



DEVELOPMENT AND EVALUATION OF A TOOL FOR JAVA STRUCTURAL SPECIFICATION TESTING

Abstract

Although a number of tools for evaluating Java code functionality and style exist, little work has been done on automated marking of Java programs with respect to structural specifications. Structural checks support human markers in assessing students' work and evaluating their own marking; online automated marking; students checking code before submitting it for marking; and question setters evaluating the completeness of questions set. This project developed and evaluated a prototype tool that performs an automated check of a Java program's correctness with respect to a structural specification. Questionnaires and interviews were used to gather feedback on the usefulness of the tool as a marking aid to humans, and on its potential usefulness to students for self-assessment when working on their assignments. Markers were asked to compare the usefulness of structural specification testing as compared to other kinds of tool support, including syntax error assistance, style checking and functionality testing. Most markers using the structural specification checking tool found it to be useful, and some reported that it increased their accuracy in marking.

October 11th 2018

Anton Dil, Sue Truby and Joseph Osunde

Contact: Anton.Dil@open.ac.uk

1 Executive Summary

Students studying M250, our second year object-oriented programming module using Java, are required to complete Java programs according to detailed syntactical, structural, functional and stylistic specifications.

Although software tools exist for code syntax, functionality and style checking, tools for structural specification checking are not widely available. The long-term goal of this project is to raise awareness of these various aspects of correctness in our assessment of students' code and to support automated assessment of these aspects of code quality for tutors and students.

The project focussed particularly on the development and evaluation of a *structural* specification tool (known as **CheckM250**), deployed in the 2017J presentation of M250, to allow tutors to check to what extent students' code met a specification. The tool was provided for use in the module IDE, BlueJ, alongside traditional tutor marking notes. The project also explored the use of automated marking in the module Virtual Learning Environment (VLE), for quick feedback to students, and overcame technical obstacles in this context.

Tutor surveys and interviews were used to gather feedback on CheckM250 and on other kinds of marking tool support and traditional resources.

Automated structural checks on code were found to have multiple use cases:

- supporting human markers in assessing students' code;
- for markers to assess their own marking (as a kind of e-monitoring tool);
- in online assessment for automated marking of students' code;
- as a step in determining if software functional tests can proceed;
- for students to perform checks on their code before submitting it for marking;
- for question setters to check completeness of questions set for students.

There was evidence of tutors favouring the use of marking tools, or of their distrusting them, or finding them an obstacle. This appeared to depend less on the tool itself than on a predisposition for or against the use of tools. Similarly, tutors' comparative rating of tools as aids to themselves versus as aids to students appeared to depend on the tutors' disposition towards tools.

Most tutors using CheckM250 found it to be useful, and some reported that it increased their accuracy in marking. Tutors not using the tool cited lack of time and the simplicity of the assignment it was trialled on. Some reservations were expressed about reliance on automated marking tools, both for markers and for students. The marking software was also shown to be useful in the VLE for automated student feedback.

The results provided indicators of topics that should be discussed with tutors and students in this context:

- how automated code marking tools may best be used in tutor and student workflow;
- how the outputs of the tools should be interpreted;
- the potential benefits and pitfalls of automated marking;
- the relationships between the outputs of various automated marking tools.

The project has also suggested ways forward in developing automated marking tools for Java code.

2 Aims and Scope

2.1 Motivation

In a face to face environment tutor and peer support are available to help students interpret questions they have been set, while students working at a distance often rely on asynchronous forum and tutor support to complete assignments. In large modules, such as M250, there are issues of scale in supporting students individually, making automated marking more important. Automated marking has been shown elsewhere to be comparable to human marking and accepted by students. [1]. Although M250 uses a variety of computer-marked online activities we have so far had limited ability to provide automated feedback on the correctness and quality of students' code in the VLE.

For tutors, hand-marking code requires careful attention to written specifications. Detailed specification of programming assignments by the module team helps to constrain the range of responses that tutors should receive, which is important for scalability of marking.

Interpretation of code specifications often proves difficult for students due to the formal vocabulary used and their limited understanding of the structural features of languages [2].

The project aimed to involve M250 tutors in iteratively developing a software tool for Java structural specification checking, and to develop a range of automated advice to tutors as markers and students as learners, in both offline and online environments.

2.3 Scope of the project

Over the course of this study a number of layers of correctness in code have been identified, of which the most important are:

1. Syntax (Compilation)
2. Structure
3. Semantics (Functionality)
4. Style

In summary, we would like students' code to compile, but we also want it to conform to a structure providing certain classes, methods and other features, and for these structures to be identifiable because they have predictable names. Next we want to know that when the code is run it behaves according to some functional requirements. Finally we would like the solution to be of a good style. We might consider all of these elements taken together to form a specification of a solution and marks may be awarded to students for successful engagement with these layers of correctness.

2.4 Relationships between layers

By *structure*, we mean that a code solution provides various externally and internally visible features, rather than that it conforms 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 executing the student's code.

