Trends in Digital Education:
Selected papers from EC-TEL 2015 Workshops CHANGEE, WAPLA, and HybridEd

MAGIC: Massive Automated Grading in the Cloud

Armando Fox[1], David Patterson[1], Samuel Joseph[2], and Paul McCulloch[3]

[1]University of California, Berkeley
[2]Hawai'i Pacific University, MakersAcademy, and AgileVentures
[3]Apex Computing Inc. and AgileVentures

Abstract. We describe our experience developing and using a specific category of cloud-based autograder (automatic evaluator of student programming assignments) for software engineering. To establish our position in the landscape, our autograder is *fully automatic* rather than assisting the instructor in performing manual grading, and test based, in that it exercises student code under controlled conditions rather than relying on static analysis or comparing only the output of student programs against reference output. We include a brief description of the course for which the autograders were built, *Engineering Software as a Service*, and the rationale for building them in the first place, since we had to surmount some new obstacles related to the scale and delivery mechanism of the course. In three years of using the autograders in conjunction with both a software engineering MOOC and the residential course on which the MOOC is based, they have reliably graded hundreds of thousands of student assignments, and are currently being refactored to make their code more easily extensible and maintainable. We have found cloud-based autograding to be scalable, sandboxable, and reliable, and students value the near-instant feedback and opportunities to resubmit homework assignments more than once. Our autograder architecture and implementation are open source, cloud-based, LMS-agnostic, and easily extensible with new types of grading engines. Our goal is not to make specific research claims on behalf of our system, but to extract from our experience engineering lessons for others interested in building or adapting similar systems.

Keywords: automatic grading, programming, software engineering, on-line education.

1 Background: Autograding for a Software Engineering Course

Automated assessment of student programming assignments was first tried over fifty years ago [10], and with the arrival of Massive Open Online Courses (MOOCs), so-called "autograders" are receiving renewed attention. The appeal is obvious: students not only get immediate feedback, but can now be given multiple opportunities to resubmit their code to improve on their mistakes, providing the opportunity for mastery learning [2]. Over their long history, autograders have evolved from test-harness libraries that must be linked against student code to web-based systems that

perform both dynamic tests and static analysis [4]. Autograders have also found use in residential classrooms, with some instructors even finding that grades on autograded programming assignments are a surprisingly good predictor of final course grades [13].

From 2008 to 2010, authors Fox and Patterson refocused UC Berkeley's one-semester (14-week) undergraduate software engineering course [6, 8] on agile development, emphasizing behavior-driven design (BDD)[1] and automated testing. A key goal of the redesign was to promote software engineering methodologies by giving students access to best-of-breed tools to immediately practice those methodologies. These tools would not only enable the students to learn immediately by doing, but also provide quantitative feedback for instructors to check students' work. We chose Ruby on Rails as the teaching vehicle because its developer ecosystem has by far the richest set of such tools, with a much stronger emphasis on high productivity, refactoring, and beautiful code than any other ecosystem we'd seen. The choice of Rails in turn influenced our decision to use Software as a Service (SaaS) as the learning vehicle, rather than (for example) mobile or embedded apps. In just 14 weeks, third- and fourth-year students learn Ruby and Rails (which most haven't seen before), learn the tools in Figure 1, complete five programming assignments, take three exams, and form "two-pizza teams" of 4–6 to prototype a real SaaS application for a nonprofit, NGO, or campus unit, over four two-week agile iterations.

Tool and URL	**Purpose**
RSpec (`rspec.info`)	Ruby unit testing
SimpleCov (`rubygems.org/gems/simplecov`)	Ruby C0 test coverage
Jasmine (`jasmine.github.io`)	JavaScript unit testing
Autotest (`github.com/seattlerb/zentest`	Continuous-testing automation
Cucumber (`cukes.info`)	behavior-driven design, integration testing
Pivotal Tracker (`pivotaltracker.com`)	agile project management
CodeClimate (`codeclimate.com`)	code quality metrics
Travis (`travis-ci.org`)	Continuous-integration testing
GitHub (`github.com`)	Source/configuration management
Heroku (`heroku.com`)	platform-as-a-service deployment

Fig. 1. The most important tools we use, all of which are either open source downloads or offer a free hosted version sufficient for class projects.

