

APPENDIX A: JAVA SPECIFICATION CHECKING: SOFTWARE NOTES

Appendix A: Software Notes and Student Use

This appendix contains notes on the CheckM250 and Compilation Helper software.

The project has identified four layers of correctness in Java code:

1. Syntax (a shorthand for whether the code compiles or not)
2. Structure (structural specification)
3. Semantics (functionality shown by methods)
4. Style (non-functional requirements for reusability, readability and similar concerns).

The focus has been on implementing and evaluating code to assess Layer 2, using software developed for this project called CheckM250, although a Compilation Helper tool has also been developed to support students in understanding errors encountered in Layer 1.

Section 1 describes relationships to similar work.

Section 2 explains more about how the CheckM250 software works, including:

- interactions between the identified layers of correctness;
- the specification language;
- design choices in assessing structural correctness;
- generating help on compilation errors;
- working with CodeRunner in the VLE, and incorporating CheckM250 in that context.

Section 3 provides some notes on ideas for future development, including known limitations and design choices.

Section 4 provides references.

Appendix 1 includes emails between the author and Michael Kölling.

1 Relationship to similar work

This project was initially inspired by the Harvard computer programming module CS50 [1], which provides an environment for students to submit solutions to C and Python exercises and have them assessed automatically for certain features. CS50 has developed tools called Check50 (for Python) and Style50 (for the C language), allowing students to perform checks on their code before it is submitted for final marking. The marking tool developed for this project has been called CheckM250, in recognition of CS50's approach.

The structural specification checking software developed in this project is similar to that described by Kiraly et al in their MeMOOC platform [2], although Kiraly et al incorporate style and functionality testing, which was not explored in this initial prototype of CheckM250. Significantly, CheckM250 is less tightly coupled: its core code is (unlike the MeMOOC software's) fixed, and makes use of a separate specification file to customise the testing. This has advantages for use in online marking and makes CheckM250 more reusable.

2 CheckM250 Software

There are many testing tools that can be deployed in the context of assessing Java code quality, but this project focussed particularly on *structural* specification testing, by which is meant testing for the

presence of particular features in students' code, rather than its conformance to expected behaviour.

The tutor version of the software was implemented using Java SE 7 (we assume that students' code will be built using Java SE 7, as this is the supplied JDK on M250), and Java SE 8 was used to modify CheckM250 for use on the VLE.

I begin with a review of the layers of correctness we assess in students' code, and their interactions.

2.1 Syntax

Syntax checking is carried out by a compiler, in our case `javac` (Java SE 7), which both tutors and students have access to. The same compiler is available in our online teaching environment (although here it is Java SE 8) and this provides access to correctness checking at this level.

To succeed at this stage, students grapple with the syntax of the language, with what constitutes a well-formed statement, how tokens may be legally used with each other, and how to combine these features to form a solution to meet a program specification.

The Java compiler provides detailed and potentially confusing feedback on compilation errors. In addition to CheckM250, this project has also developed a Compilation Helper tool for making compilation error messages friendlier, based on an existing help file adapted (with permission) from the BlueJ software environment, together with new compilation error diagnostic software developed for this project.

2.2 Structure

In the second layer, *structure* refers to a code solution providing various externally and internally visible features, rather than it conforming to a particular behaviour or functionality. Although the structure of code cannot be separated from its syntax, syntactic correctness is only a step on the way to structural correctness. The existence of structural features can be determined after compilation, but without running the student's code.

Not only can we check for the presence of certain methods or constructors in code, but whether a class provides particular instance variables, whether the class extends another class, or whether it implements a particular Java interface.

Structural correctness is therefore a pre-requisite for unit testing, which is mostly concerned with testing methods. It therefore can be used to determine if unit tests can safely proceed.

Code that appears to behave appropriately under unit tests does not necessarily meet structural requirements, although sufficiently careful unit testing might reveal this.

Finally, I have taken the view that code specification is not only about providing expected features, it is also about not providing unexpected features. It is not necessary to specify these 'background' features on a per-problem basis, as their required absence is implicit in the 'foreground' specification. However, there is room for discussion as to the extent to which such background checks should be performed.

Structural correctness was the main focus of this project.

2.3 Functionality

The functionality of code is tested by ensuring that it produces expected outputs for known inputs, which, in the case of Java, means that methods must behave as expected. There are existing tools to

assist in this kind of testing, also known as unit testing, and JUnit in particular is widely used for Java unit testing [3]. Our online CodeRunner environment also provides a simple form of unit-testing facility, although its configuration is quite laborious.

Although it might be possible, with extra effort, to configure functionality testing code to cope with an incorrect structural specification, typically this layer of testing assumes that the structural specification is correct.

Writing good unit tests is not a trivial task, but it is one that we have tools to support us with already, so it is not the main focus of this project.

Traditionally on M250 we explain to students how methods should behave, and sometimes we provide fragments of code to check functionality. Students are also expected to perform their own informal testing at this level.

2.4 Style

Style forms the final explicit layer of our assessment of correctness, in that we ask tutors to check for particular issues of style in students' assignments. Two solutions that are functionally identical may or may not follow our house style and they may be more or less readable or maintainable.

Automated style checking can be supported through tools such as Checkstyle [4] and PMD [5]. We do not currently use tools to mark style, but many tutors are aware of them. We expect that students will develop a good style over a period of time and so we use mostly formative feedback here.

There are many comprehensive external style guides, for example, Google's [6]. M250's own Code Conventions guide provides a basis for further development in this area for automated marking [7], and some work has been undertaken on developing a style sheet for use with Checkstyle.

In terms of dependencies, style is a somewhat separate issue to the other layers. We consider it to be a final layer of polish on a solution, which may be developed through refactoring or reworking of a fully structurally and functionally correct solution.

Feedback on tutors' concerns with regards to style was elicited via the survey and interview process in this project. More insight into this question is provided in Appendix B: Interview and Survey Results.

2.5 Overview of the structural checking algorithm

For our purposes, structural correctness, semantic correctness and stylistic correctness all initially require that a student's code compiles successfully.

Structural specification testing can be performed statically on the compiled code (without running the code), whereas testing of code functionality would be performed dynamically.

In testing structural correctness I am separating the programming interface between the software and a test harness from how the code beneath this interface achieves a desired outcome.

CheckM250's tasks are modified according to a specification file, which differs for each assignment set.

The CheckM250 software has the following requirements in order to function fully:

1. Required Java files must exist in the solution. The path to locate `.java` files depends on the context in which the code is running, BlueJ or the VLE.

2. Required files must successfully compile. The path to locate `.class` files depends on the context in which the code is running.

If the required files do not exist, the marking tool reports this.

If the solution classes do not compile, the marking tool generates some feedback to indicate this (and in the case of the VLE, the Compilation Helper tool generates help in achieving compilation.) This is not a limitation of the CheckM250 software so much as a limitation of marking by class reflection, which is the same approach used by several other tools. If this occurs, the tutor has to mark the code manually or the VLE cannot provide additional feedback to a student (other than help on compilation errors).

Once the solution code compiles, it is possible to dynamically generate `.class` (bytecode) files, load them dynamically into the runtime execution environment, read the specification file, parse it, and then build up a picture of the features that should be present in the candidate solution code. CheckM250 then iterates over the features that the specification requires, generating some form of output for each feature according to whether it matches the specification or not and returning that to the context from which the marking code is being run, so that some feedback can be displayed to the user.

2.6 The specification language

This section outlines the specification language used in this project. Although this is of current interest, the future direction of work suggested for the structural checking software may make this approach redundant. However, it is a key innovation in the design of this code that the specification of a solution is separated from the code that checks the specification of the solution, and this will be a feature of any further versions of this software.

The design of the specification file was deliberately simplistic, and took into account that at some stage students might have access to the file. The current design discourages a student copying and pasting into their solution. However, in the case of use in the VLE (in CodeRunner) the student does not have access to the specification file in any case. Tutors viewing the specification file is not a concern.

The default name for the specification file in the VLE is `checks.txt`. In the tutor-facing tool, the specification file has no fixed name and can be selected.

Currently the question setter produces the `checks.txt` manually.