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

Structural correctness is 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. However, successful unit tests do not guarantee structural correctness.

Automated style checking can be supported through some well-known tools such as Checkstyle [3] and PMD [4], while functionality testing is well-supported using JUnit [5].

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

As syntax, semantics, and style are already relatively well-catered for by existing tools, this project has concentrated on *structural* features of code.

3 Activities

There were two main activities in this project:

1. Software development
2. Software evaluation

The software development activities had two strands:

- Tutor-facing software
- Student-facing (VLE) software

The software evaluation activities were concentrated on the tutor-facing tool, but also sought feedback from tutors on the use of marking tools in general, on the likely use of such software tools to students, and their place in the context of the module.

More information on the software and its use in the VLE is available in Appendix A. Details of the survey questions and interview results are in Appendix B.

3.1 Software Development

CheckM250 was developed using Java reflection facilities, to support marking in BlueJ and in the module VLE under the CodeRunner [6] environment.

The approach is similar to that discussed by Kiraly et al [7], but because the structural checks performed will be different for each assignment set, specification of expected features in a candidate solution was uncoupled from the code that checks for those features. We developed a simple specification language to describe the expected features of a correct solution to a programming assignment the module team set.

Classes are tested for:

1. extending expected superclasses;
2. implementing expected interfaces
3. providing methods with appropriate modifiers, return types, names, and formal argument types;
4. providing fields with appropriate modifiers, types and names;
5. providing specified constructors, with appropriate access modifiers and formal argument types.

We have also taken the view that correct code must not only provide expected features, it must omit 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.

We therefore check that classes do not contain any unexpected:

1. non-private constructors;
2. non-private methods;
3. fields (of any access level).

The motivation for these additional checks is that, for reasons of object-oriented style, we do not expect classes to provide more *accessible* constructors or methods than were specified in the question we set. Nor do we expect classes to include *any* fields we did not specify. Such features suggest departure from the specification, and may actually cause a defect in an otherwise working solution if the code were used in a real setting. We are more forgiving of `private` constructors and methods, as these are not accessible from outside a class, and so may reasonably be used for reasons of breaking down work within a class without fear of jeopardising its outwardly visible behaviour.

3.1.1 Tutor facing tool

A graphical interface was written for the CheckM250 tool together with a BlueJ plugin to launch the tool, so that tutors could select a student project to be marked, and its associated specification file.

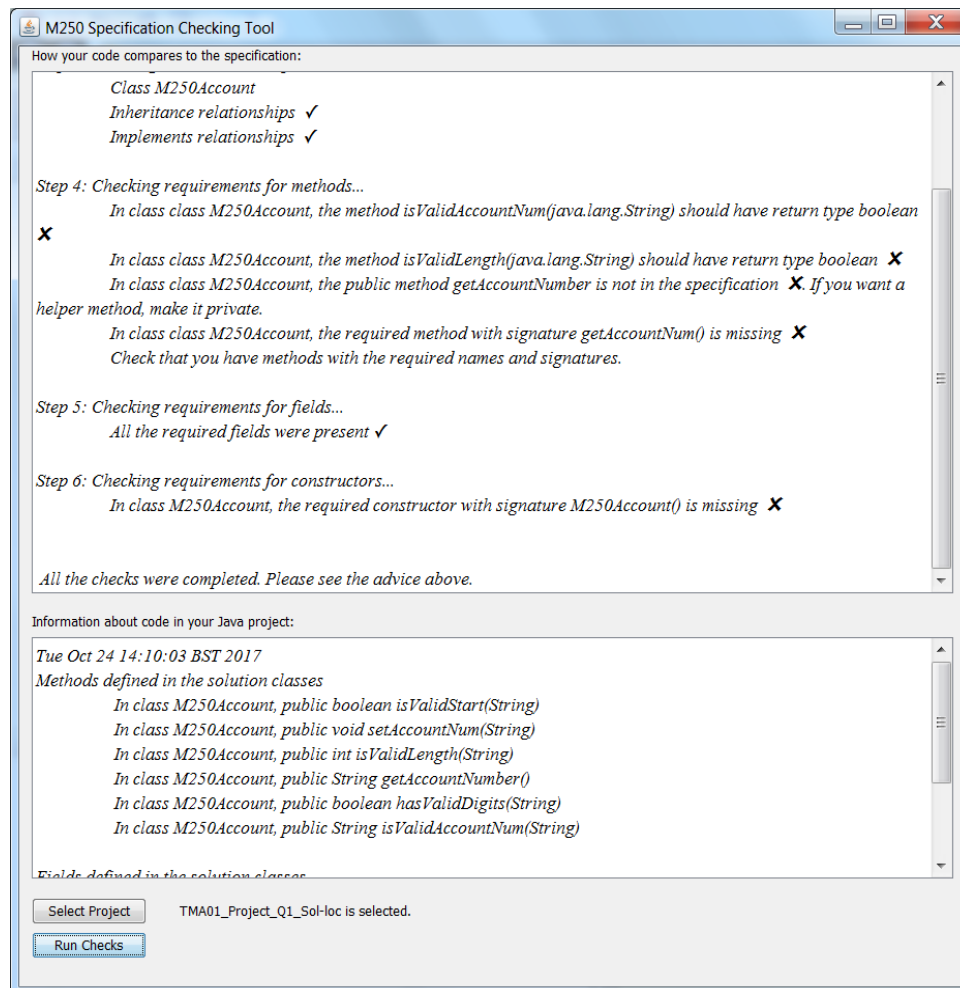


Figure 1: Example of the tutor-facing GUI with feedback on a student's project

Figure 1 illustrates the first version of the CheckM250 software, running in its own window, as