The new course was offered experimentally in 2009–2010 and was immediately successful; growing enrollment demand (from 45 in the pilot to 240 in Spring 2015) led us to write a book around the course [7] and to start thinking about how to scale it up. Coincidentally, in mid-2011 our colleagues Prof. Andrew Ng and Prof. Daphne Koller at Stanford were experimenting with a MOOC platform which would eventually become Coursera, and invited us to try adapting part of our course to the platform as an experiment. With the help of some very strong teaching assistants, we not only created Berkeley's first MOOC, but also the initial versions of the autograder tech-

[1] http://guide.agilealliance.org/guide/bdd.html

nology described here. To date, we estimate over 1,500 engineer-hours have been invested in the autograders, including contributions from MOOC alumni, from the AgileVentures[2] open development community, and from instructors using our MOOC materials as a SPOC [9].

2 Cloud Grading Architecture With OpenEdX

We adopt a narrow Unix-like view of an autograder: it is a stateless command-line program that, given a student work submission and a rubric, computes a score and some textual feedback. We treat separately the question of how to connect this program to a Learning Management System (LMS). All other policy issues—whether students can resubmit homeworks, how late penalties are computed, where the gradebook is stored, and so on—are independent of the autograder[3], as is the question of whether these autograders should replace or supplement manual grading by instructors. While these issues are pedagogically important, for engineering purposes we declare them strictly outside the scope of the autograder code itself.

2.1 Why Another Autograder?

Given that 17 autograding systems and over 60 papers about them were produced from 2006–2010 alone [11], why did we choose to build our own? First, as the survey authors point out, many existing systems' code is not readily available or is tightly integrated to a particular Learning Management System (LMS). We needed to integrate with Coursera and later OpenEdX, both of which were new and had not yet embraced standards such as Learning Tools Interoperability[4]. Unlike most previous systems, ours would need to work at "cloud scale" and respond to workload spikes: the initial offering of our MOOC in February 2012 attracted over 50,000 learners, and we expected that thousands of submissions would arrive bunched together close to the submission deadline. For the same reason, our graders needed to be highly insulated from the LMS, so that students whose code accidentally or deliberately damaged the autograder could not compromise other information in the LMS. Similarly, the autograders had to be trustworthy, in that the student assignments were authoritatively graded on trusted servers rather than having students self-report grades computed by their own computers (although of course we still have no guarantee that students are doing their own work).

[2] http://agileventures.org

[3] Due to an implementation artifact of OpenEdX, the autograders currently do adjust their scores to reflect late penalties, based on metadata about due dates provided with each assignment submission.

[4] http://imsglobal.org/lti

2.2 Student Experience and Cloud Grading Architecture

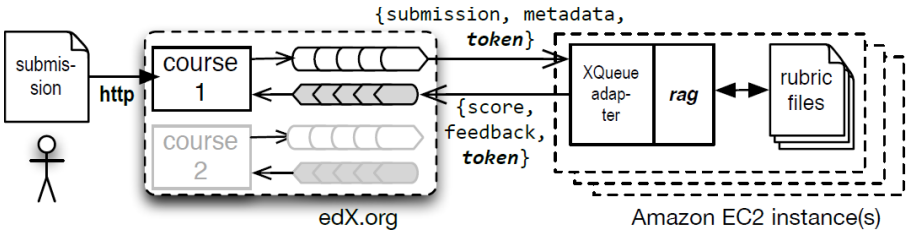


Fig. 2. Since MAGIC relies on many libraries, tools, support files, and so on, we encapsulate it in a virtual machine image that is deployed on Amazon Elastic Compute Cloud. When a new instance is started, the autograder script automatically runs from **`/etc/init.d`** and examines a deploy-time environment variable to obtain the credentials needed to make calls to the XQueues.

Our initial implementation of autograding was designed to work with Coursera and later adapted to OpenEdX. Both the API and the student experience are similar between the two. A logged-in student navigates to a page containing instructions and handouts for a homework assignment; when ready, the learner submits a single file or a **`tar`** or **`zip`** archive through a standard HTTP file-upload form. A short time later, typically less than a minute, the student can refresh the page to see feedback on her work from the autograder.

