Teaching programming using industry tools

Martijn Verkleij
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
m.f.verkleij@student.utwente.nl

ABSTRACT

Education of software development is a field of significant interest, because there is a substantial shortage of software developers both in the Netherlands[9] and outside[12]. The number of students enrolling into programming studies in the Netherlands is also increasing[1]. Many educational systems exist to aid educators with teaching programming courses. These systems are built with the specific goal of aiding teaching, but do little to prepare students for tools used in professional environments. This research will trial a combination of tools used in professional environments and show if and how these can fulfil the need of educators when teaching programming. First, the primary needs of educators of programming courses is determined. Next, the landscape of professional tools is explored, and a toolchain is determined. Then, the toolchain will be compared against the learning goals that need to be met for teaching software development. Finally, a trial with educators will be conducted with the toolchain, of which the resulting experiences and insights are reported.

Keywords

Software development education, automated feedback systems, automated assessment, industry software

1. INTRODUCTION

The job market has a significant demand for software engineering jobs[9, 12]. The influx of students into studies in this field has increased in response to this demand.

However, an increased influx of students creates a problem of scalability. Computer science educators need a scalable assessment system that can be used to track the progress of students. Automated assessment has the additional benefit of providing an additional means of feedback, which helps students to directly assess their own work.

Software systems have been designed to help in education in this field, but these systems do not prepare students for the workflow found in professional environments. If conventional systems that are in use in software development can be repurposed to support computer science education, this not only saves implementation time, but also serves a secondary goal of familiarizing students with a *Continuous Integration* workflow.

The problems will be addressed with the following research questions:

1. Can industry software be used to assist educators in teaching university programming concepts?
   (a) What are the needs of educators for such software?
   (b) Which industry software can be used for this purpose?
   (c) Does this software satisfy the needs of educators and students?

The rest of the paper is structured as follows: Section 2 elaborates on existing educational systems. Section 3 discusses an exploration into tools used in the industry that are of interest to our research question. Section 4 explains the educational needs that have been identified and that are considered relevant. Section 5 lays out a toolchain that can be used to assist educators in teaching programming. Section 6 explains recommendations and guidelines regarding the toolchain. Section 7 evaluates the toolchain from section 5 and tests it against the educational needs identified in section 4. Section 8 summarizes the points made in the above chapters and draws a conclusion.

2. EXISTING EDUCATIONAL SYSTEMS

Over the years, many systems were developed for the purpose of teaching programming. These systems started to emerge as soon as programming started to be taught, with the first reported example being Hollingsworth's system from 1960 on punch cards that ran student code versus a model solution program, and reported "assignment complete" or "wrong answer"[4].

Since then, many developments have been made. Automated assesment systems could use input-output pairs instead of model solutions, they started keeping track of program execution time and they maintained a gradebook across multiple exercises and students.

The use of Hollingsworth's punch-card assessment tool revealed that students could deliberately run malicious code and system security has been a major point of concern ever since.

2.1 Introduction of tools

In the early days of automated assessment, most systems were systems that were developed from the ground up. Tools that are currently used to do software development with, like code testing frameworks, operating system and programming toolchains did not yet exist. As soon as they did come available however, they started to be used.

31th Twente Student Conference on IT, June 5th, 2019, Enschede, The Netherlands.

In 1989, Isaacson and Scott[7] wrote a very similar program to what was already available in earlier 'first-gen' systems, but had the convenience of doing it with a shell script. A significant improvement over what Hollingsworth used at the time, an IBM 650 that only had an instruction set. It is capable of compiling student code in multiple languages and limited execution time. It automatically compiled and tested code from a directory that students uploaded their code into.

In the same year, a system was built by Reek[18] that behaved much the same way, but allowed students to run the assignment checks.

2.2 Web-based systems

The widespread adoption of the world wide web has resulted in the emergence of web-based assessment systems. Where earlier systems were essentially manually-ran scripts, mostly through a command line, web-based systems significantly increased usability by providing a user interface. Additionally, web resources can be accessed from anywhere, further improving usability.

CourseMarker, a system developed and used at Nottingham University shows the benefits a web-based system may have. It has a user interface, a content management system and provides multiple types of feedback, from assessment on correctness, complexity, speed of implementation and code style.

2.3 Programming languages

Automated assessment systems were very often built with the programming languages that were available. The first systems processed punch cards, then ALGOL, FORTRAN and Ada. Later systems worked with higher level languages like C, C++, Java and Python. Systems were also developed for other programming paradigms like logic programming in Prolog and functional programming in Lisp, Haskell and Scheme.

With the introduction of the world wide web programming languages for the web started to be used like PHP and Javascript. However, many of these online assesment systems kept using the higher level languages that were common in previous assessment systems.

2.4 State of the art

P. Ihantola et al. have categorized the developments in automated assessment software after 2005[6]. They found that of all the languages seen in recent tools, Java is by far the preferred language. Other languages include Python and C/C++.

