Considerations in Automated Marking

Joel Fenwick

The University of Queensland,
Centre for Geoscience Computing
QLD 4072
Australia
joelfenwick@uq.edu.au

Abstract

With large classes and high demands on the time of teaching academics, (as well as the need to keep marking budgets under control) evaluating the functional correctness of programming assignments can be challenging. Entirely automating the evaluation process may seem desirable but that would deny students formative feedback from more experienced programmers. This in turn reduces their opportunity to correct errors in their practice.

Instead, this paper contains a discussion of marking processes where much of the "heavy lifting" or repetitive work is automated but still allows for human feedback. We discuss the impact of automated marking on assessment design, students, and where the hard work is hidden.

The literature contains descriptions of many projects for automating various parts of the process with varying interfaces and levels of integration with external systems. In the author's opinion though, that they are not strictly *required*, and we describe a simpler set of requirements.

Keywords: Programming Assessment, Automated Marking, Assessment Design

1 Introduction

Programming assessment submissions can be evaluated in a number of ways. They can be judged on how well they have implemented specified functionality (either by direct testing or by inspection); how readable and well structured their source code is; their algorithmic complexity or runtime performance; or their design and the process used to produce them (typically in more advanced courses). In introductory and intermediate courses, the focus tends to be on the first two. Marking large numbers of such assignments requires significant amounts of time to do well and risks uneven treatment as markers tire.

While there is still a need for human judgement when it comes to evaluating things like readability, repetitive testing of functionality seems to be a suitable target for automation. Section 2 looks briefly at the history and development of automation of programming marking. However, whether existing automation packages are adopted or ad-hoc tools are employed, not every task which humans can mark is suitable for simple automation. In this discussion, we focus on black box testing of programs to be submitted and evaluated against some "well defined" specification (as opposed to more open ended "do something cool" type assignments). We will also assume that the functionality of the program (rather than the precise algorithm used to produce it) is the main point of interest.

Section 3 gives more detail about the course which provides the main context for this work. Section 4 gives core questions to be asked when evaluating the functionality of a programming assignment submission. Section 5 gives an example of simple automation work flow in terms of basic primitives. While arbitrarily complex ad-hoc solutions are possible, we limit the discussion to what can be achieved with simple tools and small amounts of custom coding.

Sections 6 and 7 discuss the impacts of this style of automation on students and on the design and description of programming assessment. Section 8 looks at *some* security/integrity considerations. Section 9 concludes with a summary of where the work hides when this type of automation is employed.

2 Some history

Very early work was done by Hollingsworth (1960), who described automated marking for a class of 80 students. A few years later, Forsythe & Wirth (1965), followed suit. There are quite a few common features between these works: The technical details of getting the tests to run are a significant issue. In order for these early graders to work, the student programs must have a particular structure (a trait mirrored in later "unit test" style testers). The authors of both systems acknowledge the possibility of student code interfering with the testing infrastructure but do not consider it to be a major issue. Both systems require manual intervention in the case of badly behaved programs. Douce et al. (2005) would later classify these types of systems as first generation systems.

Deimel & Clarkson (1978) discuss the merits of running student assignment submissions against unseen test data. Gathering student submissions was still an issue at this point. They also state a challenge which is still with us today: that students' real goal 'irrespective of the problem statement, is to produce "correct" output for the supplied input'.

Later, Benson (1985) described a system where students would use email to submit files. These would then be processed against a batch of tests with the results being available to students the next day. This was done before the deadline so that students had a chance to fix errors. These tests were made available "several days before the due date." More detailed tests were used to determine marks once the deadline

passed. An interesting approach adopted by Benson was that students could appeal their marks if they could demonstrate that proper testing on their part could not have detected the fault.

Harris et al. (2004) took the pre-testing approach even further with a system where assignments could not be submitted at all unless they passed a set of supplied tests. This requirement was enforced by the submission tool itself. This way only fully functional assignments are considered for further grading.

In this document, we will refer to automated tests accessible before the assignment deadline as *public* tests. Whereas tests used for marking will be denoted *hidden* tests. In the interests of transparency, this second set of tests should also be revealed eventually. As an aside, it is also possible not to provide sample tests, but instead to make the production of tests by students, part of the assessment (Edwards 2003).