As Figure 2 shows, the student's submitted file, plus submission metadata specified at course authoring time, go into a persistent queue in the OpenEdX server; each course has its own queue. We use the metadata field to distinguish different assignments so that the autograder knows which engine and rubric files must be used to grade that assignment. The OpenEdX LMS defines an authenticated RESTful API[5] by which external standalone autograders can retrieve student submissions from these queues and later post back a numerical grade and textual feedback. The external grader does not have access to the identity of the learner; instead, an obfuscated token identifies the learner, with the mappings to the learners' true identities maintained only on OpenEdX. Hence no sensitive information connecting a work product to a specific student is leaked if the autograder is compromised. Once a submission is retrieved from the queue, the metadata identifies which grader engine and instructor-supplied rubric files (described subsequently) should be used to grade the assignment. The engine itself, *rag* (Ruby AutoGrader), is essentially a Unix command-line program that consumes the submission filename and rubric filename(s) as command-line arguments and produces a numerical score (normalized to 100 points) and freeform text feedback. The XQueueAdapter in the figure is a wrapper around this program that retrieves the submission from OpenEdX and posts the numerical score and feedback (formatted as a JSON object) back to OpenEdX.

[5] http://edx-partner-course-staff.readthedocs.org/en/latest/exercises_tools/external_graders.html

This simple architecture keeps the grader process stateless, thereby simplifying the implementation of cloud-based graders in three ways:

1. **No data loss**. If an assignment is retrieved but no grade is posted back before a pre-set timeout, OpenEdX eventually returns the ungraded assignment to the queue, where it will presumably be picked up again by another autograder instance. Therefore, if an autograder crashes while grading an assignment, no student work is lost.
2. **Scale-out**. Since the autograders are stateless consumers contacting a single producer (the queue), and grading is embarrassingly task-parallel, we can drain the queues faster by simply deploying additional autograder instances. Since we package the entire autograder and supporting libraries as a virtual machine image deployed on Amazon's cloud, deploying an additional grader is essentially a one-click operation. (We have not yet had the time to automate scaling and provisioning.)[6]. Even our most sophisticated autograders take less than one machine-minute per assignment, so at less than 10 cents per machine-hour, MOOC-scale autograding is cost-effective and fast: even with thousands of assignments being submitted in a 50,000-learner course, students rarely waited more than a few minutes to get feedback, and we can grade over 1,000 assignments for US $1.
3. **Crash-only design [3]**. If the external grader crashes (which it does periodically), it can simply be restarted, which we in fact do in the body of a `while(true)` shell script. If the entire VM becomes unresponsive, for example if it becomes corrupted by misbehaving student code, it can be rebooted or undeployed as needed, with no data loss.

In short, the simple external grader architecture of OpenEdX provides a good separation of concerns between the LMS and autograder authors.

2.3 CI Workflow for Autograders

Since at any given time the autograders may be in use by the MOOC and several campus SPOCs (Small Private Online Courses [5]), it is important to avoid introducing breaking changes to rubric files, homework assignments, or the autograders themselves. We set up continuous integration tasks using Travis-CI, which is integrated with GitHub. When a pull request is made[7], the CI task instantiates a new virtual machine, installs all the needed software to build an autograder image based on the codebase as it would appear after the pull request, and tests the autograder with known solutions versioned with each homework, as Figure 3 shows. Each homework

[6] OpenEdX also supports an alternative "push" protocol in which each student submission event triggers a call to a RESTful autograder endpoint. We do not use this alternative protocol because it thwarts this simple scaling technique and because we would be unable to limit the rate at which submissions were pushed to our autograders during peak times.