2.6.1 Example specification for a TMA01 question

Here are the complete contents of a specification file for the one question in TMA01:

```
class:public/M250Account/java.lang.Object/  
field:private/java.lang.String/M250Account.accountNum  
method:public/java.lang.String/M250Account.getAccountNum/void  
method:public/boolean/M250Account.isValidLength/java.lang.String  
method:public/boolean/M250Account.isValidStart/java.lang.String  
method:public/boolean/M250Account.isValidDigits/java.lang.String  
method:public/boolean/M250Account.isValidAccountNum/java.lang.String  
method:public/void/M250Account.setAccountNum/java.lang.String  
constructor:public/M250Account/void
```

The interpretation of the specification is as follows. We require:

1. a public class called `M250Account`, which is in the default package (hence we have used its simple name). It extends `Object` (i.e., it doesn't need to explicitly extend any class), but (note the trailing slash), it does not implement any interfaces;
2. a private instance variable of type `String` called `accountNum`, belonging to class `M250Account`. Note its fully qualified name `M250Account.accountNum`;
3. methods `isValidLength`, `isValidStart`, `hasValidDigits`, `isValidAccountNum` and `setAccountNum`. All these methods except `getAccountNum` require single arguments of type `String`. All belong to the class `M250Account`, so we must specify their full names. (Conceivably, methods with the same name could be required in different classes.) Note the fully qualified name for the library class `java.lang.String`, whereas primitive types returned by the methods have simple names: `boolean` and `void`;
4. a public, zero-argument constructor. Note the use of `void` to indicate no arguments (see also `getAccountNum`). This may mean the question calls for a zero-argument constructor, or it does not call for any constructors, in which case the class will have this constructor provided to it automatically when it is compiled (a Java language feature).

It is not a lengthy process to hand-write a specification file such as this, though from a programmer's point of view this is not an ideal approach. However, manual creation of this file has the advantage that it requires the question setter to inspect the question carefully, and this can lead to detection of missing information in the question, a useful side-effect.

Formalising the specification further would allow testing if a specification file is valid. It would be interesting to explore this, for example using ANTLR [8], but the objective here was to keep the description simple and facilitate parsing.

The specification file enables automated checking of structural correctness, but also has the important role of separating the specification from the code that evaluates the specification.

2.6.2 Notes on the specification file

Only classes whose names appear in the specification file are assessed. This means that any support classes (i.e., ones that students do not alter) are not included in the `checks.txt` file.

If students are required to modify a given class, its specification must include both code that we require students to write and code that is supplied, as otherwise supplied code will appear extraneous when compared with the specification. This is a design choice, as unexpected features in solution classes are flagged as errors.

The order of lines in the specification file is not significant.

Each line begins with an initial token, namely `class`, `method`, `constructor` or `field`, followed by a colon. The initial token identifies the kind of specification on the line. Any line beginning with a different token is ignored, so e.g. a line beginning `comment :` will not become part of the specification and can be used to make a remark. Each line ends with a newline character.

There are several separator tokens:

- A **colon** is used to separate the kind of specification from its specification, e.g. `class:classSpec`
- **Spaces** separate modifiers, e.g. `static final`
- **Dots** separate parts of fully qualified names, e.g. `test.ClassName`
- **Commas** separate arguments in argument lists, e.g. `int,int,String`. Note, there are no parameter names specified at this time. There are no spaces between parameters. Commas are also used to separate multiple implemented interfaces.
- **Forward Slashes** separate parts of a class, method, constructor or field specification
E.g. `public/void/package.Y.method3/int,int`

1 Reference type specifications

All reference type names must be fully qualified, not simple. The exception is that where a class is in the default package we must necessarily use its simple name.

This applies to both user-defined and library classes, so for example `String` must be fully qualified and we would write `java.lang.String`. This is necessary to avoid name conflicts and to allow the specification checking software to identify the intended class or type correctly.

Currently generic types (bound and unbound) are not fully supported. See Section 3.2 for notes on this.

2 Primitive type specifications

Primitive types are specified as is, e.g. `void`, `int`, or `char`.

The keyword `void` is also used to indicate empty parameter lists.

More details and examples of each kind of specification are given in the following subsections.

3 Class specifications

- `class:modifiers/package.Name/extendedClass/implementsName,implementsName...`
- `java.lang.Object` must be specified as the extended class if no other superclass is intended.
- If there is no interface to implement, the line ends with a slash `/`
- Class names must be specified in full; for example, we refer to `java.util.Map` not `Map`.
- Multiple implemented classes can be separated by commas.

As noted earlier, support classes, by which we mean classes that students do not need to alter, are not specified in `checks.txt`. CheckM250 only examines classes specified in `checks.txt`.

4 Method specifications

- `method:modifiers/return_type/package.methodName/argument,argument...`
- Modifier lists may be empty, which means the line starts `method: /`, but normally a line would start `method:public/`
- Return types must be specified in full if they are classes; primitive types can be specified as is, including `void`.
- Arguments are separated by commas, using full names for classes.
- Specify `void` when there are no arguments.

5 Constructor specifications

Constructors are similar to methods. Because constructor names must be the same as class names, their parent class names provide sufficient information.

- *constructor:modifiers/package.ClassName/argument,argument...*
- The modifier list may be empty, which means the line starts `constructor: /`
- The constructor name is implied from the (owning) class name
- Use full names for classes as argument types.
- Specify `void` when there are no arguments.

The default zero-argument constructor must be specified explicitly (even though it doesn't appear in the class explicitly). This is because at runtime the class will nevertheless have this constructor, and we are using reflection. As a result, CheckM250 may report that a solution has a zero-argument constructor missing the modifier 'public', because Java has provided a default constructor behind the scenes, though it is not present in the student's code. This may require explanation to users.

6 Field specifications

Fields include instance variables, static variables, constants, and static constants.

- *field:modifiers with spaces/type/package.Class.name*
- Modifier lists may be empty, which means the line starts `field: /`
- Extra modifiers are separated by spaces.
- Give the full name of the field, including the class it belongs to.
- Reference type names must be given in full.

CheckM250 is not able to check initial values of fields. This is a fairly complex problem of itself, and it is debatable whether it falls under the scope of structural specification.

2.6 Compilation helper software

Personal communication with the main author of the BlueJ IDE (Michael Kölling) led to permission to use the `javac.help` file included with the BlueJ 3.1.4 distribution in producing feedback on compilation error messages. The original `javac.help` file has been substantially changed for reuse in this project and is currently named `javac_1.help`. The help file was modified to add help where none existed, and to suggest alternative reasons for particular errors arising that were not covered. Some edits were made for style of the language and formatting.

A Gnu Licence applied to the BlueJ source code – however `javac.help` is a plain text file. The need to cite the Gnu licence was, in any case, waived by Michael Kölling [9].

The software that processes the help file was written for this project and attempts to match more specific errors before more general ones, with the aim of providing more targeted advice on compilation error messages. This component invokes the Java compiler dynamically, captures the results, selects the first error (if any) using a canonical name, and then uses a lookup in `javac_1.help` to provide feedback on that error. If there is no error then this component produces a message 'Compiled OK'.

It is worth mentioning that the authors of BlueJ are abandoning this approach in favour of a more modern 'inline' display of error messages in the editor window, and amidst doubts of the effectiveness of this kind of feedback, but I would argue that the approach remains of interest in the context of online quizzes in the Moodle environment, where options for formatting feedback to students are more limited.

2.7 Working with CodeRunner

As the CheckM250 software was originally developed for use by tutors in the BlueJ environment, a number of adaptations were required for it to work in the VLE via CodeRunner questions, for students.

This section outlines important features of CodeRunner for this work, changes required to adapt the core CheckM250 code to the CodeRunner question type and reasons for decisions made in this context.

2.7.1 About CodeRunner

CodeRunner [10] is a question type for Moodle Quizzes, developed by Richard Lobb of the University of Canterbury, New Zealand.

