Evaluation of a tool for Java structural specification checking

Anton Dil
The Open University, United Kingdom
Walton Hall
MK7 6AA
+44 (0)1908 659225
a.dil@open.ac.uk

Joseph Osunde
The Open University, United Kingdom
Walton Hall
MK7 6AA
+44 (0)1908 652581
joseph.osunde@open.ac.uk

ABSTRACT

Although a number of tools for evaluating Java code functionality and style exist, little work has been done in a distance learning context on automated marking of Java programs with respect to structural specifications. Such automated 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 support, including syntax error assistance, style checking and functionality testing. Initial results suggest that most markers using the structural specification checking tool found it to be useful, and some reported that it increased their accuracy in marking. Reasons for not using the tool included lack of time and the simplicity of the assignment it was trialled on. Some reservations were expressed about reliance on tools for assessment, both for markers and for students. The need for advice on incorporating tools in marking workflow is suggested.

CCS Concepts

• Applied computing~Computer-assisted instruction • Software and its engineering~Software testing and debugging

Keywords

Open Distance Learning; Virtual Learning Environment; Automated Code Assessment; Software tools

1. INTRODUCTION

This paper reports on a pilot study of a marking tool on our institution's second year Java module, which uses a blend of online and offline teaching resources and is delivered to about 1400 primarily part-time and employed students per year via distance learning. Assignments are set by a small module team, who provide a written, indicatory marking guide to a separate group of about 50 tutors. Tutors provide written feedback to students on their submissions via an electronic submission system. Tutors are often also employed elsewhere as educators and have their own marking styles, which we moderate via monitoring procedures. We believe that the use of static and dynamic software testing tools is important for both markers and students to gain familiarity with industry-standard approaches.

Authors such as Hattie [1] see assessment as an opportunity to provide feedback and, in this initiative, we are also exploring opportunities for feedback via automated marking of formative assessments in our virtual learning environment (VLE), Moodle. It is widely acknowledged that in teaching large cohorts, as in Massive Open Online Courses (MOOCs), automation can be particularly useful, for quick feedback, self-assessment and greater availability [2].

For automated assessment, the mode of specification has much to do with the quality of the solution [3]. However, interpretation of code specifications often proves difficult for beginners due to the formal vocabulary required and understanding of the syntactic and structural features of languages [4,5]. Natural language specifications are ambiguous and this is a major cause of incorrect implementations [6].

Regular expressions have commonly been used to check conformance to functionality requirements [7] while more general frameworks for testing have evolved from Hoare-Floyd logic [8], in which assertions about a program's state are made. Specification may be expressed externally to a program, or may be formally incorporated in language design, for example, in Design By Contract as expressed in the Eiffel language [9]. There are also language-independent specification languages, which may be bound to different implementations to test correctness, for example interface description languages are used to ensure compatibility of components in distributed software systems, which may be implemented in multiple languages.

Formal specifications, which provide mathematical robustness, have been used in safety critical systems [10].

However, in the context of an introductory programming module, we are not keen to add to our students' workload by teaching additional terminologies and techniques for program specification.

Király et al. [11] describes the implementation of specification testing on a MOOC platform, incorporating structural, style and functionality tests via a cloud service. This has disadvantages of cloud service cost, potential issues with availability of the service and hard-coding of specifications.

CourseMarker is a platform supporting submission and marking of assignments that includes a variety of similar correctness checks [12]. The authors have shown that the automated marking is on a par with human marking and accepted by students.

Insa and Silva [13] have developed a library and workbench for automated assessment called ASys, including structural testing, based on verifying properties of students' code. The software uses a graphical interface to allow a user to drive the creation of a test-harness class.

Our aims are very similar to those of the aforementioned platforms. However, although tools exist to assist with many of these assessments, e.g. functionality testing through JUnit in Java [14] and style checking through tools such as Checkstyle [15] and

PMD [16], structural checking is less commonly available, particularly in Java, and so is the focus of this study. Our tool is therefore similar to that described in [11] but we have initially focused on structural testing. Many other aspects of correctness may be checked by assessment software, including efficiency and simplicity [7, 12, 13], but these will be addressed in future work.