[7] A pull request is GitHub's term for the mechanism by which a developer requests that a set of changes be merged into the production branch of a codebase.

assignment repository also has a CI task that automates the installation of the autograders and verifies their configuration.

```
Feature: Testing instructor created homeworks
  As a AutoGrader maintainer
  In order to check that the supplied homework can be graded
  I would like these homeworks to be automatically tested

  Scenario Outline: Runs AutoGrader with a given spec and subject
    Given I have the homework in "."
    When I run AutoGrader for <test_subject> and <spec>
    Then I should see that the results are <expected>
    And I should see the execution results with <test_title>
  Examples:
    | test_title   | test_subject          | spec         | expected      |
    | vs solution | solutions/hw4.tar.gz | rag/hw4.yml | Score: 100.0 |
    | vs skeleton | public/hw4.tar.gz     | rag/hw4.yml | Score: 0.0   |
```

```
When(/^I run AutoGrader for (.*) and (.*)$/) do |test_subject, spec|
  run_ag("#{@hw_path}/#{test_subject}", "#{@hw_path}/#{spec}")
end

Then(/^I should see that the results are (.*)$/) do |expected_result|
  expect(@test_output).to match /#{expected_result}/
end
```

Fig. 3. Top: Cucumber integration test that is run whenever the autograder or homework code is updated. (Cucumber is described in the next section.) The scenarios verify that, at a minimum, the autograder reports a score of 100% when run against the instructor's reference solution and a score of 0% when run against the empty "code skeleton" provided to students. Bottom: examples of the Cucumber step definitions invoked when these steps are run.

3 *rag*, a Ruby Autograder for ESaaS

Having chosen Ruby and Rails for their excellent testing and code-grooming tools, our approach was to repurpose those same tools into autograders that would give finer-grained feedback than human graders using more detailed tests, and would be easier to repurpose than those built for other languages.

`rag`[8] is actually a collection of three different autograding "engines" based on open-source testing tools, as Figure 4 shows. Each engine takes as input a student-submitted work product and one or more rubric files whose content depends on the grader engine[9], and grades the work according to the rubric.

The first of these (Figure 4, left) is **RSpecGrader**, based on RSpec, an XUnit-like TDD framework that exploits Ruby's flexible syntax to embed a highly readable unit-

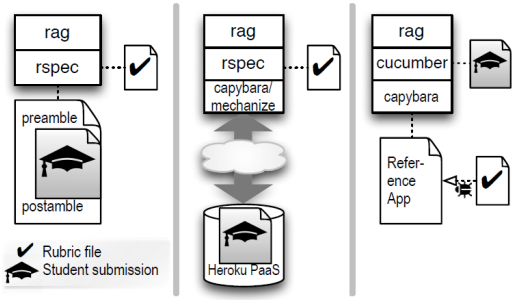
[8] http://github.com/saasbook/rag

[9] Currently the rubric files must be present in the local filesystem of the autograder VM, but refactoring is in progress to allow these files to be securely loaded on-demand from a remote host so that currently-running autograder VMs do not have to be modified when an assignment is added or changed

testing DSL in Ruby. The instructor annotates specific tests within an assignment with point values (out of 100 total); RSpecGrader computes the total points achieved and concatenates and formats the error/failure messages from any failed tests, as Figure 6 shows. RSpecGrader wraps the student code in a standard preamble and postamble in which large sections of the standard library such as file I/O and most system calls have been stubbed out, allowing us to handle exceptions in RSpec itself as well as test failures. RSpecGrader also runs in a separate timer-protected interpreter thread to protect against infinite loops and pathologically slow student code.

A variant of RSpecGrader is **MechanizeGrader** (Figure 4, center). Surveys of recent autograders [11, 4] mentioned as a "future direction" a grader that can assess full-stack GUI applications. MechanizeGrader does this using Capybara and Mechanize[10]. Capybara implements a Ruby-embedded DSL for interacting with Web-based applications by providing primitives that trigger actions on a web page such as filling in form fields or clicking a button, and examining the server's delivered results pages using XPath[11], as Figure 7 shows. Capybara is usually used as an in-process testing tool, but Mechanize can trigger Capybara's actions against a remote application, allowing black-box testing. Students' "submission" to MechanizeGrader is therefore the URL to their application deployed on Heroku's public cloud.

Finally, one of our assignments requires students to write integration-level tests using Cucumber, which allows such tests to be formulated in stylized plain text, as Figure 8 shows. Our autograder for this style of assignment is inspired by mutation testing, a technique invented by George Ammann and Jeff Offutt [1] in which a testing tool pseudo-randomly mutates the program under test to ensure that some test fails as a result of these introduced errors.