1 Question types

There are several CodeRunner question types described in the CodeRunner documentation [11]:

java_program. Here the student writes a complete program which is compiled then executed once for each test case to see if it generates the expected output for that test. The name of the main class, which is needed for naming the source file, is extracted from the submission by a regular expression search for a public class with a `public static void main` method.

java_class. Here the student writes an entire class (or possibly multiple classes in a single file). The test cases are then wrapped in the main method for a separate public test class which is added to the student's class and the whole is then executed. The class the student writes may be either private or public; the template replaces any occurrences of `public class` in the submission with just `class`. While students might construct programs that will not be correctly processed by this simplistic substitution, the outcome will simply be that they fail the tests. They will soon learn to write their classes in the expected manner (i.e. with `public` and `class` on the same line, separated by a single space)!

java_method. This is intended for early Java teaching where students are still learning to write individual methods. The student code is a single method, plus possible support methods, that is wrapped in a class together with a static main method containing the supplied tests (which will generally call the student's method and print the results).

For structural specification testing we have used both the **java_program** and **java_class** types of question. A significant difference is that **java_program** type questions can include `import` statements. However, **java_class** covers our needs at this stage.

2 CodeRunner's approach to marking

CodeRunner uses the output of any code executed in a test (in the case of Java, it makes use of output sent to the `System.out` stream) to determine whether a test has been passed or not. The question-setter indicates what the expected output should be for each test. If the test output matches the expected output, the test is passed, otherwise it is failed. It is possible to set hidden tests and to hide subsequent tests on failure of a previous test.

To test structural specification, or other non-functional tests we require, we can invoke a method to run the test code, and specify the expected output of the test when evaluating a correct solution.

To make use of this approach, the user-interface and BlueJ plugin code for CheckM250 were replaced with two single-method interfaces that would produce either an 'OK' response, or feedback suitable for displaying to students in the VLE.

The CheckM250 code required alterations to cater for the flat file structure in the CodeRunner environment, which affects where generated files can be expected to be found.

Support files were added to the CodeRunner question, and the Java classpath was altered to find these files as necessary. The specification file (`checks.txt`) does not require any modifications for the use in the CodeRunner environment.

Each test in CodeRunner is completely independent of any others, as the sandbox in which the code runs (Jobe [12]) is cleared after each test. It is therefore not possible to rely on the presence of a generated `.class` file from one test to another. This means that the testing code in this environment must ensure that `.class` files are recreated for each test.

3 Check and Precheck modes

Since CodeRunner version 3.1, a question author can provide students with a **Precheck** button [13], which students are not penalised for using, so allowing for formative feedback. This also provides authors with the chance to run some pre-tests on code before running standard test code, which is (typically) assessed. The **Check** button is used for assessed tests on students' code.

A CodeRunner question must be configured to enable **Precheck**, and individual tests must be specified for use in either **Precheck** mode, **Check** mode, or Both **Precheck** and **Check**.

The combination of **Precheck** and **Check** modes means that we can provide a 'safe' environment to students under **Precheck** and then go on to execute assessed tests under **Check**.

4 Template code

A template class is used to provide scaffolding for the marking code, and has access to some system variables, including `STUDENT_ANSWER`, which represents the code a student included in the answer box in this environment. In a typical CodeRunner question, the student's answer is incorporated as is, in the template, as a method or inner class¹.

5 Reuse of CodeRunner questions

It was reassuring to note that a CodeRunner question can be successfully duplicated, or exported and imported from one quiz to another, with all support files being correctly retained.

2.7.2 CodeRunner settings and template code

We can consider three cases for the student's code:

1. compiles and provides expected features;
2. compiles, but does not provide expected features;
3. does not compile (so does not provide expected features).

We do not know when assessing a solution which of these cases will occur.

If case 2 or 3 occurs then the default is that `javac` compilation errors would be displayed to the student. In case 2 this is because CodeRunner tests cannot be compiled, and in case 3 additionally

¹ A class embedded in another class

the student supplied code does not compile. Either case is likely to result in multiple errors being displayed in the feedback the student receives.

We do not wish to subject students to compilation errors unnecessarily, which may be confusing; more so because these messages in part depend on the template (testing) code that students do not see and should not have to consider. It is, however, possible to find a middle ground, based on the **Check** and **Precheck** options in CodeRunner v3.1.0.

The solution adopted was to develop the Compilation Helper tool, and deploy it under **Precheck**. This tool captures the first error that the compiler finds and then provides some feedback on it, to suggest how the student's answer might be fixed. Although I cannot guarantee that this advice is helpful (interpretation of compiler error messages is not straight-forward, and there may be many different possible causes for each kind of error), it should be more friendly than a dump of the compiler's error messages.

Although it has not (yet) been implemented, code style feedback could also be implemented as a **Precheck** test – i.e. for formative rather than summative purposes.

1 Question-level settings adopted

Under CodeRunner question type, the **Precheck** mode is selected (Figure 1), so that we can use both **Precheck** and **Check** buttons.

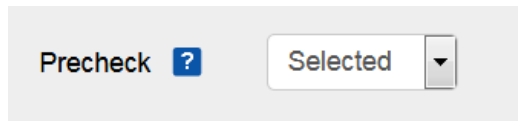


Figure 1: **Precheck** option selected

The penalty regime (Figure 2) was set low to encourage students to keep trying. Students were also allowed to restart their attempt at the quiz level. The use of the '%' mark in the penalty is not required. However, there must be a comma after the second penalty in the regime in order for the ellipsis to operate correctly (Figure 2 is missing this second comma).

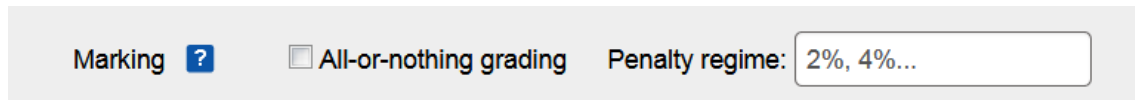


Figure 2: All-or-nothing grading and Penalty regime options

Following recommended practice for CodeRunner [11], our quiz settings required students to pass *all* tests (All-or-nothing should be ticked) run against their code to score *any* marks. The rationale is that buggy code is not acceptable and that students will attempt to achieve fully correct code if we do not reward them with part marks for partly correct code.

The Grading is set to 'exact match' (Figure 3), which is standard. Exact match means the output must exactly match what we expected, which is appropriate in this case. The student's output is therefore either correct or incorrect.

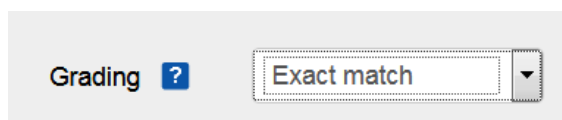


Figure 3: Grading set to Exact match

The user Interface is the standard Ace editor (Figure 4).

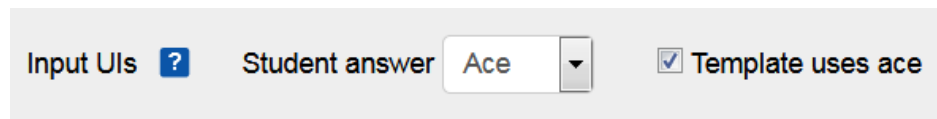


Figure 4: Input User Interface options with Ace selected for template and answer

2 Adding to the classpath in CodeRunner

To access the CheckM250 software jar and the root directory, the path needs altering for the compiler and the interpreter. To do this, in the **Advanced customisation** section **Parameters** box (Figure 5) it is necessary to add parameters:

```
{ "compileargs" : [ "-cp .:CheckM250ForCodeRunner.jar" ],  
  "interpreterargs" : [ "-cp .:CheckM250ForCodeRunner.jar" ] }
```

Note, this is a JSON array, with two keys, `compileargs` and `interpreterargs`.

^ Advanced customisation

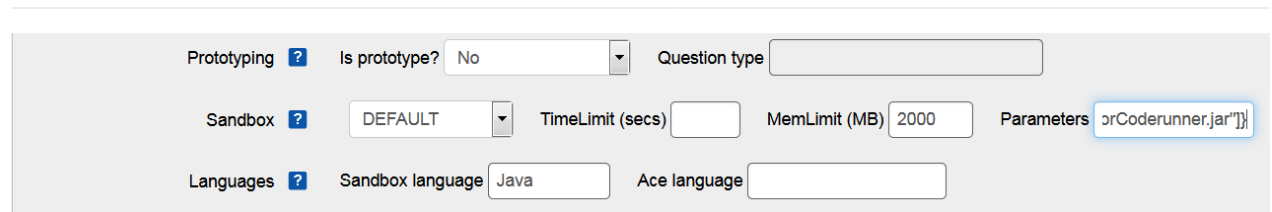


Figure 5: Parameters configuration to use CodeRunner.jar

Note also the use of the Unix path separator, ":", since this code is running on a Unix system.