2. STRUCTURAL SPECIFICATIONS

By structure we mean that a solution provides various externally and internally visible features, such as methods or fields, rather than that it conforms to a particular behaviour or functionality.

To ensure that students understand the vocabulary of the Java language, some assignments we set prescribe these structural features of solutions. This helps to ensure that markers will receive relatively constrained responses, which assists comparability of work and scalability of marking. This paper particularly concerns marking prescriptive assignments of this kind, although we also set open-ended questions to allow students to explore their own approaches to solving problems.

Whilst Balzer at al. [17] state that 'a specification is a description of what is desired, rather than how it is to be realized', testing structural correctness allows us to check for correct use of language features, which is important pedagogically. We nevertheless follow the principle that specifications should be operational, i.e. formal enough to enable automated testing of whether a proposed implementation meets the specification.

As in [13] our tool makes use of a specification file, but in our case this does not require generation of a separate Java test-class. Our structural specifications can be hand-edited quite simply or may potentially be generated by other means. Our structural tests are not intended to be exhaustive; rather they cover features of code that we would normally expect tutors to comment on in students' solutions.

Another difference in our work is that we test not only for the presence of required features, but for the absence of some features on grounds of object-oriented programming style, even though they would not affect the functionality of students' code. This is implicit, rather than specified on an individual assessment basis.

For our purposes, structural, functional and stylistic correctness all initially require that a student's code compiles successfully, as this facilitates automation.

Structural correctness is also required for unit testing code to compile, so a key purpose of our tool is to determine if unit tests could succeed, which is important for fully automated online testing. Conversely, code that appears to behave appropriately under unit tests does not necessarily meet structural requirements, although sufficiently careful unit testing might reveal this.

2.1 Attitudes towards marking tool support

A further aim of this work is to explore attitudes towards tools as assistants to markers, and for fully automated marking. There is evidence of broad acceptance in the context of automated marking, for example [2] and [12] report on the benefits of the objectivity of automated feedback and regular assessment of progress for students, however issues such as the specificity of feedback that should be provided and the appropriate focus or weighting of marking are less clear.

Whereas adoption of automated marking assumes accuracy of the tools is high, our tool-supported marking process affords us the chance of exploring how tools might be usefully incorporated into a more human-focused marking process. Other authors have assumed that question setters would determine the weight assigned to various aspects of correctness, but in our context, there has traditionally been a degree of latitude expected in marks awarded and marking guides are indicatory, so we wished to explore markers' attitudes towards different aspects of correctness and how they should be weighted. Our tools' output for markers is advisory and accompanies our written marking guide.

3. METHODOLOGY

Our project aims to involve tutors on our Java programming module through a collaborative process in the design of code marking tools, and to gather feedback on attitudes towards tools for marking support, as well as for student use, via online forum interactions, VLE usage data, surveys and interviews. We cannot require tutors or students to take part in evaluation of prototype tools, therefore our participants are volunteers.

In assessing attitudes towards tools, we have initially concentrated on four aspects of code correctness: Syntax, Functionality and Style, as the most common aspects included in other similar tools [12], and Structure, which is less commonly included.

A BlueJ plugin [18] was developed to integrate the structural checking tool into the module coding environment, for tutor use. The tutor receives detailed output describing features of a student's solution and where it departs from the expected specification. Occasionally, tutors have to fall back on manual inspection of the students' code, due to non-compilation of a solution, which is to be expected.

Tutor use of the BlueJ plugin was supported via online forums, and the tool was updated several times during the module presentation following early feedback on usability issues. These initial informal discussions informed the design of an online, anonymous survey, which was open to all tutors, whether they had used the marking tool or not. In addition to providing a forum for tutors who chose not to participate in the trial of the marking tool, the online survey allowed us to pose some generic questions about how tutors mark code, about marking tools, and the relative utility of various resources we provide and might provide in future.