Grader	Based on	Student Assignment Type
RSpec Grader	unit testing	Student submits one or more class files for black-box and glass-box unit tests
Mechanize Grader	remote integration testing	Black-box tests stimulate student app deployed on public cloud and parse/analyze output
Feature Grader	simplified mutation testing	Evaluates coverage and fragility of student-provided integration tests by inserting bugs into reference app

Fig. 4. Summary of the autograder engines based on our repurposing of excellent existing open-source tools and testing techniques. Only the RSpecGrader is Ruby-specific.

[10] jnicklas.github.io/capybara, rubygems.org/gem/mechanize
[11] http://w3.org/TR/xpath20/

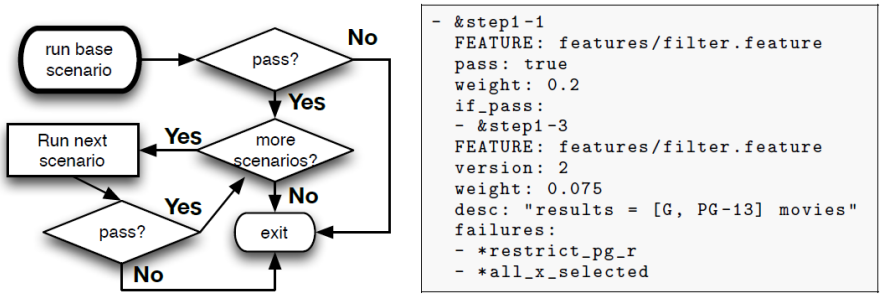


Fig. 5. FeatureGrader workflow and example YAML file. In this case if Step1-1 passes, Step1-3 will be run next. Earlier steps must be less restrictive than later steps (if the earlier step fails, there should be no way that a later one could pass). failures are the two student-provided Cucumber scenarios that should fail when run because of mutations (bugs) inserted in the app.

Specifically, **FeatureGrader** (Figure 4, right) operates by working with a reference application designed so that its behavior can be modified by manipulating certain environment variables. Each student-created test is first applied to the reference application to ensure it passes when run against a known-good subject. Next the FeatureGrader starts to mutate the reference application according to a simple specification (Figure 5), introducing specific bugs and checking that some student-created test does in fact fail in the expected manner in the presence of the introduced bug.

4 Lessons and Future Work

Both surveys of autograders ask why existing autograders aren't reused more, at least when the programming languages and types of assignments supported by the autograder match those used in courses other than the one(s) for which the autograder was designed. We believe one reason is the configuration required for teachers to deploy autograders and students to submit work to them. Since we faced and surmounted this problem in deploying our "autograders as a service" with OpenEdX, we can make them easy for others to use. We already have several instructors running SPOCs based on our materials [9] using OpenEdX, not only using our autograders but creating new assignments that take advantage of them. We are completing a major refactoring that should allow our autograders to be used entirely as a service by others and a toolchain to create autogradable homeworks for use in conjunction with the ESaaS course materials.

We now discuss how we are addressing ongoing challenges resulting from lessons learned in using these autograders for nearly three years.

Tuning rubrics. When rubrics for new assignments are developed, it is easy to overlook correct implementations that don't match the rubric, and easy to forget "preflight checks" that may cause the grader process to give up (for example, checking that a function is defined in the appropriate class namespace before calling it, to avoid a "method not found" exception). Similarly, if tests are redundant—that is, if the same

single line of code or few lines of code in a student submission causes all tests in a group to either pass or fail together— then student scoring is distorted. (This is the more general problem of test suite quality in software engineering.) In general we try to debug rubrics at classroom scale and then deploy the assignments to the MOOC, relying on the CI workflow to ensure we haven't broken the autograding of existing assignments.

Avoiding "Autograder-Driven Development." Because feedback from the autograder is quick, students can get into the habit of relying on the autograder for debugging. To some extent we have turned this into a benefit by showing students how to use RSpec and Cucumber/Capybara on their own computers[12] and run a subset of the same tests the instructors use, which is much faster and also gives them access to an interactive debugger.

Combining with manual grading. In a classroom setting (though usually not in a MOOC), instructors may want to spot-check students' code manually in addition to having it autograded. The current workflow makes it a bit awkward to do this, though we do save a copy of every graded assignment.