The survey by Douce et al. (2005) divides automated testing systems into three generations. First generation systems which relied on technical tricks to operate. Second generation systems used tools already available from the operating system. Third generation systems made use of the web and included a wider variety of testing approaches.

One example (from many) of these third generation type systems is BOSS (Joy et al. 2005). It provides both a web interface and network client application to allow students to lodge code to be tested. This allowed both student testing against a public test set with the possibility of re-submission and testing against the hidden set for staff assessing final submissions. Of particular note here, is the remoteness of the testing. The tool is accessed from a client machine (eg student laptop) but the tests are executed on a server.

Ihantola et al. (2010) carried out a follow up survey covering the years 2006–2010. They identify classifications of testing: using a framework such as JUnit; comparing the output of running programs; scripting the build, test and comparison; and experimental approaches. They concluded "too many new systems are developed" but that a reason this occurred was that tools were complete enough to meet the needs of the course they were created for but not necessarily general enough to be applied elsewhere[1].

2.1 Isn't this a solved problem?

After literature spread over 50 years, isn't automation of programming assignment marking a solved problem by now? Not really, no. The developments behind the generations described in the 2005 survey are not monotonic improvements towards a fixed goal. Moving from the first generation to the second, the ability to construct and run tests at all became less of a problem because there was now greater support from the operating system. Writing comparison based tests became simpler. Similarly, there are less troublesome ways to gather assignment submissions than collecting punch cards or individual emails. Some systems incorporate the submission mechanism (eg BOSS).

The main characteristics of interest in the third generation are greater levels of integration with other systems, and other interfaces to the testing system (typically web interfaces). Systems in this generation aim for reusability between assessments and applicability to a variety of languages. There is still work to be done here though. Tests must still be designed for each assessment (even if just in the form of *input:output* pairs). Also language flexibility typically means one of three options:

- The system already has some support for the chosen language.
- Output matching is being done at a level where the language is irrelevant. For example: capturing text output at the OS level; examining file contents after a run is completed or exchanging messages across a network.
- The system provides an interface to write plugins or subclasses for the chosen language (eg GAME (Blumenstein et al. 2008)).

In their survey, Ihantola et al. (2010) distinguished between automation for marking programming competitions versus "systems for (introductory) programming education." While competition marking is an interesting area, formative feedback does not seem to be a consideration there. On the education side, the parenthesis around "introductory" are important here. More advanced courses have additional requirements or make use of lower level features which are not needed in introductory courses. For example, Solomon et al. (2006) describe the LinuxGym tool for assessing and training students in the use of shell and scripting. They draw a distinction between LinuxGym and BOSS (Joy et al. 2005) due to the fact that their tasks require modification of system state rather than producing output.

2.2 What about xUnit?

A number of automated testing systems (eg BOSS) can make use of libraries from the "xUnit" family. These include PyUnit for Python and JUnit for Java and are derived from a Smalltalk testing library written by Kent Beck (Fowler 2014, Python developers 2014*b*, JUnit project 2014). Individual tests are written as methods of classes which inherit from a class in the xUnit library. These methods can throw exceptions to indicate that a test has failed. After a batch of tests has run, a report can be presented indicating which tests passed and which failed.

In the author's experience, with well written tests, xUnit is an effective means to test an API. In assignments however, this would indicate a library is being written or where the internals of the code have been specified. For example, the assignment is written as: you must write a class X which has

- a method `int thing(int x, int y, int z)` which returns the median of its arguments.
- a method `String meth(String a, String b)` which ...

In more advanced assessments, it may not be desirable to specify implementation details at this granularity. Students could be expected to make their own design decisions rather than be constrained by tests of internals. In these cases, tests using reflection might not naturally fit with specified functionality. They also do not test the external interface. An additional test interface would need to be specified. Now, it is possible to use the xUnit structure to describe tests against external programs (the test functions can contain arbitrary statements in the relevant language), but there does not seem to be any special advantage in doing so.

[1] They also note that experimental systems tend to disappear from the web.

