. Maxim, A. Venugopal. FrontDesk: an enterprise class Web-based software system for programming assignment submission, feedback dissemination, and grading automation. Advanced Learning Technologies, 2004.

[9] S.H. Edwards. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. Proceedings of the International Conference on Education and Information Systems: Technologies and Applications (EISTA '03), 2003.

[10] W. Visser. Test input generation with Java PathFinder. Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '04), Vol. 29, 2004.

[11] T. Xie, D. Marinov, W. Schulte and D. Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. Lecture Notes in Computer Science, 2005.

[12] Y. Kannan, K. Sen. Universal symbolic execution and its application to likely data

structure invariant generation. Proceedings of the 2008 international symposium on Software testing and analysis(ISSTA '08), 2008.

[13] P. Godefroid, N. Klarlund, K. Sen. DART: directed automated random testing. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05), 2005.

[14] M. Boshernitsan, R. Doong, A. Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. Proceedings of the 2006 international symposium on Software testing and analysis (ISSTA '06), 2006.

[15] T. Xie, N. Tillmann, J. de Halleux, W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. Dependable Systems & Networks(DSN '09), 2009.

[16] E. Alba, J.F. Chicano. Software Testing with Evolutionary Strategies. Lecture Notes in Computer Science, Vol. 3943, 2006.

[17] D.A. Norman. Cognitive artifacts in J. M. Carroll (Ed.), Designing interaction: Psychology at the human-computer interface. Cambridge University Press, 1991.

[18] A.V. Aho, J.D. Ullman. The theory of parsing, translation, and compiling. Prentice-Hall, 1972.

[19] E.J. Chikofsky, J.H. Cross. Reverse engineering and design recovery: a taxonomy. Software, IEEE, Vol. 7, No. 1, 1990.

[20] I. Hernan-Losada, C. Pareja-Flores, A.J. Velazquez-Iturbide. Testing-Based Automatic Grading: A Proposal from Bloom's Taxonomy. Advanced Learning Technologies (ICALT '08). 2008.

[21] S.C. Eick, J.L. Steffen, E.E. Sumner. Seesoft-A Tool for Visualizing Line Oriented Software Statistics. IEEE Transactions on Software Engineering, Vol. 18, No. 11, 1992.

[22] J.I. Maletic, A. Marcus, L. Feng. Source Viewer 3D (sv3D) - a framework for software visualization. Proceedings of the 25th International Conference on Software Engineering, 2003.

[23] Y. Guéhéneuc, H. Albin-Amiot. Recovering binary class relationships: putting icing on the UML cake. ACM Conf. on OO Programming Systems Languages and Applications, 2004.

[24] http://Moodle.org/. Last accessed 23rd of January 2011.

[25] http://download.oracle.com/Javase/1.3/docs/tooldocs/solaris/Javap.html.Last accessed 23rd of January 2011.

[26] http://checkstyle.sourceforge.net. Last accessed 23rd of January 2011.

[27] http://docs.Moodle.org/en/Development:DML_functions. Last accessed 23rd of January 2011.

[28] http://ant.apache.org/. Last accessed 23rd of January 2011.

[29] http://www.junit.org/. Last accessed 23rd of January 2011.

[30] http://emma.sourceforge.net/. Last accessed 23rd of January 2011.

[31] http://metrics.sourceforge.net. Last accessed 23rd of January 2011.

[32] D.R. Krathwohl. A Revision of Bloom's Taxonomy: An Overview. Theory into Practice, Vol.41, No. 4, 2002.

[33] http://code.google.com/p/Javaparser. Last accessed 23rd of January 2011.

[34] D.A. NORMAN. Emotional Design: Why We Love (or Hate) Everyday Things. Basic Civitas Books, 2003.

[35] P. W. Oman, C. R. Cook. A paradigm for programming style research. SIGPLAN Notices, Vol. 23 No. 12, 1988.

[36] L. Malmi, A. Korhonen, R. Saikkonen. Experiences in automatic assessment on mass courses and issues for designing virtual courses. Innovation and technology in computer science education (ITiCSE '02), 2002.

[37] http://docs.Moodle.org/en/Development:Assignment_types. Last accessed 23rd of January 2011.

[38] L.J. Arthur. Measuring Programmer Productivity and Software Quality. John Wiley, 1985.

[39] A. Hayes, P. Thomas, N. Smith, K. Waugh. An investigation into the automated assessment of the design-code interface. Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE '07), 2007.