Grading for style. As Douce et al. observe [4], one flaw of many autograders is that "A program. . . may be correct in its operation yet pathological in its construction." We have observed this problem firsthand and are developing "crowd-aware" autograders that take advantage of scale to give students feedback on style as well as correctness. This work is based on two main ideas. The first is that a small number of clusters may capture the majority of stylistic errors, and browsing these clusters can help the instructor quickly grasp the main categories of stylistic problems students are experiencing [15]. The second is that within a sufficiently large set of student submissions, we can observe not only examples of excellent style and examples of very poor style, but enough examples in between that we can usually identify a submission that is *slightly* more stylistic than a given student's submission [12]. We can then use the differences between two submissions as the basis for giving a hint to the student whose submission is stylistically worse.

Cheating. Woit and Mason [14] found that not only is cheating rampant (in their own 5-year study and supported by earlier studies), as demonstrated dramatically by students who got high marks on required programming assignments but failed the exact same questions when they appeared on exams, but also that students don't do optional exercises. Notwithstanding these findings—and we're sure plagiarism is occurring in both our MOOC and campus class—plagiarism detection has been a non-goal for us. We use these assignments as formative rather than summative assessments, and we have the option of using MOSS[13]. That said, we continue to work on strengthening the autograders against common student attacks, such as trying to generate output that mimics what the autograder generates when outputting a score, with the goal of getting the synthetic output parsed as the definitive score.

[12] These and all the other tools are preinstalled in the virtual machine image in which we package all student-facing courseware, available for download from saasbook.info.

[13] http://theory.stanford.edu/~aiken/moss

5 Conclusions

The autograders used by ESaaS have been running for nearly three years and have graded hundreds of thousands of student assignments in our EdX MOOC, our campus Software Engineering course, and many other instructors' campus courses (SPOCs) that use some or all of our materials. The substantial investment in them has paid off and we are continuing to improve and streamline them for future use. Instructors interested in adopting them for their course, or in creating adapters to connect them to other LMSs, are invited to email **`spoc@saasbook.info`**.

Acknowledgements

We thank Google for early support of this effort as well as support for the current refactoring to further scale the courses that use this technology. We thank the technical teams at Coursera and edX for their early support for this course by providing the necessary external grader APIs. We thank the AgileVentures team for both helping steward the MOOC and providing substantial development and maintenance effort on the autograders, especially with their contribution of the CI workflows.

Finally, thanks to the many, many UC Berkeley undergraduate and graduate students who have contributed to the development of the autograders, including James Eady, David Eliahu, Jonathan Ko, Robert Marks, Mike Smith, Omer Spillinger, Richard Xia, and Aaron Zhang.

References

1. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008), http://www.amazon.com/Introduction-Software-Testing-Paul-Ammann/dp/0521880386
2. Bloom, B.S.: Mastery learning. Mastery learning: Theory and practice pp. 47–63 (1971)
3. Candea, G., Fox, A.: Crash-only software. In: Proc. 9th Workshop on Hot Topics in Operating Systems. Sante Fe, New Mexico (Jun 2009)
4. Douce, C., Livingstone, D., Orwell, J.: Automatic test-based assessment of programming: A review. J. Educ. Resour. Comput. 5(3) (Sep 2005), http://doi.acm.org/10.1145/1163405.1163409
5. Fox, A.: Viewpoint: From MOOCs to SPOCs: How MOOCs can strengthen academia. Communications of the ACM 56 (Dec 2013), http://cacm.acm.org/magazines/2013/12/169931-from-moocs-to-spocs/abstract
6. Fox, A., Patterson, D.: Crossing the software education chasm. Communications of the ACM 55(5), 25–30 (May 2012)
7. Fox, A., Patterson, D.: Engineering Software as a Service: An Agile Approach Using Cloud Computing. Strawberry Canyon LLC, San Francisco, CA, 1st edition edn. (2014)
8. Fox, A., Patterson, D.A.: Is the new software engineering curriculum agile? IEEE Software (September/October 2013)