launched from the BlueJ environment. Tick marks are provided where structural specification requirements have been met, and crosses where they have been failed. The final version of CheckM250 provided to tutors incorporated some user-interface and functionality improvements.

3.1.2 Student facing tools

This project also explored running structural specification checking code in the module VLE under CodeRunner type questions, for automated feedback. The use of the debugging facility in CodeRunner was crucial to understanding the working of code in this environment.

Compilation Errors

It was necessary here (as in BlueJ) to take into account not only that students' code may not meet the specification we have set for them, but that it may not be possible to compile it at all.

```
Syntax Error(s)
__Tester__.java:29: error: cannot find symbol
    public integer getCurrentWeek()
           ^
    symbol:   class integer
    location: class Shelter
__Tester__.java:233: error: cannot find symbol
System.out.println(s.getAnimals().size()==1); // true
                   ^
    symbol:   method getAnimals()
    location: variable s of type Shelter
__Tester__.java:234: error: cannot find symbol
System.out.println(s.getAnimals().get(0).getName().equals("Biddy")); // true
                   ^
    symbol:   method getAnimals()
    location: variable s of type Shelter
__Tester__.java:235: error: cannot find symbol
System.out.println(s.getAnimals().get(0).getKind().equals("cat")); // true
                   ^
    symbol:   method getAnimals()
    location: variable s of type Shelter
__Tester__.java:236: error: cannot find symbol
System.out.println(s.getAnimals().get(0).getDescription().equals("Black and white")); //
                   ^
    symbol:   method getAnimals()
    location: variable s of type Shelter
__Tester__.java:237: error: cannot find symbol
System.out.println(s.getAnimals().get(0).getWeek()); // 1
                   ^
    symbol:   method getAnimals()
    location: variable s of type Shelter
6 errors
```

Figure 2: Example of standard compiler feedback on compilation errors, in the VLE

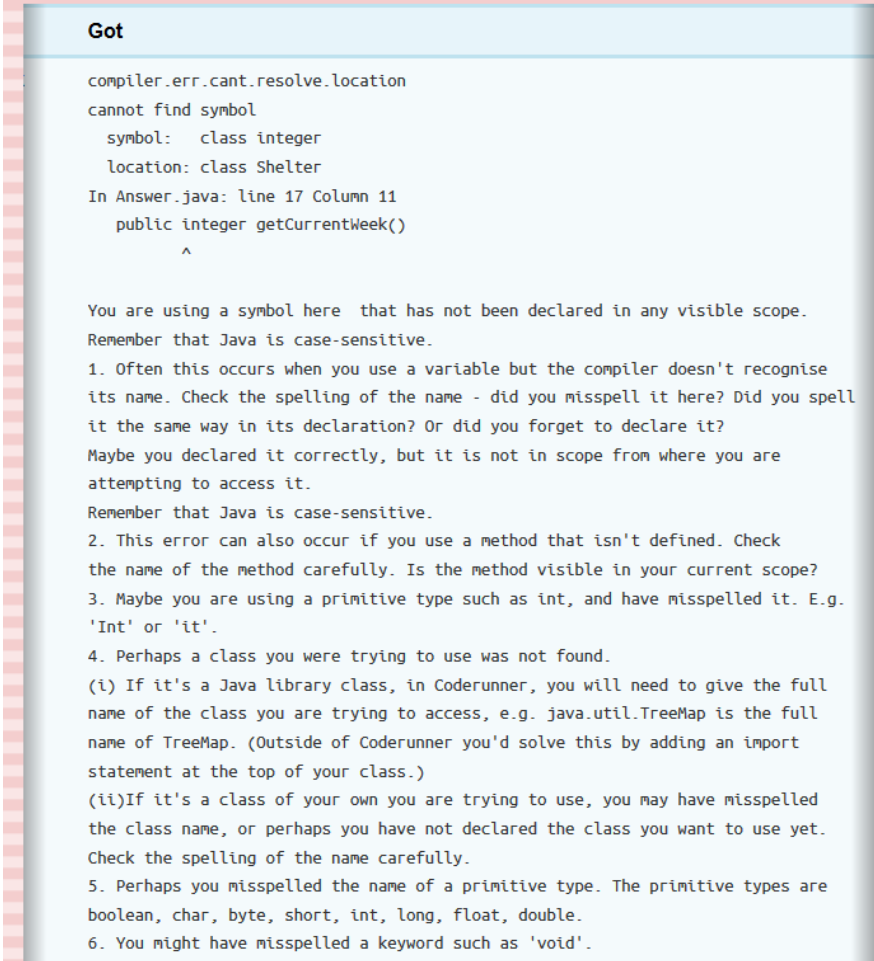
Figure 2 illustrates that the normal feedback for a case when compilation errors have occurred in a