[40] M. Roper. Software Testing. The McGraw-Hill Companies. 1994.

[41] http://www.junit.org/apidocs/org/junit/Assert.html. Last accessed 23rd of January 2011.

[42] G.J. Myers, C. Sandler. The Art of Software Testing. John Wiley & Sons. 2004.

Appendix A. Bloom's taxonomy

Bloom's taxonomy identifies three domains of leaning. Each domain has its learning objectives. In the most known of cognitive domain (shown in Figure A), the objective is to remember what has being taught [32]. In order to achieve ultimate objective, a list of activities is presented as series of levels or prerequisites, which suggests that one cannot effectively address higher levels until those below have been managed.

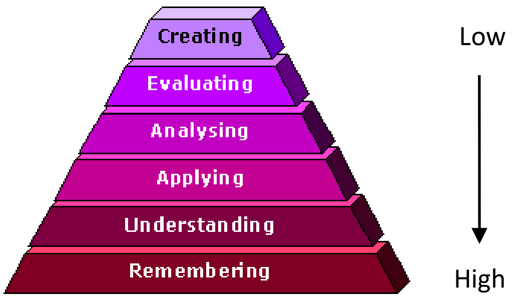
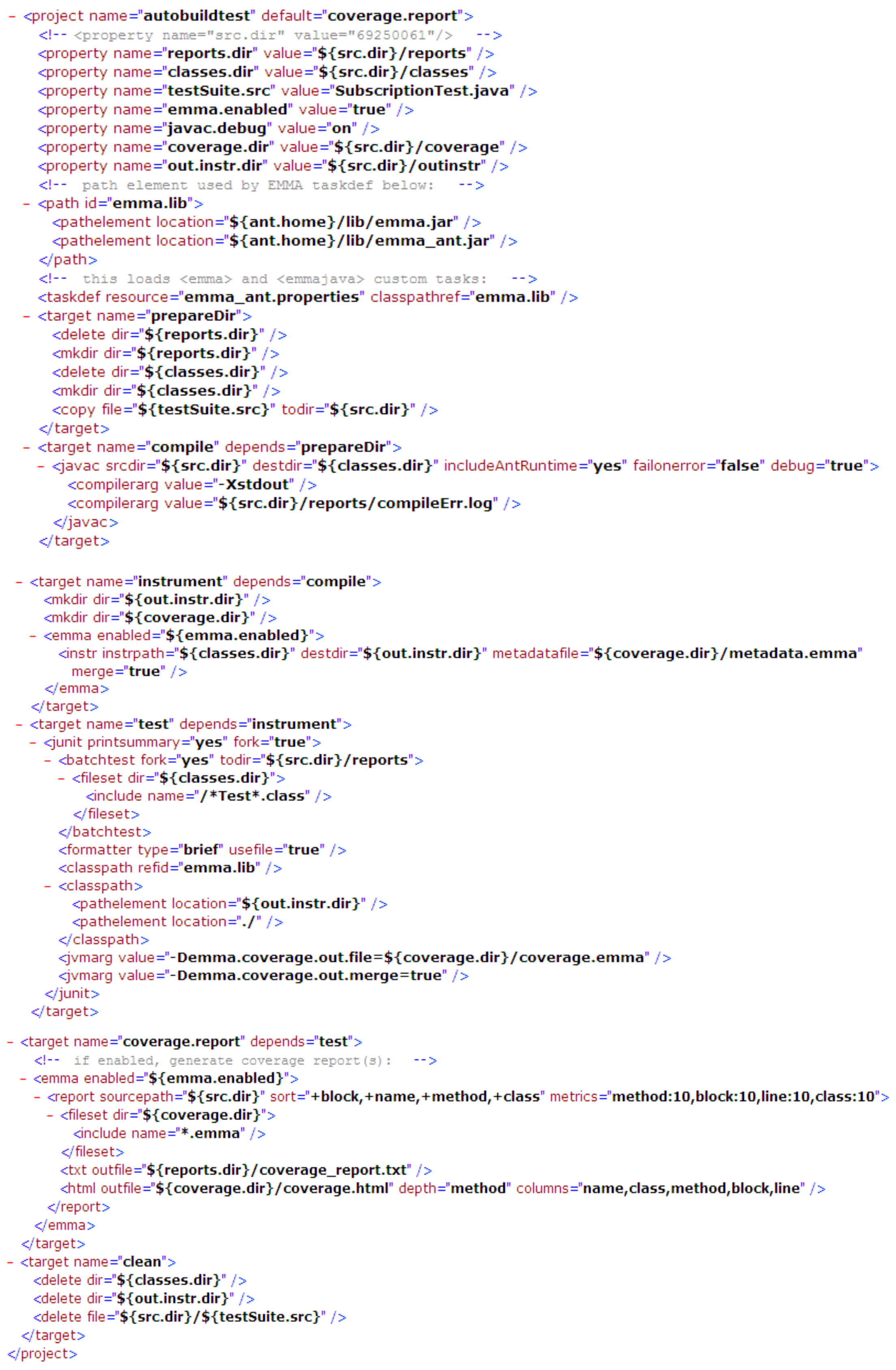