3 Context

The driver for this work is a course with the dual purposes of teaching systems programming concepts and improving programming skill. The previous run of the course had over 300 students and the current offering has over 400. The assessment consists primarily of traditional programming assignments. Marking has two components: functionality ($\sim 85\%$) and style ($\sim 15\%$). The functionality mark is based solely on whether the program produces the correct results and system interactions. It must not only say the right things but also not leave processes running or consume unacceptable amounts of system resources. But, apart from a criterion of not "taking too long" to run, the performance of the algorithms used is not a concern. This part of the marking is done entirely with automated black box testing using simple `bash` scripts. While a single staff member needs to check on the process occasionally, performing this part of the marking is fairly undemanding.

The style component requires attention from human markers who grade submissions on clarity, structure and adherence to a supplied style guide. However, the relatively small fraction of the overall marks means that fine gradations in readability and structure are not required and markers don't need to spend a lot of time doing it. This process ensures that students still receive feedback about how humans read their code.

While the course doesn't go as far as Harris et al. (2004) in rejecting submissions which aren't functionally perfect; submissions which don't pass at least some functionality tests are not marked.

The assignment tasks typically consist of sequences of interactions or commands for users. For example, various card or grid based games; agents interacting with a simulation environment; or system automation. This requires student programs to both be able to recognise valid interactions and to maintain state.

As discussed later, this lends itself to rubrics where marks for more complex tasks depend on successful completion of earlier subtasks. For example: "make a single valid move" leads to "play a complete game".

Assignment submission is done by committing code to a version control repository (`subversion` in this case). This neatly handles re-submission and time stamping as well as exposing students to professionally useful tools. Gathering submissions for marking only requires: a list of students, two version control commands[2] and a `bash` *for* loop.

In terms of testing, students are given access to public tests soon after the release of assignment specifications. These can be tested using two supplied commands, the first checks the student's current version. The second checks out the student's most recent commit and tests that. This acts as a check that the students are committing correctly (and haven't forgotten to add files) and that they committed what they thought they did.

4 Questions

Three main questions to be considered when determining a functionality mark:

1. To what extent does the code give the correct answer/results in response to valid input?
2. To what extent does the code handle bad input or bad system states gracefully?
3. What does the code do while processing? / How does the code arrive at that answer?

4.1 Question 1 — How does the code behave under good conditions?

All that is required here is a means to provide prepared "good" inputs; a means to capture and examine the output and actions of the system; and a set of matched inputs and outputs to indicate the correct response.

4.2 Question 2 — How does the code behave under bad conditions?

At its most basic, this just means ensuring that your test collection checks that error messages are properly triggered. Depending on the level of the course, there may be other things which should be tested. For example, empty lines (or empty input entirely) should not cause programs to loop or terminate ungracefully. There are also failures in the environment to be considered. Checking how a program responds to an instruction to read from a non-existent file is relatively easy; forcing failure to create a file because the directory is readonly requires a little more work; inducing a system call failure due to "out of resources" requires more work.

4.3 Question 3 — What does the code do while processing?

In some assessments, there may be other considerations beyond whether the code produced the correct answer. This may include whether the code:

- used forbidden calls.
- has the correct asymptotic complexity.
- has acceptable run time.
- is "safe" (eg in terms of concurrency).
- "leaks" resources.

If the assessment required that students use particular approaches, it may be possible for students to use alternate approaches or libraries which dodge the point of the assessment or avoid the work. For example reading and writing from/to disk files instead of pipes or calling built in sort functions instead of writing their own. Simple text searches for particular strings will catch some abuses but are not guaranteed to stop the truly determined (especially if the language in use is amenable to obfuscation). However, since human markers would still be looking at the code, use of obfuscation techniques would hopefully be noticed. It is up to the individual assessor how much time should be devoted to searching for this type of abuse.

Determining asymptotic complexity by sampling could be attempted programatically, provided that worst case instances are known, but is beyond the scope of this work. More general timing runs could also be used as an assessment criteria but could much more simply be used as a proxy for "must not loop indefinitely". In that case, stopping programs which "take too long" is sufficient (and a necessary self defence measure as well).