3 Template Code modifications

1. Import statements

To access the CheckM250 code inside the CodeRunner jar, the `checks` package needs to be imported by the template class:

```
import checks.*; //Includes CheckM250 class and SharedUtils
```

All the auxiliary utility code is in the `CheckM250ForCoderunner.jar`, so no other import statements should be needed to run that code as long as the jar is accessible. This covers both Compilation Help and Specification Check tests.

2. The **Precheck** facility was used to allow students to perform two pre-tests: compiling their code and checking its structural specification. The template was modified to include the following Twig² code:

```
{% if not IS_PRECHECK %}  
    {{ STUDENT_ANSWER | replace({'public class ': 'class '}) }}  
{% endif %}
```

² Twig is a templating engine, allowing for dynamic inclusion of code in this environment

The interpretation of this templating code is that if CodeRunner is not running in **Precheck** mode, the `STUDENT_ANSWER` will be included in the context in which this templating fragment appears (which is within the testing code class body). Additionally, the words 'public class' in the student's answer are replaced by 'class'.

As already noted, including the student's answer in the template is unsafe, as the student's answer is included as is within the testing code, but we only used this approach when CodeRunner is *not* running in **Precheck** mode.

3. Subsequently I have copied the student's answer to a string

```
private static String studentAnswer =
    "{{STUDENT_ANSWER | replace({'public class ': 'class '}) | e('java')}}";
```

Again, the words 'public class' were replaced by 'class'. More significantly, special characters in the `STUDENT_ANSWER` were escaped according to Java escaping rules, so that the `studentAnswer` string is well-formed.

This second construct is safe, in the sense that it can be compiled whether the student's code is valid or not.

4. Further the student's answer was saved to a fixed location:

```
public static void main(String[] args) {
    __Tester__ main = new __Tester__();
    SharedUtils.writeStudentAnswer("Answer.java", studentAnswer);
    main.runTests();
}
public void runTests() {
    {{ TEST.testcode }};
}
```

The method `writeStudentAnswer`, which is in the `checks.SharedUtils` class, takes the escaped student's answer (from the variable `studentAnswer`), and saves it physically to the provided filename, in this case `Answer.java`. The remainder of the code here is standard template code to invoke the question setter's test code.

The shortcut of saving to a fixed file name has been taken, although it may have been possible to infer the correct file name based on the student's answer. At compile time, because the word 'public' has been stripped from the class definition, the compiler accepts this situation. (Otherwise Java rules dictate that the compiler would generate an error because the student's answer was not saved in a file whose name matched its class name.)

Note, I have assumed the student's answer is a complete class (not a method).

5. When running in **Check** mode, the Twig code in part 2 above results in the student's code being literally included in the testing code so that we have direct access to its features for testing purposes. If it turns out that the code does not compile at this point all we can say is that we gave the student the opportunity to work this out in **Precheck**, so now they will have to deal with the messiness of compilation errors.

The fact that in both **Precheck** and **Check** cases we also include the student's answer as a string in our testing code means that we can write the answer to disk, compile it, generate a `.class` file, dynamically load it, and then run our structural checking code on the result, assuming it compiles.

4 CodeRunner Support files

Three support files (Figure 6) are required to make the marking tools work.

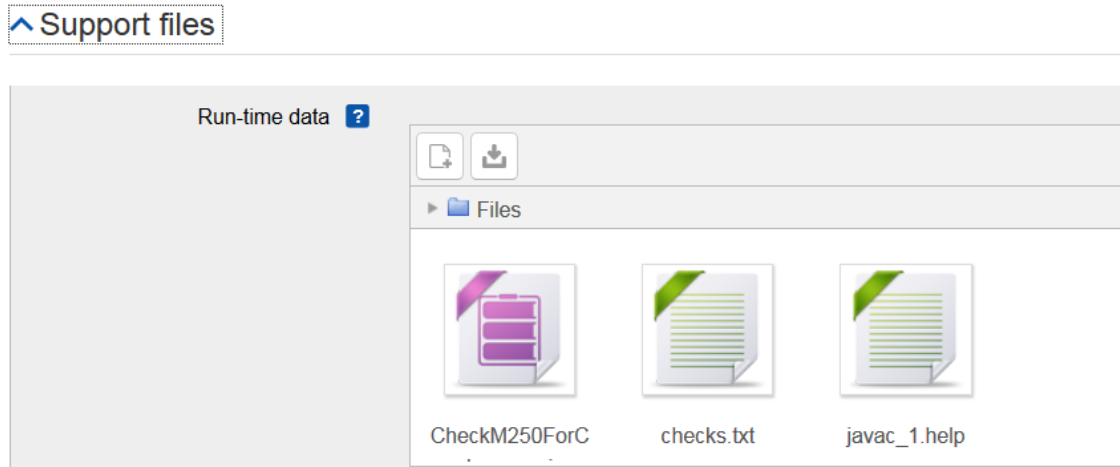


Figure 6: Support files required in CodeRunner

The support files required in the CodeRunner environment are:

1. `CheckM250ForCodeRunner.jar`, which contains all the support software, including compilation help and specification checking code, plus related code such as methods to read and write files in the CodeRunner environment, and in particular the method `writeStudentAnswer()`.
2. `checks.txt`, the structural specification file for the question. This file should only be needed for the specification checking.
3. `javac_1.help`, the modified version of BlueJ's compilation help feedback file. This file should only be needed for the Compilation Helper.

2.7.3 Rubrics

We explain to the students how to use the **Precheck** and **Check** buttons. Unfortunately, **Check** in CodeRunner does not have the same meaning as **Check** in other interactive question types. **Check** in CodeRunner may incur a penalty, whereas **Check** in other question types does not.

1 Example rubric on the use of the Precheck and Check buttons

You can **'Precheck'** your answer without penalty, which checks the specification of your code. When you do this, we will check that you have methods and variables required by the question. This is an experimental feature.

If your code passes the **'Precheck'**, you can **'Check'** your answer next.

Although you can **'Check'** and **'Precheck'** your answer repeatedly, you will only be able to **'Submit all and finish'** once. (To try again, you need to restart the quiz.)

2 Example information on the Compilation Helper tool

This test checks whether compilation succeeds or fails, i.e. whether your code is syntactically correct, not that it meets any other requirements.

If your code compiles correctly, you will see the output 'Compiled OK', and a green tick.

If your code does not compile correctly, you will receive feedback on the first error found in your code. Scroll down to see the feedback.

We hope that the feedback on any errors in your code is helpful, but please note that this is an experimental facility:

We refer to your code as `Answer.java`. Don't be put off by this - it's just a container for your code. You are allowed to include more than one class in the answer box, and you do not have to write a class called `Answer`.

It is often not easy to determine why a particular error occurred, so the help messages you will see are often of a general nature, rather than specific to your code. This is particularly so for certain kinds of errors that occur for a variety of unrelated reasons. Read through the different suggestions and one of them will relate to your code.

For some kinds of error there will not be any help, because they are less common and we haven't implemented any help on them yet. Please let us know if you find a compilation error and no help on it.

Any code you enter in the answer box below must be in a class. A class has this general form:

```
class SomeClassName
{
    //instance variables, methods and constructors here...
}
```

2.7.4 Example Feedback

Example feedback for a failed test

Example feedback when not passing all tests is illustrated in Figure 7. This includes generic feedback:

Your code must pass all tests to earn any marks. Try again.

and feedback for each test, pointing out which tests have been failed.

Using the 'Show differences' button, differences between the student's output and the expected output can be highlighted.