Figure A: Cognitive domain of Bloom's taxonomy[8]

[8] Revised taxonomy of the cognitive domain following Anderson and Krathwohl (2002)

Appendix B. Default Ant script



```
- <project name="autobuildtest" default="coverage.report">
    <!-- <property name="src.dir" value="69250061"/>   -->
    <property name="reports.dir" value="${src.dir}/reports" />
    <property name="classes.dir" value="${src.dir}/classes" />
    <property name="testSuite.src" value="SubscriptionTest.java" />
    <property name="emma.enabled" value="true" />
    <property name="javac.debug" value="on" />
    <property name="coverage.dir" value="${src.dir}/coverage" />
    <property name="out.instr.dir" value="${src.dir}/outinstr" />
    <!-- path element used by EMMA taskdef below:   -->
  - <path id="emma.lib">
      <pathelement location="${ant.home}/lib/emma.jar" />
      <pathelement location="${ant.home}/lib/emma_ant.jar" />
    </path>
    <!-- this loads <emma> and <emmajava> custom tasks:   -->
    <taskdef resource="emma_ant.properties" classpathref="emma.lib" />
  - <target name="prepareDir">
      <delete dir="${reports.dir}" />
      <mkdir dir="${reports.dir}" />
      <delete dir="${classes.dir}" />
      <mkdir dir="${classes.dir}" />
      <copy file="${testSuite.src}" todir="${src.dir}" />
    </target>
  - <target name="compile" depends="prepareDir">
    - <javac srcdir="${src.dir}" destdir="${classes.dir}" includeAntRuntime="yes" failonerror="false" debug="true">
        <compilerarg value="-Xstdout" />
        <compilerarg value="${src.dir}/reports/compileErr.log" />
      </javac>
    </target>

 - <target name="instrument" depends="compile">
     <mkdir dir="${out.instr.dir}" />
     <mkdir dir="${coverage.dir}" />
   - <emma enabled="${emma.enabled}">
       <instr instrpath="${classes.dir}" destdir="${out.instr.dir}" metadatafile="${coverage.dir}/metadata.emma"
         merge="true" />
     </emma>
   </target>
 - <target name="test" depends="instrument">
   - <junit printsummary="yes" fork="true">
     - <batchtest fork="yes" todir="${src.dir}/reports">
       - <fileset dir="${classes.dir}">
           <include name="/*Test*.class" />
         </fileset>
       </batchtest>
       <formatter type="brief" usefile="true" />
       <classpath refid="emma.lib" />
     - <classpath>
         <pathelement location="${out.instr.dir}" />
         <pathelement location="./" />
       </classpath>
       <jvmarg value="-Demma.coverage.out.file=${coverage.dir}/coverage.emma" />
       <jvmarg value="-Demma.coverage.out.merge=true" />
     </junit>
   </target>

- <target name="coverage.report" depends="test">
    <!-- if enabled, generate coverage report(s):   -->
  - <emma enabled="${emma.enabled}">
    - <report sourcepath="${src.dir}" sort="+block,+name,+method,+class" metrics="method:10,block:10,line:10,class:10">
      - <fileset dir="${coverage.dir}">
          <include name="*.emma" />
        </fileset>
        <txt outfile="${reports.dir}/coverage_report.txt" />
        <html outfile="${coverage.dir}/coverage.html" depth="method" columns="name,class,method,block,line" />
      </report>
    </emma>
  </target>
- <target name="clean">
    <delete dir="${classes.dir}" />
    <delete dir="${out.instr.dir}" />
    <delete file="${src.dir}/${testSuite.src}" />
  </target>
</project>
```