Testing safe operation under concurrency (where this is a reasonable expectation of students) would

[2] `svn checkout` for source code and `svn log` for time stamp information.

be difficult to achieve without special tools, but a rough test may be possible by testing with number of clients/requests/actions simultaneously. It will not guarantee that the code is thread safe but it may catch some instances which aren't.

Detecting resource usage generally, may be tricky or require extra tools. However, for the specific case of memory leaks, `valgind` could be employed on a number of platforms (Valgrind developers 2014).

5 Simple Automation

For this discussion, we are assuming that the following capabilities are available in some form:

1. A means to extract student submissions and compile them (where required).
2. A means to execute submitted programs programatically and to specify inputs fed to those programs.
3. A means to capture output from the executing programs.
4. A means to compare text or the contents of files with other files.
5. A means to report the results of the above.
6. A means to gather the above into a command or batch.

For example, the first item depends on the submission system, but the rest of the above can be done relatively easily with simple shell scripting or using Python's subprocess module (Python developers 2014*a*). Additional primitives which may be "nice to have" but not required include:

7. A mechanism to compare prefixes of files (eg the first 200 bytes) rather than whole files.
8. A means to automatically terminate programs which run for more than a specified number of seconds.

In more advanced settings where a number of programs may need to run simultaneously, Item 2 may require that this task doesn't block. It will also be desirable to ensure that all the started programs terminate at the conclusion of the test.

Work by Isaacson & Scott (1989) gives some examples of simple shell operations which may be useful here and also an example script. However, that script may be more detailed than is required for simple testing and would need to be customised.

With those primitives in place, the workflow for an assignment will look something like:

Pre-submission

After coming up with a concept and an initial specification:

1. *Write a working "reference" implementation of the assignment.* This allows problem areas or tasks that are harder than intended, to be identified *before* they stress the students unnecessarily. It also means that a sample solution will be available later without relying on the student body to produce one.
2. *Refine the specification (and implementation) to fix problems as they are discovered.*
3. *Create the private test set.*
4. *Create the public test set.* The "expected" outputs for both sets should be generated from the reference implementation to ensure consistency.
5. *Release assignment specification and public tests.*
6. *Update the specification and public tests.* This will be necessary if ambiguities or errors are discovered in either. To avoid problems discussed later in Section 6, it is a good idea to state that the specification trumps public tests (but that students should report contradictions so they can be fixed). If changes are made, it is important to correct the reference implementation and the private tests at the same time.

Marking

1. *Gather assignment submissions*
   The details will depend on the submission system, but a collection of subdirectories (one per student is ideal).
2. *Filter pass*
   Search for forbidden calls or commands. This is an opportunity to check the assignments for anything really nasty before compilation. See Section 8 for possible considerations.
3. *Compile submissions*
   This is quicker where the compiler has a command line interface[3]. If a build management tool such as `make` or `scons` is available then having the students submit the relevant files[4] may be helpful. Submissions which do not compile can be removed from consideration or repaired (depending on the rules of the course) at this stage.
4. *Run tests for each student*
   It will probably be necessary to monitor this process in order to restart it if one of the programs hangs[5] or kills the tester (in the case of systems programming assignments). In the case of trouble, testing can be resumed with the next submission. The problematic submission can be separated out for more cautious testing.

   In the author's experience, only a small fraction of assignment submissions ever cause problems which require manual intervention.
5. *Collate test results.*
   This is significantly easier if the per-student script/batch outputs something like a comma separated list of results (and an id) which can be concatenated and loaded into a spreadsheet for easy viewing.

   As well as being necessary for determining a grade, this can serve as a sanity check for tests. If very few submissions pass a given test, it should be reviewed to ensure the "expected answer" is correct.
6. *Rerun tests if required.*

[3] Some suites such as Visual Studio have a command line interfaces as well, but they are not always immediately obvious.

[4] Or for simple projects with known files and structure, copying a standard build file into the directory.

[5] This will only be a problem if you don't have timeouts in place.

7. *Make results and tests available to students*
   It is important to note that this does not mean that all mark components must be released at the same time. The results of automated testing can be released well before the human marked components are finished. This means that students can have an idea about how they performed quickly.