Tutors were asked to rate the utility of syntax error help, structural testing, functionality testing and style checking tools, to themselves, and to students. Use of the marking tool was also investigated, via 20 questions of which 13 were closed response and 7 were open response, providing an opportunity for tutors to clarify reasons for their answers to closed questions as well as provide more general feedback on marking and tools.

Example output from the structural checking code, repurposed for deployment on our VLE via CodeRunner [19] questions, is shown in Figure 1.

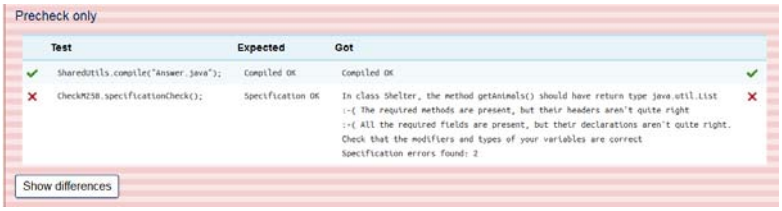

Figure 1 CodeRunner output for a failed structural check.

Twenty responses from tutors were received, including eight from tutors who had used the structural checking tool and twelve from tutors who had not.

Due to tutor availability, subsequently six tutors out of the eight who had used the structural checking tool were interviewed using a semi-structured script, to get a deeper understanding of attitudes towards the tool and marking tools in general. Tutors were asked about their experience in using the tool, issues they encountered, and how it affected their marking. An inductive analysis of the interview transcripts was then performed to draw out the commonly occurring themes. Tutor attitudes towards student use of the structural marking tool and related tools was also explored.

Although the tool was run by about a third of our cohort in our online environment, via a formative CodeRunner quiz, this paper is concerned only with the tutors' appraisals of the proposed marking tools.

4. RESULTS

4.1 How the marking tool was used

An early result of this investigation was that the need for the question setter to write a structural specification served as a cross-check on the completeness of the question we were setting. This was unexpected, but ultimately very useful.

Depending on when the tool was used in the marker's workflow, it functioned either as a backup check that no structural errors in the student's solution had been missed, or as a way to locate parts of student's code to be commented on before marking by hand.

Tutors realised that the tool would not be very helpful if the submitted code was very far off the mark to begin with, however, it was acknowledged that this is where we need the human marker to take the lead.

Students were said to seldom submit code that does not compile, suggesting that they may see compilation errors as a sign of failure, and they prefer not to submit incomplete work, though they could gain marks by doing so.

Averaged time to mark for tool users was 5-10% more than for non-tool users, but there was not enough data to infer statistical significance of this difference.

Six of eight tutors said that the tool was 'somewhat likely' or 'very likely' to spot errors they'd have missed.

Seven of eight tutors said that they would be 'quite likely' or 'very likely' to recommend the tool to other tutors. For these tutors, improved accuracy was valued, while increased time to mark was the main concern of the one tutor 'not likely' to recommend the tool.

4.2 Ratings of tools by tutors

Table 1 shows aggregated ratings of software tools by the 20 tutors surveyed (i) to them as markers and (ii) their judgement of the tool's usefulness to students, based on a Likert scale. Only the structural checking tool was actually tested by a proportion tutors, so the responses are primarily based on tutors' perception of the tools' potential usefulness.

Table 1 Tool utility to tutors and students, judged by tutors

(i) 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
(ii) Tool use to students			
Unit tests	.26	.58	.79
Style checking	.22	.39	.72
Syntax error help	.20	.40	.70
Structural checking	.17	.33	.72

The table entries are ordered by Extremely, Very + Extremely and Moderately + Very + Extremely responses. (Other available responses were: 'Not at all useful', 'Somewhat useful' and 'Moderately useful'.)