student's solution code is simply to display the results returned by the compiler, without any interpretation. Although 6 errors are listed in Figure 2, there is actually just one syntax error in the student's code: 'cannot find symbol', relating to the use of the word 'integer' rather than 'int'. The remainder of the compilation errors actually relate to *structural* specification problems – methods that the student has not provided that are required in order to run the functionality testing code that CodeRunner is trying to run.

In modern IDEs such errors are displayed in-line with the code, however, in the context of the VLE and CodeRunner, where we have limited options for formatting feedback, it may be useful to attempt to summarise for students any problems the compiler has encountered.

To deal with this case, additional compilation error assistance software was developed. Permission to adapt the BlueJ Java 7 compilation error help file from BlueJ was obtained from the main author, Michael Kölling, to facilitate this. Advice on errors was added in a number of places where none was available, and in generating feedback an attempt was made to match specific errors before more generic ones.

Figure 3 provides an example of the enhanced feedback on a compilation error produced by this new tool.



The screenshot shows a light blue window titled "Got" with a red border. It displays a compiler error and a list of six numbered tips for troubleshooting. The error message is: "compiler.err.cant.resolve.location cannot find symbol symbol: class integer location: class Shelter In Answer.java: line 17 Column 11 public integer getCurrentWeek() ^". The tips are: 1. Check spelling and scope. 2. Check method name and visibility. 3. Check primitive type spelling. 4. Check class name and visibility. 5. Check primitive type spelling. 6. Check keyword spelling.

```
compiler.err.cant.resolve.location
cannot find symbol
  symbol: class integer
  location: class Shelter
In Answer.java: line 17 Column 11
  public integer getCurrentWeek()
         ^
```

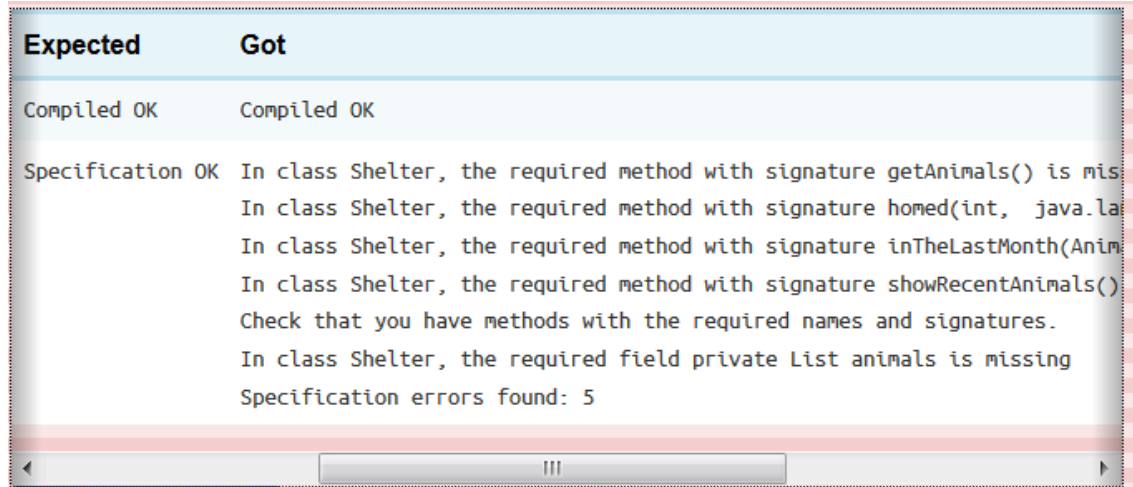
You are using a symbol here that has not been declared in any visible scope. Remember that Java is case-sensitive.

1. Often this occurs when you use a variable but the compiler doesn't recognise its name. Check the spelling of the name - did you misspell it here? Did you spell it the same way in its declaration? Or did you forget to declare it? Maybe you declared it correctly, but it is not in scope from where you are attempting to access it. Remember that Java is case-sensitive.
2. This error can also occur if you use a method that isn't defined. Check the name of the method carefully. Is the method visible in your current scope?
3. Maybe you are using a primitive type such as int, and have misspelled it. E.g. 'Int' or 'it'.
4. Perhaps a class you were trying to use was not found.
(i) If it's a Java library class, in Coderunner, you will need to give the full name of the class you are trying to access, e.g. java.util.TreeMap is the full name of TreeMap. (Outside of Coderunner you'd solve this by adding an import statement at the top of your class.)
(ii) If it's a class of your own you are trying to use, you may have misspelled the class name, or perhaps you have not declared the class you want to use yet. Check the spelling of the name carefully.
5. Perhaps you misspelled the name of a primitive type. The primitive types are boolean, char, byte, short, int, long, float, double.
6. You might have misspelled a keyword such as 'void'.