   The trick here is to find a way to make the information available in a human readable way. Adding forty columns (one for each test) per assignment to a coursework management system's marks return feature does not produce particularly readable results. An alternative would be to just make the private tests available at this point. This is not the same as the students knowing exactly what the marker recorded though. In the author's course, a simple additional program makes this fine grained information available to the students.

8. *Complete remainder of marking*
   That is, the non-automated parts.

6 Impacts/Challenges — Students

Employing this type of approach can have an impact on students. In the author's experience, three ways students can be affected (positively or negatively) are:

- There can be a collision between a strict application of a specification, and the expectation among (some) students that specifications are merely "advisory".
- Students are exposed to methodical testing and the idea of test driven development.
- Students can work "to the tests" rather than the specification.

6.1 "Advisory" versus strict specifications

Some students seem to take the view that results which vaguely match the specification are sufficient. If the students are accustomed to vague rubrics, encountering something requiring strict compliance can be a shock. While looking approximately correct may fool human markers, who have strictly limited reserves of time and alertness; the same can not be said for machine checking.

On the other hand, it may be that human markers decide to take a flexible view of matching. It is tricky though, to describe programatically the wide variety of answers which a human would consider "close enough" (eg using regular expressions). Doesn't this indicate a weakness with automated marking in that it lacks the required flexibility? Not necessarily. In many cases it is easier to specify that something should be "exactly this" instead of "something like this". It may also be that following the requirements exactly, takes no more coding effort than following them approximately.

Trying to help students by allowing greater flexibility can be counter-productive, since it often leads to students wanting a formal specification of precisely what variance is permitted and what is not. However, one way to allow some flexibility without very complex specification is to define acceptable behaviour in terms of the behaviour of standard functions and tools. For example, "if `scanf` can get the correct integer from it, then it is valid input".

Another way the warped view of the importance of following specifications manifests is in students substituting their own measures of partial success. Deimel & Pozefsky (1979) argued that "programs have to do more than just work" but the *work* aspect seems to have been deprecated. Now, students adopt measures like "hours spent" or "having written lots of code" as substitutes for doing what the specification says. This situation is certainly not unique to situations of automated marking, but it definitely occurs here.

6.2 Methodical testing

Automated marking emphasises the importance of testing for students because they are told that their marks depend on being able to produce exact matches. Some students may not have seen how effective disciplined testing can be in identifying flaws and regressions.

To allay concerns about strictness of testing, batches of public tests can be provided to students prior to the assessment deadlines. If the test mechanism is exposed to students as well, then there are additional benefits.

- The sufficiently keen students can create their own test batches and share them with fellow students.
- It is easier for students to reproduce the circumstances of a failing test in order to debug their code.
- Formative feedback and transparency: After marking, students can reproduce the marking process in order to check their marks or understand where they went wrong.
- The test mechanism can be used as an example program (especially if programs which interact with other programs are discussed in the course).

6.3 Tests versus specification

In some instances, students misuse the public tests by replacing the goal of writing a program which complies with the specification with a "simpler" goal of writing a program which passes the tests. Isaacson & Scott (1989) note that this can discourage students from considering for themselves what test inputs would be appropriate to confirm the correctness of the program.

Aside from thwarting the educational purpose of the assessment, this reliance solely on public tests is flawed on two counts: First, it can result in trying to debug a program without understanding what it is supposed to do and why. This in turn increases the risk of regressions. Second, the public tests and the hidden tests are different. Code which produces correct responses to one set without properly implementing the underlying functionality has no guarantee of doing well against a different set. Even when these facts are made known to students, the wrong emphasis seems hard to shift.

7 Impacts/Challenges — Assessment Design

As well as having impact on students, applying automation has impacts on assessment design as well.

The first consideration if automation is to used is whether the assigned task is amenable to black box testing at all. In work done in the context of programming competitions (but applicable here), Forišek

(2006) describes come features which make a task unsuitable.

- The set of possible correct answers is (relatively) large.
- Only a small amount of output is required — and that small amount of output is statistically likely to be correct.
- There is a simple but incorrect heuristic for the problem.