9. Fox, A., Patterson, D.A., Ilson, R., Joseph, S., Walcott-Justice, K., Williams, R.: Software engineering curriculum technology transfer: Lessons learned from MOOCs and SPOCs. Tech. Rep. UCB/EECS-2014-17, EECS Department, University of California, Berkeley (Mar 2014), http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-17.html
10. Hollingsworth, J.: Automatic graders for programming classes. Commun. ACM 3(10), 528–529 (Oct 1960), http://doi.acm.org/10.1145/367415.367422
11. Ihantola, P., Ahoniemi, T., Karavirta, V., Sepp¨al¨a, O.: Review of recent systems for automatic assessment of programming assignments. In: Proceedings of the 10th Koli Calling International Conference on Computing Education Research. pp. 86–93. Koli Calling '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1930464.1930480
12. Moghadam, J., Choudhury, R.R., Yin, H., Fox, A.: AutoStyle: Toward coding style feedback at scale. In: 2nd ACM Conference on Learning at Scale. Vancouver, Canada (March 2015), http://dx.doi.org/10.1145/2724660.2728672, short paper
13. Navrat, P., Tvarozek, J.: Online programming exercises for summative assessment in university courses. In: Proceedings of the 15th International Conference on Computer Systems and Technologies. pp. 341–348. CompSysTech '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2659532.2659628
14. Woit, D., Mason, D.: Effectiveness of online assessment. In: Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education. pp. 137–141. SIGCSE '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/611892.611952
15. Yin, H., Fox, A.: Clustering student programming assignments to multiply instructor leverage. In: 2nd ACM Conference on Learning at Scale. Vancouver, Canada (March 2015), http://dx.doi.org/10.1145/2724660.2728695, short paper

Appendix: Code examples

```
1  describe "#my_sum" do
2    it "should be defined" do
3      expect { my_sum([1,3,4]) }.not_to raise_error
4    end
5    it "does not use built-in library function" do
6      arg = [1,2]
7      expect(arg).not_to receive(:sum)
8      my_sum(arg)
9    end
10   it "returns correct sum [20 points]" do
11     my_sum([1,2,3,4,-5,5,-100]).should be_a_kind_of Fixnum
12     my_sum([1,2,3,4,-5,5,-100]).should == 90
13   end
14   it "returns 0 for the empty array [10 points]" do
15     expect { my_sum([]) }.not_to raise_error
16     my_sum([]).should == 0
17   end
18 end
```

Fig. 6. In an RSpecGrader rubric, some test cases are "sanity checks" without which the assignment isn't even graded (lines 2–9) while others contribute points to the student's score. Ruby's dynamic language features allow us to easily check, for example, that a student's implementation of a "sum all the numbers" method does not simply call a built-in library method (line 7).

```
it "has checkboxes for ratings [5 points]" do
  @page.form_with(:id => 'ratings_form').checkboxes.each do |checkbox|
    checkbox[:id].should =~ /ratings_\w+/
  end
end

it "displays movie rating in 2nd column of movies table [5 points]" do
  rating = @page.search("table[@id=movies]/tbody/tr[1]/td[2]").text
  rating.should_not be_empty
end

it "should only display movies of selected rating [20 points]" do
  %(G PG PG-13 R).each do |rating|
    # Check corresponding rating, uncheck all other ratings
    ratings_form.checkboxes.each do |box|
      if box['id'] == "ratings_#{rating}" then box.check else box.
          uncheck end
    end
    response = form.button_with(:id => 'ratings_submit').submit
    MoviesTable.new(response).each_table_row do |row|
      row.columns[:rating].should == rating
    end
  end
end
```

Fig. 7. This excerpt of three test cases from a MechanizeGrader rubric runs against a student's deployed full-stack application.

```
Feature: display list of movies sorted by different criteria

Scenario: sort movies alphabetically
  Given movies are in the database
  When I follow "Movie Title"
  Then I should see all movies sorted by title in increasing order
```

```
When /^(?:|I )follow "([^"]*)"$/ do |link|
  click_link(link)
end
```

Fig. 8. Cucumber accepts integration tests written in stylized prose (top) and uses regular expressions to map each step to a *step definition* (bottom) that sets up preconditions, exercises the app, or checks postconditions. Step definitions can stimulate a full-stack GUI-based web application in various ways, including remote-controlling a real browser with Webdriver (formerly Selenium) or using the Ruby Mechanize library to interact with a remote site. Our code blocks are in Ruby, but the Cucumber framework itself is polyglot.