Figure 3: Example of enhanced feedback on a compilation error in CodeRunner

Note that only the first of the compilation error messages has been interpreted, to avoid overburdening students. The effort is concentrated on interpreting what the student might need to do to fix the first problem the compiler encountered.

Figure 4 shows the structural specification errors in the same solution (after successful compilation), this time as reported by the CheckM250 software running in CodeRunner.



Expected	Got
Compiled OK	Compiled OK
Specification OK	<pre>In class Shelter, the required method with signature getAnimals() is missing In class Shelter, the required method with signature homed(int, java.lang.String) is missing In class Shelter, the required method with signature inTheLastMonth(Animal) is missing In class Shelter, the required method with signature showRecentAnimals() is missing Check that you have methods with the required names and signatures. In class Shelter, the required field private List animals is missing Specification errors found: 5</pre>

Figure 4: Structural specification errors reported by CheckM250 in CodeRunner

The structural errors are now reported in a much more user-friendly way, indicating what features are missing from the student's solution.

3.2 Tutor and student involvement

A key aim of this project was to involve tutors in the development of a tool we hoped would be useful to them in marking, in evaluating it and suggesting ways in which it might be improved or used in future.

The intention was not to replace tutors with automated marking (although this is desirable in the VLE for quick feedback), but to provide a tool that could be used alongside traditional tutor notes.

3.2.1 Tutor engagement

Invitations to use CheckM250 were sent out to all tutors via M250 tutor forums and CAMEL messaging, but participation was necessarily optional. A dedicated forum on the M250 tutors website was provided [8], allowing quick interactions regarding usability issues and updates to the tutor-facing marking tool.

CheckM250 was initially offered for use on TMA01, with the expectation that the next iteration of the tool would be updated based on feedback on the initial prototype.

After a request for the tool to be available on TMA02, CheckM250 was updated to handle M250 library classes. A third version of the tool produced for TMA03 incorporated additional handling of generic types.

Survey

An anonymous survey containing a mixture of closed and open questions was created on Online Surveys [9] (formerly known as Bristol Online Surveys) and was available for one month from 12/01/2018, to coincide with marking the first TMA. The feedback received therefore relates to that

marking activity, rather than TMA02 and TMA03. (<https://openuniversity.onlinesurveys.ac.uk/code-marking-on-m250>)

There were 20 respondents out of 56 tutors on M250 in 2017J. Of the respondents, 8 had used the specification checking tool, so the majority of responses came from tutors who did not use the tool. Nine tutors are known to have used the tool. Some questions were directed only at tutors who had or hadn't used the tool.

A series of closed questions asked tutors general questions about marking on M250, and asked them to rate the usefulness of resources available or under investigation for use on the module for students. Tutors were also asked to rate potential usefulness of software tools that might be deployed in assessing code quality, both for tutor use and for student use, including the CheckM250 tool.

Follow-up open questions were used to gather more detailed feedback.

Interviews

Six experienced tutors were subsequently interviewed about their experience on M250 and in using CheckM250.

The interview results were transcribed and an inductive thematic analysis was performed to produce some insights into tutor experience in using the marking tool, reasons for not using it, and attitudes towards marking tools more generally and tutor concerns when marking students' code.

A summary of the tutor survey and interview analysis is contained in the following sections.

4 Findings

Tutors appeared to be weighing up the utility of the tool with respect to the time it takes to use it and the increased accuracy it may bring to their marking. For some tutors this trade-off was worthwhile; for others it was not.

The main findings regarding tool use are:

1. Tutors who did not use CheckM250 cited lack of time, simplicity of the assignment it was trialed on, and self-sufficiency.
2. Tutors appeared disposed either towards using a variety of marking aid tools, or to avoiding marking tools.
3. Tutors disposition towards students using the same tools appeared to depend on the tutor's disposition towards tool use.

Regarding the marking task:

4. Students' TMA code usually compiles. This is significant from the point of view of automated marking because compilation is required for automated marking. This suggests that the need for hand-marking could be the exception for some marking tasks.
5. Students may be opting not to submit code that does not compile, limiting their opportunities for feedback from tutors and opportunities for partial credit.
6. Tutors do not generally undertake to fix students' code when it doesn't compile. Only a quarter of respondents did this quite or very often.
7. Using the marking software may slow marking down, or potentially speed it up, depending on how the tool is used in the marking process and on familiarity with the tool.
8. Most tutors commented that using CheckM250 did not change their normal marking practice significantly, although the tool was used

- retrospectively to check that no structural specification errors had been missed;
 - before marking as a way to locate areas to comment on;
 - interactively, while marking assignments.
9. Most tutors using CheckM250 believed it was somewhat or very likely to pick up errors they would miss.
 10. The tool can be used to direct tutors to areas of students' work that require feedback.
 11. The majority of tutors who tested CheckM250 would use it again, and recommend it to other tutors.