For example, in combinatorial problems, a program could pick an answer at random and have a non-trivial chance of getting marks. Where programs need to produce more output (which must all be coherent), this will likely be less of a problem. The last point however, has wider applicability since it is roughly equivalent to avoiding the work as described in 4.3.

Assuming that the task admits automated marking, the following factors need to be considered when describing the task and choosing what to assign marks to:

1. Precision
2. Visibility
3. Isolation
4. Determinism
5. Recognition of partial success

7.1 Precision

If something is not described sufficiently precisely and unambiguously, then it can't be tested effectively. For example, consider a program where the communication protocol between client and server for networking or IPC assessments is left for students to design; while the interface to the client (and possibly the server) is specified. This may well be desirable in more advanced assessments where students are expected to be able to do such things. However, it does mean that the network/IPC can't be tested in isolation and that both components are required to function in order for marks to be awarded. Depending on the difficulty of the task this may or may not be acceptable.

If components are to be tested separately, then a reference implementation or test rig simulating the corresponding component will be required.

7.2 Visibility

In contrast to a human marking code by inspection, with automated testing, if an event is not visible then it can't be assigned marks. Intermediate steps needed to produce results may not naturally produce output. For example, opening a network connection or successfully reading data from a file. Depending on the complexity of the overall task, it may be desirable to allocate some marks to these steps.

Actions which interact with the system (kernel) state may be visible with the right tools[6] but they seem to be either unreliable for short lived events or not simple to employ[7]. A rough test could be to produce output when each stage of the process is performed successfully. Merely outputting "Success" does not mean it actually happened though, so the code would also need to be tested to see if it accurately reports failures. Practically, it seems fairer to explicitly test for failures and leave successes to be assessed in later steps.

In the case of intermediate results, there may be ways to expose them but that exposure must also be specified. This may enlarge/complicate the assessment specification further. The presence of extra internal state information may distort the output of the program with clutter. It may also give students an unrealistic view of programming practice.

7.3 Isolation

If an event or result can't be isolated from other output, then it can't be marked. If a result is indicated by the presence of an easily extracted string in the output, then this may not present much of a challenge. However, if the test is equivalent to "is the first part of the output is correct?", it is a bit more fiddly. Two solutions here are either:

1. compare only prefixes of the output and ignore any differences after a certain threshold.
2. Ensure that the program/system can be stopped as soon as possible after the event of interest.

The first option is not difficult to code but we want to minimise any custom coding required so let's consider the second one. A simple way to have natural stopping points is to have programs which process distinct operations and prompt between them. Then specify what should happen at end of input / disconnection. This means that the most basic unit of simple testing is "empty input"[8]. More sophisticated tests can then be built up from that starting point: one interaction then stop; two interactions then stop, ...

The importance of handling end of input properly should be emphasised to students, but if public tests are available, then failures in this aspect will be readily apparent. There is a side benefit here in that handling end of input properly is not something which students seem to consider when left to their own devices.

7.4 Determinism

To keep testing simple and transparent, the behaviour of (correctly written) programs being tested should be deterministic. This does limit the use of things like random numbers unless a pseudo-random number generator is specified and the seed can be easily specified. Rather than do this, a simpler option is to read streams of values from files rather than the randomiser. For example, instead of shuffling cards, specify a file which contains a pre-ordered deck.

Where a number of processes or threads are involved, accidents of scheduling can lead to race conditions. Two cases to consider here:

1. The ordering/interleaving of output varies, but the decisions made by the programs are the same. If success can be determined by the presence of particular strings in the output, then a search could be made just for those strings. If the output from different parties can be distinguished somehow (eg relevant lines have a known prefix), then the relevant lines can be filtered. Alternatively, simply sorting the lines of text before comparison deals with the ordering issue quite neatly[9].

[6] Possibilities include: polling with system tools or library interposing.

[7] This is not to say that they lack merit, merely that they fall outside the scope of "simple automation".

[8] After argument checking to allow the program to start at all.