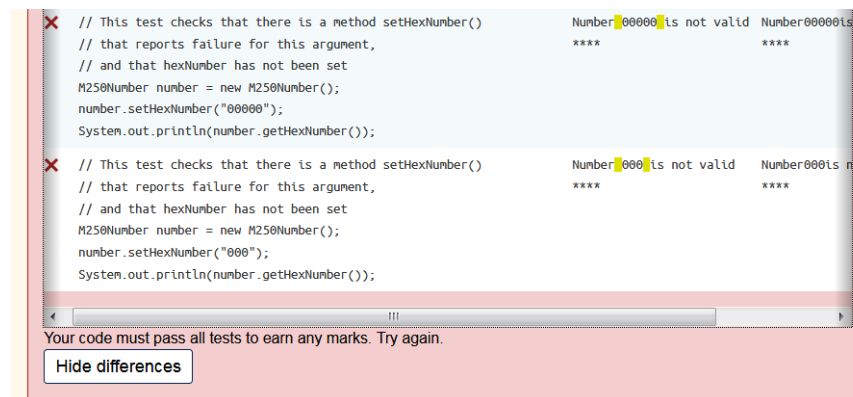


Figure 7: Feedback at the question level, after pressing 'Show differences' button

The highlighting should help the student understand that there are some spaces missing in their own output, which is on the right.

Example quiz level feedback for the 'Practice for TMA03' Quiz

Feedback is also generated via quiz level settings:

We're glad you took the time to do the Practice for TMA03 quiz - we hope it's helped you to get ready to submit your actual TMA03 and it should also help you prepare for question 3 in the exam.

However, this level of feedback is only received by a student if the state of their quiz moves to 'Finished' by their pressing the Submit and Finish button.

2.7.5 CodeRunner tests for specification testing

Currently there are two tests supported by the code developed in this project:

- `SharedUtils.compile("Answer.java")` runs a compilation check, and generates help on compilation errors in the student's answer. This supports Layer 1 checking.
- `CheckM250.specificationCheck()` runs the structural specification check, with feedback on structural specification errors. This supports Layer 2 checking.

The expected output for these tests is `Compiled OK` and `Specification OK`. If the expected output is not seen, the feedback from the tool is shown to the student.

1 mark was awarded for each test (whether specification or functional) in our examples. These marks are then aggregated to determine the question level mark, modified by the penalty regime and how often the student used the **Check** button. The question level mark then contributes to the quiz level mark.

1 Example CodeRunner test case for the Compilation Helper

Figure 8 shows a typical configuration for the Compilation Helper tool.

^ Test cases

The screenshot shows the configuration interface for a test case. It includes the following fields and controls:

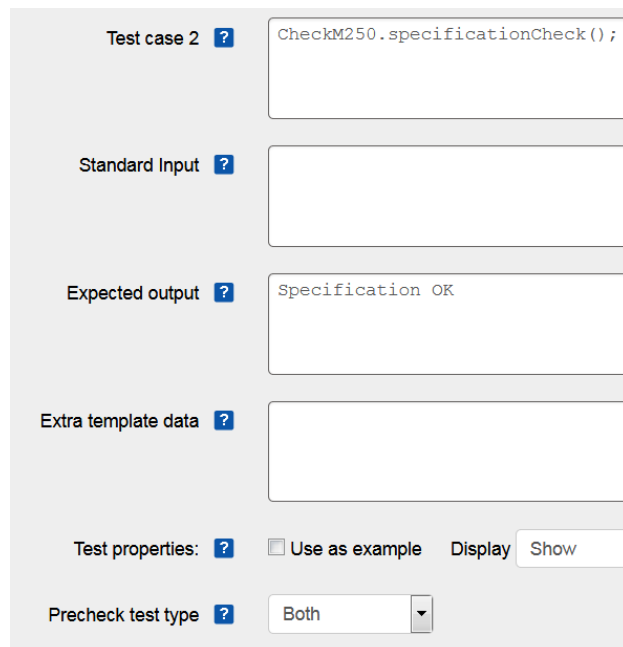
- Test case 1**: `SharedUtils.compile("Answer.java");`
- Standard Input**: (Empty text area)
- Expected output**: `Compiled OK`
- Extra template data**: (Empty text area)
- Test properties**:
 - Use as example
 - Display: `Show` (dropdown menu)
 - Hide rest if fail
 - Mark: `1.000` (input field)
 - Ordering: `0` (input field)
- Precheck test type**: `Both` (dropdown menu)

Figure 8: Typical configuration to use the Compilation Helper component

Note that **Precheck** test type is set 'Both', to run this test for both **Precheck** and **Check** buttons. It is also suggested to tick 'Hide rest if fail', to prevent further tests from running if the student's code does not compile.

2 Example CodeRunner test case for Structural Specification check

Figure 9 illustrates a typical test case to run a structural specification check in CodeRunner.



The screenshot shows the configuration for a test case in CodeRunner. It includes the following fields and options:

- Test case 2**: `CheckM250.specificationCheck();`
- Standard Input**: (Empty text area)
- Expected output**: `Specification OK`
- Extra template data**: (Empty text area)
- Test properties:** Use as example Display
- Precheck test type**: Both (dropdown menu)

Figure 9: Typical configuration to use the structural specification checking component

As for the Compilation Helper, the **Precheck** test type is set to 'Both' **Precheck** and **Check**, so that structural checks are not abandoned during functionality tests. This change is advisable since subsequent functionality tests may succeed even though the structural checks do not (and we still require structural correctness). The option 'Hide rest if fail' should also be ticked to prevent further tests from running in the event of compilation or structural checks failure.

In **Precheck** mode tests marked as '**Check**' are never compiled. In **Check** mode, as configured, both **Precheck** and **Check** tests are compiled and **Check** tests will cause a compilation error if the **Precheck** has not been passed. This is why we have advised students to first pass the **Precheck** before attempting the **Check**.

Future work will explore added provision of code style help in this environment.

3 Example output for a correct solution

Figure 10 illustrates the feedback when pressing the **Precheck** button and both Compilation and Structural specification are correct:

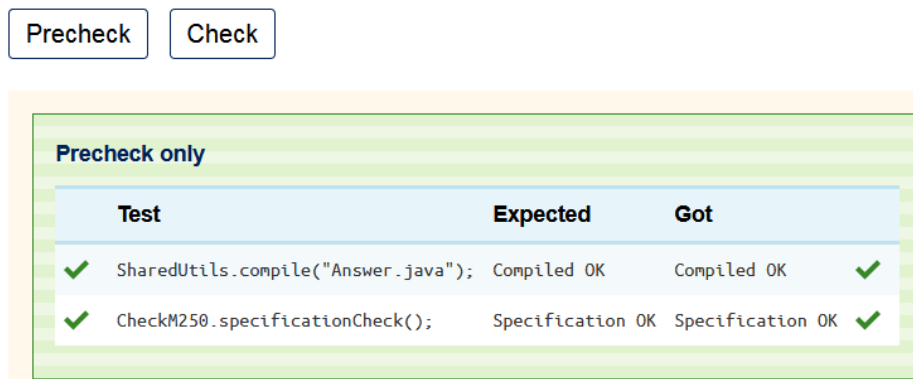


Figure 10: Example output for correct compilation and structural specification

It has been reported that the green ticks can be quite motivating to students and we noted that some students made dozens of attempts before succeeding.

4 Standard functionality test cases

Other tests we wish to run for a question (i.e., standard functionality tests) use the normal CodeRunner approach, predicting the expected output for executing some code. These tests are run under **Check** mode, configured as shown in Figure 11, under the **Precheck** test type.

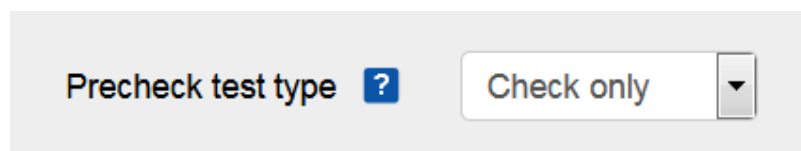


Figure 11: **Precheck** configuration for functionality tests

It is also worth noting that this setting is only honoured when **Precheck** is selected for the question type (see Section 2.7.2).

2.8 Student use in the VLE

This section provides some more evidence for the usefulness of the marking tools in this project in the context of CodeRunner questions and student use in the VLE.

Three formative CodeRunner quizzes were deployed in 2017J and two of the marking tools developed during this project were deployed on two of these quizzes:

- ‘Practice for TMA02’ included the Compilation Helper tool
- ‘Practice for TMA03’ included both the Compilation Helper and the Structural Specification Checking tools.