The results of the survey and interviews suggested topics for discussion with tutors:

1. Ways in which marking software might be incorporated into tutors' workflow when marking;
2. the relationships between the outputs of automated tests, including dependencies that would allow or prevent automated tests from being run at all;
3. the relative importance of various aspects of correctness of students' code, including weighting of marks in assessment and the importance of providing feedback on any issues detected;
4. the likelihood of automated tests detecting errors that tutors might not notice;
5. the importance of retaining the human aspect of marking, to provide nuanced feedback, and to deal with areas of code quality that automated tools are less able to provide feedback on.

4.1 Tutor evaluation of tools

Tutors were asked about the utility of various teaching and marking resources, including CheckM250. Not all resources were available for tutors or students to access (some were proposals), so responses were based on tutors' professional judgement. Only suggested marking tools are included in Table 1 below. For more results, please see Appendix B.

A Likert scale was used to solicit ratings as 'not at all', 'slightly', 'moderately', 'very' or 'extremely' useful. Based on sums of proportions of ratings, tutors valued these resources as follows, ordering by very + extremely, then by moderately + very + extremely. Responses were received both from tool users (N=8) and non-tool users (N=12).

Table 1 Tool utility to tutors

Tool use to tutors	E	V+E	M+V+E
Unit tests	.26	.74	.74
Structural checking	.21	.47	.69
Style checking	.20	.45	.65
Syntax error help	.16	.37	.53

Table 2 Tool utility to students, judged by tutors

Tool use to students	E	V+E	M+V+E
Unit tests	.26	.58	.79
Style checking	.22	.39	.72
Syntax error help	.20	.40	.70
Structural checking	.17	.33	.72

The results indicated that whilst all of these tools were of some interest, they are not core to requirements; however, this is not unexpected given the traditional nature of our delivery being reliant on a printed text and offline software activities. Also, part of the aim is to develop these tools for online use by students in contexts where tutors are not available, to provide quick feedback.

The higher rating of unit (functionality) tests as compared to structural checking may indicate a misunderstanding on tutors' part of how unit testing would work in practice, because it is not possible in general to perform unit tests unless structural tests are successful. We attribute this anomaly to familiarity with the idea of unit testing as compared to the idea of structural checking, as well as to the preponderance of non-tool users in the respondents.

Considering the 'Extremely useful' column, these tools are considered of more or less equal utility to tutors and students, as judged by tutors. When combining the top two utility ratings, tutors considered all of these tools to be more useful to them than they are to students. This is expected for syntax error help (which tutors should not need). It is also worth noting that tutors' had not seen structural checking operating in the VLE for student use at this stage. Some evidence of its usefulness in this context is given in Appendix A.

Higher ratings for utility were given to our traditional resources, such as the module text and existing software activities, than to marking tools.

4.1.1 Correlations and associations

Consistently higher ratings of marking tools were awarded by the tool-using respondents than by those who did not trial the marking tool.

Ratings of unit testing, structural checking and style checking were all correlated at a statistically significant level (Spearman's rho, $p < 0.01$, two-tailed sigma). The exception is that syntax error help is not well correlated with other tools' utility. This is expected, since tutors should not particularly need this tool, which is more appropriate for students.

The highest correlation was for utility of unit testing versus structural checking, which was rho = 0.801 for student use. The 95% confidence interval for this result is 0.556 to 0.918, indicating a moderately high degree of correlation.

Tutor ratings of structural checking were also significantly associated between use for self and use for students (Somers' $d = 0.625$, $p < 0.001$). Thus, the relative utility of the tool itself may be less significant than the predisposition of the tutor towards tools. Higher association scores were found for Unit testing ($d = 0.687$, $p < 0.001$) and Style checking ($d = 0.706$, $p < 0.001$).

These results suggest that if a tutor rates one of these tools as useful, they will also rate the others as useful, and vice versa, whether for their own use or for student use.

Interview analysis provided some explanation for these dispositions for tutors' attitudes towards marking tools.

4.1.2 Tutor attitude to tools

Survey open responses and interviews helped identify the following themes emerging relating to the CheckM250 tool, and to tool use in general:

Time, Accuracy, Attitude towards tools, **Focus** of marking, and **Need** for a tool. A thematic analysis using these themes was then performed on the interview results.

These themes might be viewed as dichotomies as shown in Table 3.

Table 3: Tool use attitudinal coding themes as dichotomies

Theme	Negative	Positive
Time available to engage with the tool	No time to use, slows marking down; impacts on students' time	Worth investing the time
Quality of marking	No need for complete accuracy; could detract	Accuracy matters; improves marking and feedback
Attitude towards tools	Over-reliance on tools is an issue	Tools help us do our job better
Focus of teaching and testing	It's not all about structural specification - there are other things to provide feedback on	Correct structural specification (also) matters
Need for a tool	The task is too simple to warrant use of a tool	Even with simple tasks, we make mistakes tools can find

The 'need for a tool' theme was introduced because some tutors were responding that the tasks didn't warrant the tool use, suggesting that if the task had been more complex they would have been more open to tool use. This is therefore a separate issue to whether the tutor thinks tools are worth using at all, without reference to the complexity of the task.