[9] Assuming that any ordering is equally valid

2. The programs make different decisions under different event orderings. This occurs in situations like networking assignments where a number of clients need to start and connect to a server. For example, the clients represent players in a game and which "seat" the player occupies is significant. Introducing pauses after starting the server and between each client start would seem to be a solution here. However,

   - determining the time to wait can be tricky. Too long a delay and the tests will take a long time to run, frustrate users and slow down the testing unnecessarily. If the delay is too short, the tests may react erratically on a heavily loaded machine.
   - forcing the testing to operate in serial, removes the need for students to write thread-safe code.

   An alternative solution is to make the ordering depend on something predictable. For example, requiring each player to give their name and then seating players in lexicographic order, gives predictable results[10] but does not force connections to be spaced out.

7.5 Recognition of partial success

Testing for matching output does not leave much room for "part marks". Either the program passes the test or it doesn't. This can mean that a program which has 90% of a task perfectly correct could still record a "fail" for that task. Alternatively, a program could behave correctly under most but not all correct inputs. To mitigate this risk:

- A number of tests should be employed to mark against each subtask to help recognise programs which are capable of completing that task under some conditions.
- The tasks for the assignment should be examined in the light of precision, visibility and isolation to see whether they *should* be subdivided. This may require interface changes, such as extra output.

The subdivision of tasks needs to be appropriate for the level of the course. A balance needs to be struck here between rewarding partial progress versus the need for programmers to produce working code.

8 Security and Integrity Considerations

The preceding discussion assumes that the marker runs student code or build instructions in a context other than the student's account. This will always be risky to some degree: Programs could be submitted which attempt to gain access to system privileges or to course information; or to disrupt the marking process itself. Alternatively, programs may simply be badly written and abuse system resources.

Possible approaches to *mitigate* risks could be some combination of:

- *Identifying problem programs before running them.* This will never be possible in the completely general case but simple checks can be made for calls to external programs. Depending on the programming language, any uses of inline assembly language or unusual compiler directives are candidates for further examination.
- *Preventing programs from being able to do undesirable things.* Approaches such as library interposing or automated code substitution to replace "potentially dangerous" calls with more restricted ones. The author has employed this in the past to protect against fork bombs. In the general case, this would be harder because all of the valid ways a call could be used would need to be accounted for.
- *Isolating "bad" code so there is nothing for it to attack.* This could include running code in a separate/limited account but could extend to running in a separate environment. For example, a virtual machine or a chroot/jail. Steps in this category may require help from systems administrators.

Care needs to be taken however that the environment used for marking does not differ significantly from the one accessible to students. If the test environment is protected in ways that cause it to behave significantly differently to the students' environment under normal conditions, then questions of fairness must be considered. On the other hand, if the development environment is "too safe", students will not learn how to recognise and debug problems "in the wild". With this in mind, where possible, protection measures should be optional in that they can be applied (or not) without affecting the main results. See (Ihantola et al. 2010) for other possibilities.

9 Conclusion — Where is the work hiding?

Now that we have these primitives and workflow for assignments which will be marked programatically, we now summarise the important question of "where is the work hiding?". After all, the fact that simple tests can be administered programatically, does not make the process trivial.

Much of the work in dealing with this type of assignment is front-loaded. The specification, reference implementation[11] and public tests are all needed before the assignment is released. Decisions about precisely how programs will be evaluated can't be deferred until a time after the assignments have been submitted. This work requires a greater amount of the teacher's time, with reduced amount of tutor/teaching assistant time. The net budgetary affect of this shift needs to be considered.

Significant detail is required in the specification. It must describe precisely how the program is to behave and what the output and side effects are to be. Examples of interactions will probably be needed. All this means that while specifications may not actually be more complicated, they may be large. In the author's case for a second year course, this results in specifications of (roughly) between five and eight pages once common boiler-plate text is removed.

In conclusion, while simple automation (however it is accomplished) does move some work earlier in the course, it can significantly reduce the marking burden. Further, because of the reduced academic effort involved when dealing with submissions, scales quite well.

References

Benson, M. (1985), 'Machine assisted marking of programming assignments', *SIGCSE Bull.* **17**(3), 24–

[10] Yes, this assumes one agrees not to test with duplicate identifiers.

[11] While it is possible to start with a partially complete implementation, doing so creates problems later.