‘Practice for TMA01’ did not use either of these tools.

We intended that these quizzes would help students prepare for their TMAs, by providing similar questions to parts of their actual assignments. In addition the quizzes provided a platform for testing the software developed through this project.

2.8.1 How many engaged?

Table 1 presents some statistics on the numbers of students who engaged with the formative quizzes in 2017J.

Table 1: Some statistics for M250 formative quizzes in 2017J

	Practice for TMA01	Practice for TMA02	Practice for TMA03
Week opened	4	11	15
Reg at week (N)	1409	1347	1284
All Attempts	917	514	479
Unique Pls	917 ³	492	416
Repeat factor (Attempts / Unique Pls)	1	1.06	1.15
% of Reg25 (N = 1385)	66.2	35.5	30.0
% Attempts / Reg at week	65.1	38.1	37.1
% of total engaged (N = 1012)	90.6	48.6	41.1
Finished state	224	155	124
% Finished / Attempts	24.4	30.2	25.9
Forum postings	403	162	241
Checks	4114	1582	954
Mean Checks / attempt	4.5	3.1	2.3
Max checks	94	80	57
Median Checks	1	1	1
Checks Mode	1	1	1
Checks count Skewness	5.2	6.3	7.6

³ ‘Practice for TMA01’ only allowed 1 attempt per user.

There were 1012 unique IDs in these data sets, representing 73% of the Reg25 cohort taking part – about three quarters of students. About two thirds of Reg25 engaged with the first quiz, and about a third with the second and third quizzes. We do not attribute this drop off in engagement to the use of the CheckM250 software tools on the second and third quizzes; rather we suggest it is the engagement of students with formative quizzes in the later stage of the module that has dropped.

Table 2: Number of formative quizzes attempted by students

Number of formative Quizzes done	N	% of total engaged (N = 1012)	% of Reg25 (N = 1385)
1	439	43.3	31.7
2	330	32.6	23.8
3	242	23.9	17.5

Table 2 shows how many quizzes were attempted by students (e.g. 439 students did just one quiz). Regarding the quizzes on which the new marking tools were deployed:

- 264 students did both TMA02 and TMA03
- 228 did TMA02 but not TMA03
- 152 did TMA03 but not TMA02

A few students continued to use the quizzes after the module had officially finished, as they still had access to the website for some months. These students may have been preparing to retake the module, or simply continuing to practice.

2.8.2 Student postings

After the initial setup, the CheckM250 and Compilation Helper facilities embedded in the ‘Practice for TMA02’ and ‘Practice for TMA03’ quizzes were used alongside normal functionality tests.

There were many examples of students posting their code and asking for help in fixing it, as we intended, as well as general queries on how to interpret results returned by the CodeRunner environment.

Students were also able to compare approaches to solving problems, without fear of discussing an assessed question. Because they were able to share answers, they were also able to copy and paste each other’s code into their own environment and test it immediately, to help each other fix it. There was evidence of less able students learning from more able ones.

Next I have provided some examples of student postings supporting the use of this formative approach to teaching, and illustrating the use of the structural checking and compilation helper tools.

1 Welcoming the chance to practice and the use of Precheck

The comment below mentions the **Precheck** facility (with compilation and specification help) made available in Practice for TMA02 and TMA03, which was not available on summative iCMAs, and also shows appreciation for the opportunity to practice using a formative quiz.

deffo worth the practice I think, even if it is just for the exam. The best way to learn is to practice! I also like the sound of the precheck - it would be useful if the iCMAs could have a precheck also! 😊

<https://learn2.open.ac.uk/mod/forumng/discuss.php?d=2568602#p18489331>

2 On help received, and practising as opposed to reading

Thank you so much for your help ... I do enjoy trying to write code but I often find it difficult to make the leap from the books to a real practical question like these practice questions.

I find I get so much more out of these exercises compared to anything else.

<https://learn2.open.ac.uk/mod/forumng/discuss.php?d=2582065#p18728352>

3 Soliciting help

An example of a student soliciting help on how to write their answer in the form of a class:

can anyone help please ? I do not know (the code for) how to "write " the class M250Number ?

<https://learn2.open.ac.uk/mod/forumng/discuss.php?d=2431765#p17608638>

4 Specification error through misspelling a method name

In this example from Practice for TMA02, the student reports that their code works in Bluej, because they have consistently misspelled a method name, but the code does not meet the question specification and causes a compilation error in CodeRunner:

<https://learn2.open.ac.uk/mod/forumng/discuss.php?d=2480565#p17910340>

*Can some one please help me.
I'm getting error:*

```
__Tester__.java:294: error: cannot find symbol
    e.reduceToldling();
      ^
symbol:   method reduceToldling()
location: variable e of type Engine
1 error
```

here is my code in relation to "reduceTolding()"

Engine class:

```
public void reduceTolding()...
```

The student has misspelled the method name `reduceToIdling`, substituting an `l` for an `I`.

The test code does not compile, so the unit test cannot be run. However, running a structural check on this code would flag that the expected method (`reduceToIdling`) was not found, and another method with an unexpected name (`reduceToldling`) was found.

5 CheckM250 finds error missed by unit testing

A student was alerted to a problem with the specification of their `equals` method – it should have taken an `Object` as a parameter. This was flagged by CheckM250 in Practice for TMA03 (Figure 12).

I have completed the task when i run to check so all is ok with passing it but when i click to pre-check i get this:

CheckM250.specificationCheck();	Specification OK	In class Animal, the required method with signature equals(java.lang.Object) is missing. Check that you have methods with the required names and signatures. Specification errors found: 1
---------------------------------	------------------	--

My equals i created this:

```
public boolean equals(Animal obj)
{
    return (obj.getWeek() == this.getWeek()) && obj.getKind().equals(this.getKind()) && obj.getName().equals(this.getName());
}
```

I am unable to add @Override unless i change the method header to :

```
public boolean equals(Object obj)
```

But if i do that i am unable to do this -> obj.getWeek() // as obj does not have that method (this i know).

So how are we to override the method so it passes the check?

Figure 12: CheckM250 finds an error not picked up by unit testing

Notice the feedback from CheckM250:

In class Animal, the required method with signature equals(java.lang.Object) is missing. Check that you have methods with the required names and signatures. Specification errors found: 1.

The student's query indicates that they have understood that they needed to override the `equals` method and they have made the first step to correct this, then run into a dereferencing problem while attempting to access the method `getWeek`.

Another student responded to the question:

"to override methods, they need to have the same signature, that is, the same method name and argument types, and number of arguments. Otherwise you are overloading them instead.

So have a look at equals in Object, you need to use the same signature to override.

Then have a look in Unit 11. "Writing an overriding equals method" - that hopefully should help 😊"

The student responded

“yep looking at [unit 11](#) i see i was overloading instead of overriding Thank you.

A review of the student’s submissions shows that on their 10th attempt at the problem they changed `equals(Animal)` to `equals(Object)` as required.

The student’s progress suggests an understanding of the separation between the specification of their code and its functionality. The student first fixed the header of the method `equals` and then continued to ask in the forums about how to get the body of the method correct (how to cast the object received to an `Animal` so they could access its methods).

In this case our own tests run under the **Check** part of the question did not check for students overloading rather than overriding `equals()`, and all the functionality tests for this student were passed. However, even though the main **Check** in CodeRunner was passed, the student continued working on the problem until they understood why they had failed the **Precheck** and then fixed it so it passed both **Precheck** and **Check**.

This example also illustrates that code may pass functionality tests, but fail specification tests.

<https://learn2.open.ac.uk/mod/forumng/discuss.php?d=2582065#p18578573>

6 CheckM250 finds wrong return type for method

Figure 13 shows an example from a student with an incorrect return type for their `getAnimals` method, which was flagged by CheckM250 as an error.

```
I'm missing something very obvious, but the Pre Check is not liking:  
  
public Set<Animal> getAnimals()  
{  
    return animals;  
}
```

Figure 13: Method with an incorrect return type flagged by CheckM250

It compiles fine in BlueJ, but not in CodeRunner. Any thoughts? Tips?