None of the code marking tools were rated 'extremely' useful by the majority of respondents, indicating that whilst they are all 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. Part of our aim is to develop these tools for online use by students in contexts where tutors are not available, for quick feedback.

The higher rating of unit 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 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.

4.2.1 Correlations and associations

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 correlated 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 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 or against tools, and this is explored in the next section.

4.3 Themes in the interview data

Six experienced tutors who had used the structural checking tool were interviewed. The inductive analysis of interview transcripts led to the themes in Table 2, with both positive and negative views expressed under each theme.

Table 2 Themes around tool use by tutors

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	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

There are overlapping issues across these themes, which are now described:

Time: Tutors weighed up the utility of a 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. Some of the time cost may be accounted for by familiarity or the way in which the tool was incorporated in the marking workflow.

In the case of students, some tutors worried that using a tool would add to their workload.

Accuracy: Some tutors suggested that the tool would not spot errors they wouldn't, or that the difference the tool might have made was small. This relates to the complexity of the marking task also.

Some appreciated confirmation of their own accuracy. Indication of having missed an error was embarrassing to some, while for others it proved that this was what tools were good for. This was in spite of tutors having spotted similar lapses in other tutors' marking when they were undertaking peer monitoring duties.

For some it is important to provide feedback on all defects that are noticed, because commenting on them should help students perform better in future. However, some feared discouraging students by pointing out all of their failings.

Attitude towards tools: Some tutors believed that they did not need the tool, as they would spot errors anyway, or that over-reliance on technology was problematic, because viewing automated feedback would lead to loss of critical awareness or a bias in marking focus.

Likewise, tutors were concerned that students may come to rely on a tool like this if they used it regularly, an issue also reported by Chen [20].

Focus of teaching and testing: The structural checking tool highlights a particular aspect of code quality, but this may be misleading if other areas of code quality are not indicated. There is a fear that a tool might direct attention too much towards certain areas and away from others, including aspects that are less amenable to tool support.

Of course, the intention is not to exclude hand-marking by humans, but to support marking by humans of a particular aspect of code quality that is amenable to automation. The intention is to provide support for marking other aspects of code quality that may be automated, eventually, and tutors also supported this idea.

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.

Our thematic analysis shows similar concerns to those expressed in Davis' technology acceptance model [21] 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.

4.4 Common errors found by the tool

We found that the structural checking tool was able to detect errors that would prevent unit tests from succeeding and to find errors that markers should comment on for pedagogical reasons. Misspelled variable and method names, use of wrapper types for primitive types, incorrect access modifiers and misuse of the `static` modifier were all cited as errors that markers had not noticed when reading over students' code.

Some detected errors may result in code that passes unit tests. An example we saw was failure to override an `equals` method. In this case, our own unit tests had failed to check for overloading rather than overriding and all the functionality tests were passed.

Likewise, use of the `static` modifier could easily be missed by a unit test checking for functionality of a method, and may indicate students reverting to a procedural rather than an object-oriented style of coding.

This is an acknowledged hazard [12] of automated testing – it relies 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.

5. CONCLUSIONS

A prototype structural specification checking tool was developed and tested on a distance learning Java programming module with a large cohort and 50 markers. In addition to a BlueJ plugin tool, a version of the tool was deployed on the module's VLE, where it was used extensively.

Markers who used the tool observed that it helped them find errors in students' work, even if it slowed them down somewhat, though some indicated that a changed workflow might actually lead to shorter working times when using the tool.

Tutors who did not use the tool cited lack of time or the need for the tool on a simple assignment, and tended to indicate that they would not miss errors in students' code; however, we found several kinds of errors commonly missed by markers.

Tutors who expressed positive views of the tool also tended to consider it would be useful to students, whilst those who thought it of less value also considered it of less value to students. Tutors tended to favour use of various tools, or none.

Some tutors reported that the tool acted as a self-assessment of their marking, depending on the workflow adopted.

For student use, tutors expressed concerns over workload and direction of attention towards particular correctness concerns.