25.
URL: *http://doi.acm.org/10.1145/382208.382516*

Blumenstein, M., Green, S., Fogelman, S., Nguyen, A. & Muthukkumarasamy, V. (2008), 'Performance analysis of game: A generic automated marking environment', *Computers and Education* **50**(4), 1203–1216.

Deimel, Jr., L. E. & Clarkson, B. A. (1978), The todisk-watload system: A convenient tool for evaluating student programs, *in* 'Proceedings of the 16th Annual Southeast Regional Conference', ACM-SE 16, ACM, New York, NY, USA, pp. 168–171.
URL: *http://doi.acm.org/10.1145/503643.503681*

Deimel, Jr., L. E. & Pozefsky, M. (1979), 'Requirements for student programs in the undergraduate computer science curriculum: How much is enough?', *SIGCSE Bull.* **11**(1), 14–17.
URL: *http://doi.acm.org/10.1145/953030.809543*

Douce, C., Livingstone, D. & Orwell, J. (2005), 'Automatic test-based assessment of programming: A review', *J. Educ. Resour. Comput.* **5**(3).
URL: *http://doi.acm.org/10.1145/1163405.1163409*

Edwards, S. H. (2003), Teaching software testing: Automatic grading meets test-first coding, *in* 'Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications', OOPSLA '03, ACM, New York, NY, USA, pp. 318–319.
URL: *http://doi.acm.org/10.1145/949344.949431*

Forišek, M. (2006), 'On the suitability of programming tasks for automated evaluation', *Informatics in Education* **5**(1), 63–73. Copyright - Copyright Institute of Mathematics and Informatics 2006; Document feature - ; Last updated - 2011-06-03.

Forsythe, G. E. & Wirth, N. (1965), 'Automatic grading programs', *Commun. ACM* **8**(5), 275–278.
URL: *http://doi.acm.org/10.1145/364914.364937*

Fowler, M. (2014), 'Xunit'.
URL: *http://www.martinfowler.com/bliki/Xunit.html*

Harris, J. A., Adams, E. S. & Harris, N. L. (2004), 'Making program grading easier: But not totally automatic', *J. Comput. Sci. Coll.* **20**(1), 248–261.
URL: *http://dl.acm.org/citation.cfm?id=1040231.1040264*

Hollingsworth, J. (1960), 'Automatic graders for programming classes', *Commun. ACM* **3**(10), 528–529.
URL: *http://doi.acm.org/10.1145/367415.367422*

Ihantola, P., Ahoniemi, T., Karavirta, V. & Seppälä, O. (2010), Review of recent systems for automatic assessment of programming assignments, *in* 'Proceedings of the 10th Koli Calling International Conference on Computing Education Research', Koli Calling '10, ACM, New York, NY, USA, pp. 86–93.
URL: *http://doi.acm.org/10.1145/1930464.1930480*

Isaacson, P. C. & Scott, T. A. (1989), 'Automating the execution of student programs', *SIGCSE Bull.* **21**(2), 15–22.
URL: *http://doi.acm.org/10.1145/65738.65741*

Joy, M., Griffiths, N. & Boyatt, R. (2005), 'The boss online submission and assessment system', *J. Educ. Resour. Comput.* **5**(3).
URL: *http://doi.acm.org/10.1145/1163405.1163407*

JUnit project (2014), 'JUnit FAQ'.
URL: *https://github.com/junit-team/junit/wiki/FAQ*

Python developers (2014*a*), '17.1. subprocess Subprocess management Python v2.7.7 documentation'.
URL: *https://docs.python.org/2/library/subprocess.html*

Python developers (2014*b*), '25.3. unittest — Unit testing framework — Python v2.7.7 documentation'.
URL: *https://docs.python.org/2/library/unittest.html*

Solomon, A., Santamaria, D. & Lister, R. (2006), Automated testing of unix command-line and scripting skills, *in* 'Information Technology Based Higher Education and Training, 2006. ITHET '06. 7th International Conference on', pp. 120–125.

Valgrind developers (2014), 'Valgrind: Supported platforms'.
URL: *http://valgrind.org/info/platforms.html*