The student is actually confusing compiling correctly with specified correctly. The output the student saw from the quiz is shown in Figure 14 and reports that the code ‘Compiled OK’ (here the student has a green tick), but the second output indicates that the answer did not meet the structural specification. This is a separation of concerns in correctness that students often do not grasp.

Test	Expected	Got
✓ SharedUtils.compile("Answer.java");	Compiled OK	Compiled OK ✓
✗ CheckM250.specificationCheck();	Specification OK	In class Shelter, the method getAnimals() should have return type java.util.List ::-(The required methods are present, but their headers aren't quite right ::-(All the required fields are present, but their declarations aren't quite right. Check that the modifiers and types of your variables are correct Specification errors found: 2 ✗

Show differences

Figure 14: CheckM250 feedback on errors in method and field declarations

Another student responded, adding bold formatting to emphasize the key point in the feedback.

As per the error message:

*"In class Shelter, the method getAnimals() should have return type java.util.**List**"
So it seems like you were expected to have created a List of animals, not a Set?*

This led to the student re-reading the question and fixing their answer.

In the student's defence, this was their second attempt at the problem, and they apparently did not notice that the question had changed, as variants were being used. In their first attempt at the problem, the question called for a `Set` to be used. However, this example illustrates two things well:

- i) confusion between compilation and other kinds of correctness;
- ii) the power of the feedback in generating a discussion, and alerting a student to a problem with their code. The two problems identified are that the instance variable should have been based on a `List`, and that the getter method for that variable needed the same type.

<https://learn2.open.ac.uk/mod/forumng/discuss.php?d=2611174#p18777393>

7 Formative quiz helping to pass a TMA

Finally, the feedback below encourages us to continue using formative quizzes:

Hi, i just want to thank everyone especially ... i did very well in the real TMA and without the help completing the practice TMA i would not of had a clue so thanks



<https://learn2.open.ac.uk/mod/forumng/discuss.php?d=2466343#p17839248>

2.8.3 Lessons learned

The Compilation Helper tool can be used in the VLE to provide more friendly feedback to students on compilation errors, so assisting understanding of Layer 1 correctness.

We found that the structural checking tool applied to Layer 2 was able to detect errors that would prevent unit tests from succeeding, to detect errors that unit tests might not detect, and to find errors that markers would comment on for pedagogical reasons.

Tutors had reported that misspelled variable and method names, use of wrapper types for primitive types, incorrect access modifiers and misuse of the `static` modifier were errors they had on occasion not noticed when reading over students' code and these are examples of errors that CheckM250 can pick up in the VLE and advise students on.

We saw that we could help students to make progress towards a correct solution, with peer support. In Example 4, the student's code ran correctly in BlueJ, because they had made an error consistently in their own code. In cases like these, students' own testing will not pick up the error, and specification checking is helpful.

We noted also that some such errors may nevertheless result in code that passes unit tests, such as in Example 5 of Section 2.8.2, failure to override an `equals` method. In this case, our own unit tests had not checked for overloading rather than overriding `equals`, and all the functionality tests were passed.

The use of the `static` modifier could easily be missed by a unit test, but it is worth flagging as a specification error to students, as it may indicate reverting to a procedural rather than an object-oriented style of coding.

It is an acknowledged hazard [14] of automated testing that it depends on formulation of appropriate tests, and this applies particularly to unit testing. However, we have found that structural specification may be made more complete with less effort.

These results also support our use of specification tests in both **Precheck** and **Check** modes. A common issue is due to substitutability of argument types, for example consider a student writing:

```
public short something(short x) { return x; }
```

when the method header should be

```
public int something(int x) { return x; }
```

The student's code could pass unit tests expecting an `int` to be returned by the method, but the code would fail a structural specification check.

We also noted some limitations with the quiz report data generated by the VLE, which we are discussing with the VLE developers.

This pilot has also helped to inform discussion of appropriate settings for CodeRunner questions on M250, with and without the use of the marking tools developed in this project.

3 Notes on potential for future development

This section provides notes on software development opportunities to be explored for this project.

3.1 Higher priority ideas

3.1.1 Intuitive summary of candidate solutions (for tutor-facing software)

Currently the information provided to tutors is generated by reflection and covers any classes determined to be required in the candidate solution.

A current example of information provided to tutors is as follows:

```
Fri Jan 19 10:23:03 GMT 2018
```

```
Methods defined in the solution classes
```

```
In class SneakyHoverFrog, public void setColour(OUColour)
In class SneakyHoverFrog, public void setIsSneaking(boolean)
In class SneakyHoverFrog, public boolean getIsSneaking()
In class SneakyHoverFrog, public void startSneaking()
In class SneakyHoverFrog, public void stopSneaking()
In class SneakyHoverFrog, public void panicIfCanBeSeen(Amphibian)
```

```
Fields defined in the solution classes
```

```
In class SneakyHoverFrog, private boolean isSneaking
```

```
Constructors defined in the solution classes
```

```
In class SneakyHoverFrog, public SneakyHoverFrog()
```

However, the java bytecode disassembler, `javap` [15] also allows us to determine information about a `.class` file's contents:

```
>jdk...\bin\javap -private SneakyHoverFrog.class
```

Note that the use of `-private` as a parameter to `javap` extends the information to `private` members of a class. This could be used to form the text in the 'information' panel in the CheckM250 output for tutors. For example it might read:

```
public class SneakyHoverFrog extends HoverFrog {
    private boolean isSneaking;
    public SneakyHoverFrog();
    public void setIsSneaking(boolean);
    public boolean getIsSneaking();
    public void setColour(ou.OUColour);
    public void startSneaking();
    public void stopSneaking();
    public void panicIfCanBeSeen(Amphibian);
}
```

This output reflects the order of declaration of members in the file. Note that it includes

- inheritance information;
- implements information (none in this case);
- fields;
- constructors;
- methods.

Using this approach, `javap` needs running for each class found in a candidate solution. However, it may be confusing to a marker to include classes that are provided as support files. Excluding reporting on support classes adds a layer to the problem: the need to exclude these classes within the code that invokes `javap`.

3.1.2 Alternative approach to specification of solutions

A specification such as `java.util.Map<Integer, java.lang.String>` becomes a raw `java.util.Map` at runtime because of Java's type erasure⁴ [16]. CheckM250 can currently only report on such structures with respect to their raw types.

The java bytecode disassembler (`javap`) output includes generic type parameters. This suggests an opportunity to use a disassembled representation of each required class in the solution as an alternative form of structural specification.

The current specification file `checks.txt` includes the names of the classes required in the candidate solution. If using `javap`, it would be necessary to determine the names of required classes a different way, perhaps through a file that lists the names of the required classes, or by placing examples of appropriate solution classes in a designated solutions folder.

The parsing of the specification would necessarily need to be different and may be more complex to achieve, since the current specification has been written to facilitate parsing.

Another tool worth considering in this context is TypeTools [17].

⁴ Raw types and type erasure are backward-compatibility fixes introduced in Java 1.6, when generic types were added to the language.

3.1.3 Specification of formal parameter names

Names of formal parameters are not currently checked by CheckM250. It may be worth exploring adding this functionality at a later date, since we may specify these in assignments (although it is not common).

Currently the feedback simply says that a method has or doesn't have the required *signature* (since signatures do not include parameter names).

In Java 6 and 7 the `-g` argument to the compiler allows generating bytecode information about parameter names. Investigation suggested that it might require a third-party library like `asm` [18] to retrieve information about parameter names in that context. Because development has initially been focussed on Java SE 7, this approach has not been explored further.