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.

Discussing tool use has resulted in some preliminary guidelines for use of tools in supporting human markers: Marking tools should not impact too much on tutors' time to mark, but advice on workflow may help to mitigate this issue. We also need to clarify which aspects of correctness concern us, and how they should be weighted, as tutors expressed different points of focus in their own marking. By providing data on commonly missed errors in students' code, we can raise awareness of where tools can out-perform human markers, even in the context of simple assignments. Some aspects of code quality may be best suited to human feedback, so it will be important to clarify which aspects of code we want our markers to focus on and which to delegate to tool support. Finally, it is important that we emphasize that markers (and students) should not use marking tools as a substitute for their own appraisal of a solution's correctness.

Future work will explore approaches to automated generation of specifications, as well as gather more quantitative data on how often tutors miss errors in code, and what kinds of mistakes students most commonly make.

6. ACKNOWLEDGMENTS

The authors acknowledge the support of the Open University's eSTEeM centre for STEM pedagogy.

7. REFERENCES

[1] J. Hattie, *Visible Learning for Teachers*. Routledge, 2012.

[2] C. MacNish. "Java Facilities for Automating Analysis, Feedback and Assessment of Laboratory Work." *Computer Science Education*. 10, 2 (2000), 147–163. DOI:https://doi.org/10.1076/0899-3408(200008)10:2;1-C;FT147.

[3] R. S. Pressman, *Software Engineering: A practitioner's approach*, 3rd ed. McGraw-Hill, 1992.

[4] 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.

[5] Robins, A. et al. 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*.

[6] M. Bano, "Addressing the challenges of requirements ambiguity: A review of empirical literature," *5th International Workshop on Empirical Requirements Engineering, EmpiRE 2015 - Proceedings*. pp. 21–24.

[7] V. Pieterse, 2013. Automated Assessment of Programming Assignments. *3rd Computer Science Education Research Conference on Computer Science Education Research*. 3, April (2013), 45–56. DOI:https://doi.org/http://dx.doi.org/10.1145/1559755.1559763.

[8] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[9] "Eiffel Software." [Online]. Available: https://www.eiffel.com/values/design-by-contract/. [Accessed: 25-May-2018].

[10] B. Potter, J. Sinclair, and D. Till, "An introduction to formal specification and Z, Prentice Hall", *1996*, 2nd ed. Prentice-Hall, 1996.

[11] S. Király, K. Nehéz, and O. Hornyák, "Some aspects of grading Java code submissions in MOOCs," *Research in Learning Technology*, vol. 25, Jul. 2017.

[12] 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.

[13] D. Insa and J. Silva, "Semi-Automatic Assessment of Unrestrained Java Code: A Library, a DSL, and a Workbench to Assess Exams and Exercises," *Proc. 2015 ACM Conf. Innov. Technol. Comput. Sci. Educ. - ITiCSE '15*, no. January 2015, pp. 39–44, 2015.

[14] "Junit." [Online]. Available: https://junit.org/junit5/. [Accessed: 25-May-2018].

[15] "Checkstyle." [Online]. Available: http://checkstyle.sourceforge.net/. [Accessed: 25-May-2018].

[16] "PMD an extensible cross-language static code analyzer." [Online]. Available: https://pmd.github.io/. [Accessed: 25-May-2018].

[17] R. and N. G. Balzer, "Principles of Good Specification and Their Implications for Specification Languages," *Software Specification Techniques*, pp. pp. 25–39., 1986.

[18] "Bluej Extensions." [Online]. Available: https://bluej.org/extensions/extensions.html. [Accessed: 25-May-2018].

[19] R. Lobb and J. Harlow, "Coderunner," *ACM Inroads*, vol. 7, no. 1, pp. 47–51, 2016.

[20] P. M. Chen, "An automated feedback system for computer organization projects," *IEEE Trans. Educ.*, vol. 47, no. 2, pp. 232–240, 2004.

[21] 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.