The thematic analysis shows similar concerns to those expressed in Davis' technology acceptance model [10] in which users balance perceived usefulness with perceived ease of use, but it also raises pedagogical questions about tool use and questions about markers' judgement of their own abilities.

There are overlapping issues across these themes, explored below, so the categorisations that follow may be debated.

Time: Students versus Tutors

Tutors weighed up the utility of CheckM250, both with respect to the time it takes to use and the increased accuracy it may bring to their marking. There were also concerns that students' workload may be increased by using the tool and the extra information it provides. For some tutors increased time for increased accuracy in marking using the tool is worthwhile, while for others it is not.

Quality of marking

Tutors commented on how they felt about the tool detecting errors. Reactions ranged from embarrassment at having missed an error, to acceptance that this was what tools were good for, and recognition of having seen similar errors in colleagues' work when undertaking monitoring duties.

For some tutors it is important to detect all issues, because commenting on them should help students perform better in future. For others, it is more important for a student to get the 'big picture' correct.

It was suggested that receiving detailed negative feedback might be demoralising, but also that 'tick marks' from a tool might lead students to believe they are fully correct, when a tool only covers part of an aspect of code quality.

Tutors may worry that a tool is impersonal. There is no doubt that a human can generate more nuanced feedback than the tool is capable of.

The focus of teaching

Several tutors remarked that CheckM250 did not cover code style or other issues they would

comment on, and there was a fear of a tool directing attention towards certain aspects of code quality. However, CheckM250 is only intended to assist with one aspect of marking.

Some aspects of code quality may be less amenable to automation. For example, a style checking tool can pick up whether code is commented, but not whether the comments are sensible.

In some cases we felt that tutors were not making a clear distinction in their own minds between these aspects of quality and what kinds of tools could tackle them. The focus of this project was particularly on specification checking, but not to the exclusion of other kinds of automated code quality assessment or with the aim of removing the human marker.

The need for a tool

If the assignment is seen as simple to mark then the need for tool support is less clear. However, this also suggests that if the task were more complex, tutors would be more open to tool use. This is therefore a separate issue as to whether the marker thinks tools are worth using at all, without reference to the complexity of the task.

There is a concern that a marker could pay less close attention to student's code when given a tool like this to work with. Likewise there is a concern that students will come to rely on a tool, rather than use their own judgement.

In some cases, tutors feel that they will not fail to detect errors themselves, which we find more problematic. Our feeling is that not only will humans overlook issues that tools will not, but that a tool can help the tutor quickly identify places where feedback can be given.

5 Impact

5.1 Module team

Insights into the scope for automated marking support for tutors and for students in the VLE, based on the CodeRunner environment, have been gained.

Different use cases for structural specification checking tools by question setters, tutors, and students have been discovered.

Sometimes our provided TMA solutions do not precisely meet our own specifications, and we rely on markers to cope with small discrepancies, or we provide updated solutions when alerted to issues. In writing a more formal specification of an expected solution, question setters are afforded a way of checking the completeness of the question, and testing the conformance of their own solutions, before the question is provided to students and the solution is provided to markers.

The dependencies between and separate concerns of different aspects of code correctness have been clarified and this advice should be passed on to tutors for future reference, along with advice on incorporating marking tools in workflow.

5.1 Tutor marking

CheckM250 has been used by ten tutors on up to three TMAs each in the 2017J presentation, and we have received requests to continue supporting the tool in 2018J.

Tutor feedback suggested the most common errors the tool spotted that tutors had missed:

- misspelling of names of variables or methods, including in the case of the name;
- misspelling of types, e.g. in use of wrapper types versus primitive types;
- incorrect access modifiers (e.g. `public` instead of `private`);

- inappropriate use of the `static` modifier.

It is notable that all of these errors may not prevent code from functioning, and the code may pass functional (unit) tests, but it nevertheless does not meet specifications set by the question.

5.2 Student use

Two tools were deployed on two formative quizzes. Sue Truby produced 'Practice for TMA02', which included the Compilation Helper tool, while 'Practice for TMA03' included both the Compilation Helper and the Structural Specification Checking tools. The compilation error checking code was used 993 times (over both quizzes) while the structural marking code was used 479 times by students. These quizzes will be used again in 2018J, and the structural checking code will additionally be available in 'Practice for TMA02'.

CheckM250 was shown to have potential to assist students in making appropriate connections between their implementations of programs and the specification they have been required to work to. There were examples of forum postings quoting feedback from the marking tools, which facilitated students assisting each other.

Reuse of these tools could offer iterative feedback of progress towards meeting a specification, supporting self-help and self-evaluation, which are both important in distance learning.

Occasional student confusion between the layers of correctness outlined in this project is evident, suggesting a need for reinforced teaching in this area. Deploying marking tools of the kind developed in this project is arguably a good way to do this.