However, In Java 8 there is the compiler option `-parameters` and this results in more useful information being added to the bytecode. Example code from Oracle tutorials `MethodParameterSpy` class online [19] illustrates accessing information about parameters from bytecode. Figure 15 illustrates access to the name of a parameter `aRunner` in class `Runner.java` from its bytecode file (class `Runner`).

```
C:\OULocal\TEMPdelete\THA03_Project_Q4_Sol>"\Program Files\java\jdk1.8.0_131\bin
"\javac -cp ou.jar -parameters *.java
C:\OULocal\TEMPdelete\THA03_Project_Q4_Sol>"\Program Files\java\jdk1.8.0_131\bin
"\javac -cp ou.jar -parameters *.java
C:\OULocal\TEMPdelete\THA03_Project_Q4_Sol>"\Program Files\java\jdk1.8.0_131\bin
"\java MethodParameterSpy Runner
  Number of constructors: 1
public Runner()
  Number of parameters: 0
Number of declared constructors: 1
public Runner()
  Number of parameters: 0
  Number of methods: 9
public int Runner.compareTo(java.lang.Object)
  Return type: int
  Generic return type: int
  Parameter class: class java.lang.Object
  Parameter name: aRunner
  Modifiers: 4096
  Is implicit?: false
  Is name present?: true
  Is synthetic?: true
```

Figure 15: Example access of parameter name information from bytecode in Java 8

3.2 Lower priority enhancements

1. If the project in Bluej changes, the tutor-facing tool does not currently detect this; the tool must be relaunched. This problem does not arise in student-facing code. This is only a minor inconvenience on the tutor facing side.
2. The need to specify `void` for empty parameter lists in the specification file could possibly be removed. This is not a defect. This is moot if an alternative approach to specification is explored.
3. It may be possible to perform tests without writing the student's answer to disk, by loading and compiling the student's classes dynamically in memory instead. However, the current approach of creating a physical file is working. Not a defect.
4. CheckM250's reporting on zero-argument constructors could be improved to account for Java's provision of a default constructor when none is present in the student's code. The fall back is making sure that students understand about default constructors.

3.3 Also considered

The following items have also been considered for further code development, but it is suggested here that these should **not** be addressed.

1. If the student's code does not compile, the specification testing has to end. This is reasonable, since the code cannot be parsed in this case, and it would be difficult to extract parts of it that might be compiled for comparison with a specification. Human testing is the intended fall-back.
2. CheckM250 does not compile software automatically for tutors; they must press a compile button themselves. A future version of the tutor-facing software might perform this step automatically. There is no plan to implement this feature in CodeRunner for students. It is typical in BlueJ and CodeRunner to have to press a compile button anyway.
3. Extraneous public classes in a student's solution may suggest that it is fundamentally incorrect. However, the most likely reason for the presence of classes that appear to be extraneous is a student misspelling a required class name. Since CheckM250 will already report the absence of a required class, students are in a position to notice that their own class has a similar, but not the same name (e.g., it may be missing an initial capital letter). Adding a check for extraneous public classes might result in duplication of feedback, mentioning the correctly spelled name and the incorrectly spelled name.
4. CheckM250 does not allow an explicit check for the existence of specified user-defined Java interfaces. However, CheckM250 can check whether classes implement a user-defined interface and this will fail anyway if the interface is not provided. So there is another way to check the same requirement.
5. By design the software does not check for access modifiers on classes. M250 does not discuss access modifiers on classes, or inner classes at all, and so students will generally be unaware that there is a significance to this. If told that there is an access modifier missing on a class they are likely to be surprised rather than enlightened, so it does not seem important to implement this feature. (In CodeRunner there are currently additional issues with specifying classes as public, due to the need to save answers in a file, whose name ought to match a public class file name.)
6. Support for handling of unbound type parameters, which are an advanced language feature not covered in OU modules, could be added. However, if they were to occur they would appear as `Object` types at runtime, due to type erasure [16], and could presumably be specified that way.
7. Initial values of fields are not checked and this would be difficult to implement with any generality. It is unusual for TMA questions to specify initial values of fields apart from in constructors and for constants. If this is to be addressed, it should be limited to constants (static or non-static) initially, then we can consider whether constructor initialisation can be handled.

4 References

- [1] CS50: Introduction to Computer Science [Online] <https://online-learning.harvard.edu/course/cs50-introduction-computer-science> Accessed October 11th 2018
- [2] S. Király, K. Nehéz, and O. Hornyák, "Some aspects of grading Java code submissions in MOOCs," *Research in Learning Technology*, vol. 25, Jul. 2017.
- [3] "JUnit." [Online]. Available: <https://junit.org/junit5/>. Accessed: 25-May-2018.
- [4] "Checkstyle." [Online]. Available: <http://checkstyle.sourceforge.net/>. Accessed: 25-May-2018.
- [5] "PMD an extensible cross-language static code analyzer." [Online]. Available: <https://pmd.github.io/> Accessed: 25-May-2018.
- [6] "Google Java Style Guide" [Online] <https://google.github.io/styleguide/javaguide.html> Accessed 11-Sep-2018
- [7] M250 Code Conventions [Online] https://learn2.open.ac.uk/pluginfile.php/2414045/mod_resource/content/5/M250_Code_Conventions_ed2.pdf Accessed 11-Sep-2018
- [8] ANTLR [Online] <http://www.antlr.org/> Accessed 11-Sep-2018
- [9] Personal Communication with Michael Kölling, emails in Appendix 1, 18th April 2018
- [10] Lobb, R. and Harlow, J. 2016. CodeRunner. *ACM Inroads*. 7, 1 (2016), 47–51. DOI:<https://doi.org/10.1145/2810041>.
- [11] CodeRunner question types [Online] <http://CodeRunner.org.nz/mod/book/tool/print/index.php?id=184> Accessed October 10th 2018
- [12] Jobe Sandbox [Online] <http://CodeRunner.org.nz/mod/book/view.php?id=179&chapterid=655> Accessed September 11th 2018
- [13] Note on **Precheck** availability in CodeRunner v 3.1 Released <http://CodeRunner.org.nz/mod/forum/discuss.php?d=46> Accessed September 11th 2018
- [14] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming," *Journal on Educational Resources in Computing*, vol. 5, no. 3, pp. 4–20, 2005.
- [15] javap - The Java Class File Disassembler [Online] <https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html> Online accessed 2-Oct-2018 Accessed September 11th 2018
- [16] Type erasure [Online] <https://docs.oracle.com/javase/tutorial/java/generics/erasure.html> accessed 14th-Sep-2018
- [17] TypeTools [Online] <https://github.com/jhalterman/typetools> Accessed 14th-Sep-2018
- [18] ASM [Online] <http://asm.ow2.org/asm33/javadoc/user/org/objectweb/asm/ClassReader.html> Accessed October 11th 2018
- [19] Obtaining names of method parameters [Online] <https://docs.oracle.com/javase/tutorial/reflect/member/methodparameterreflection.html> Accessed 14th-Sep-2018

Appendix 1: Communication with Michael Kölling

Emails from 18th April, 2018.

From Anton Dil to Michael Kölling

Greetings Michael,

I hope you are well... Congratulations on your professorship!

I've written a small program that aims to reuse the javac.help file distributed with BlueJ, which I have somewhat modified and extended. My understanding is that this kind of reuse is acceptable under the GNU license, but I thought I would just ask you if there's anything I should be aware of around that. My intention is to use it behind the scenes in a code grader as the first step. The grader doesn't proceed if the code doesn't compile, but if there's a compilation error I thought I would offer some help. (I'm not sure students are in the habit of pressing the '?' in BlueJ I must say.) And sometimes there are so many reasons an error could come up I'm not sure it necessarily helps to list six reasons an error could arise. Anyway, that was the idea. I suppose the blackbox data also suggests some particular errors one should concentrate on explaining well.

If you have any comments on this, do please let me know. I suppose I would need to make the code available as open source outside of the system I want to use it with (Moodle). But the code itself is minimal so far – it's the modified javac.help that forms the bulk of it at this stage.

*Incidentally, I got the impression that this help was intended to be more specific than it seems to have wound up being, given the presence of * and {} markers that could presumably be filled in with some information, which I haven't seen done in the context of the '?' button, but maybe I missed that happening.*

Kind regards,

Anton

From Michael Kölling to Anton Dil

Dear Anton,

You are very welcome to use that file if it is useful to you. If the GNU license and re-publication is a problem or too cumbersome, I am also very happy to give you permission to use this file without these license restrictions.

We have not maintained this file for quite some time, and we are actually phasing out its use in BlueJ. As you say, not too many students click the question mark, and we started being less convinced of the value.

For the latest BlueJ version, we have rewritten the editor, with a substantially changed method of error handling and reporting (done continuously in the background now, with error display in popups close to the occurrence).

In other words: The file may well be out-of-date with current compiler versions, but you are very welcome to use it for your own software.

Kind regards,

Michael

King's College London