University impact

A number of technical issues were overcome in porting the automated marking code to the VLE, and we now have a better understanding of how to configure CodeRunner questions to incorporate automated assessment of some aspects of Java code quality.

Exploration of the iCMA statistics for CodeRunner will feed into discussions with the VLE development team, to try to improve the data made available to module teams, and allow for more detailed investigation of how students are using the facilities.

6 Conclusions and future work

A prototype structural specification checking tool for Java was developed and tested on M250. In addition to a BlueJ plugin tool, a version of the software was deployed on the module's VLE, where it was used extensively.

1. The software developed offers a way for students to check their understanding of specifications, to a large extent without the need to consult their tutor. Reuse of the tool offers iterative feedback of the student's progress towards completing code according to the required specification.
2. CheckM250 offers a way of quickly checking some aspects of a solution, and we hope that tutors may therefore give more attention to other aspects of code that are less easily tested automatically, for example the readability of the student's code.
3. Module teams have a tool they can use to check that provided solution code meets our own question specifications.
4. Tutors who used the software observed that it helped them find errors in students' work,

though it may have slowed them down somewhat. Some indicated that a changed workflow might lead to shorter working times. The tool could also act as a self-assessment of marking, depending on the workflow adopted.

5. We noted that structural specification checking should succeed for unit testing to take place and that it may detect errors that unit tests have not catered for.

Although there were some concerns expressed over the use of automated marking tools, we consider the project to have provided good evidence for the advantages of automated assessment of code quality in a variety of scenarios.

Future work

Although it has been simple to write structural specification files by hand, future work will explore approaches to automated generation of code specifications, as well as gather more quantitative data on the nature of errors in student code.

There is scope to explore how accurate tutors' marking is compared to the tool, though this would require much manual investigation.

Planned updates to the software aim to add generic type parameter checking, which is currently impossible due to Java's type erasure. Automated testing of Java code style for tutors and in the VLE will also be explored in future work. (More discussion of the software is available in Appendix A.)

7 Deliverables

1. 7th Esteem Annual Conference 2018 presentation (PPT)
2. Esteem Project Report and appendices (PDFs)
3. Online Survey Responses (PDF)
4. Anonymized Interview transcripts (Zipped Word documents)
5. Java source code (Zip archives) for
 - a. CheckM250 tool for BlueJ, with BlueJ plugin interface
 - b. CheckM250 tool for CodeRunner, with Compilation Helper tool
6. Dil, A., Osunde, J. 'Evaluation of a tool for Java Structural Specification Testing', 10th International Conference on Education Technology and Computers, Tokyo, October, 2018

8 List of Figures and Tables

Figure 1: Example of the tutor-facing GUI with feedback on a student's project

Figure 2: Example of standard compiler feedback on compilation errors in the VLE

Figure 3: Example of enhanced feedback on a compilation error in CodeRunner

Figure 4: Structural specification errors reported by CheckM250 in CodeRunner

Table 1 Tool utility to tutors

Table 2 Tool utility to students, judged by tutors

Table 3 Tool use attitudinal coding themes as dichotomies

9 References

- [1] Morris, D.S. 2003. Automatic grading of student's programming assignments: An interactive process and suite of programs. *Proceedings - Frontiers in Education Conference, FIE. 3*, (2003), S3F1–S3F6. DOI:<https://doi.org/10.1109/FIE.2003.1265998>.
- [2] Y. Qian and J. Lehman, "Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review," *ACM Trans. Comput. Educ.*, vol. 18, no. 1, pp. 1:1–1:24, 2017.
- [3] "Checkstyle." [Online]. Available: <http://checkstyle.sourceforge.net/>. [Accessed: 25-May-2018].
- [4] "PMD an extensible cross-language static code analyzer." [Online]. Available: <https://pmd.github.io/>. [Accessed: 25-May-2018].
- [5] "JUnit." [Online]. Available: <https://junit.org/junit5/>. [Accessed: 25-May-2018].
- [6] R. Lobb and J. Harlow, "CodeRunner," *ACM Inroads*, vol. 7, no. 1, pp. 47–51, 2016.
- [7] S. Király, K. Nehéz, and O. Hornyák, "Some aspects of grading Java code submissions in MOOCs," *Research in Learning Technology*, vol. 25, Jul. 2017.
- [8] Tutor forum on CheckM250 [Online]
<https://learn2.open.ac.uk/mod/forumng/view.php?id=1226234> Accessed 12th October, 2018
- [9] Online Surveys [Online] <https://www.onlinesurveys.ac.uk/> Accessed 12th October, 2018
- [10] F. D. Davis, "Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology," *MIS Q.*, vol. 13, no. 3, pp. 319–340, 1989.

10 List of Appendices

Appendix A – Software Notes and Student Use

Appendix B – Interview and Survey Results

11 Statement of Ethical Review

An ethical review was undertaken according to the University's code of practice and procedures before embarking on this project.

Staff Survey Project Panel (SSPP) Number 101