They also found that unit testing is the most popular method of feedback generation, followed by output comparison. Another recent development is the use of a pool of correct answers for checking[13].

Further, new systems have been developed that allow manual assessment as well, by allowing the educator to see submitted code and provide additional feedback per student if desired.

Security of recent systems is improved by sandboxing, to increase the security of an automated assessment server. Other ways of detecting malicious code are done through Static Analysis[2].

3. EXISTING INDUSTRY TOOLS

Relevant industry technologies considered for use in an automated assessment system are Version Control Systems, Continuous Integration Systems and Code Inspection tools for Java and Python.

3.1 Version control systems

Version Control Systems have existed since the 80's, when the first proprietary solutions started to appear. Version Control Systems can be categorized in Revision Control Systems, Client-server Version Control Systems and Distributed Version Control Systems.

Revision Control Systems like RCS and SCCS track revisions of files on a local machine. These are considered the first version control systems. Today their use is limited, as other Version Control Systems have superseded these systems.

At around the same time, client-server version control systems started to appear. Collaboration is possible in these systems because all revisions are submitted to a server, from where changes can be made by multiple users. Popular open-source implementations are CVS and Subversion. Closed source alternatives like ClearCase, Perforce Helix and Team Foundation Source Control also see some use.

Later, distributed version control systems like Git, Bitkeeper, Mercurial (hg) and Bazaar were developed. These open source systems take a peer-to-peer approach, all clients keep full copies of the history of the repository, which allows for offline work and has the advantage of not relying on a single server for operation.

3.2 Continuous integration

Continuous integration is a software development practise that focuses on often merging developers' working copies into a single main copy. Within the practises used in this technique, we are most interested in build automation.

Build automation is the practise of using a system that executes builds on code. These builds are used to check whether the code compiles and runs, but is also used to run code inspection tools. The idea is that after every contribution of code the code is built, such that problems are identified quickly.

Popular tools in this regard are the open-source Cruise Control, Gump and Jenkins and the closed-source Bamboo, TeamCity, TravisCI and Team Foundation Server.

3.3 Code inspection and code review

Code inspection in industry contexts is usually done by both dynamic and static code analysis. Dynamic code analysis techniques usually consist of unit tests, system tests or simply compiling the code. Static analysis techniques usually check coding convention in terms of syntax, detecting violations of code style standards.

3.4 Web services

Many web services exist that fulfil some, if not many of the technologies mentioned above. Some of these services combine version control with build automation to form true Continuous Development platforms. Github together with TravisCI, Gitlab (CI) and Bitbucket with Bamboo are the most prominent examples. Web services also offer code inspection. Some of these web services include SonarQube, Codacy and Better Code Hub.

4. EDUCATIONAL NEEDS

4.1 Automated feedback systems

Automated feedback systems have been developed for the purpose of helping students with learning to program. Programming is considered to be a hard concept to learn[11] and helping them individually is becoming too hard given the increasing amount of students[14]. Giving feedback to students is important, and helps them in learning new concepts[19]. By automating this feedback, students can receive more of it. Feedback received by automated feedback systems can be used by students to directly assess their work, which gives them insight into their learning progress[3].

Types of feedback

The most popular types of feedback that exist in automated feedback systems are *knowledge about mistakes* and *knowledge about how to proceed*[10].
Knowledge about mistakes concerns reporting mistakes students make, which can be done through unit tests like in *COALA*[16] or *Testovid*[20].
Knowledge about how to proceed helps students take a next step by giving a hint on how to proceed, either by giving back explicit help messages based on common mistakes like *Proust*[8], or by guiding the student through by laying out a structure for the student to fill in[5].

Feedback generation

The technologies most often used to generate feedback are *automated testing*, *static analysis* and *program transformations*[10]. *Automated testing* is usually done by running unit tests against student code to find problems. *Static analysis* finds problems by looking for common syntax errors[15] or calculation of metrics like cyclomatic complexity. *Program transformations* concerns trying to match student code with a model solution by abstracting away differences between the two that do not affect their behaviour, such as done in SIPLeS-II[21].

Adaptability

Adaptability describes the extent to which an educator can define exercises in the system and influence the feedback given to students. The most often used techniques here are *model solutions*, *test data* and *solution templates*.
With *model solutions* frameworks can check student solutions by comparing the outputs or comparing the structure of the solution. To support multiple algorithms that can generate the correct answer one can either provide multiple model solutions, or the framework can use *program transformation* techniques.
Test data concerns scripts, unit tests or simply input-output pairs that check student code.
Solution templates give students a skeleton in which they have to ‘fill the gaps’. This restricts the student in the type of solution they can give for a particular problem.

4.2 Interview

In order to get a more accurate insight into the educational needs as they are experienced by educators a plenary interview was conducted with educators in the field of Computer Science. The interviewees were asked about their opinions both in favour and against automated assessment systems and the importance of feedback types and feedback generation techniques.

Major arguments in favour of automated assessment were:

- Teaching assistants lose a lot of time on what are regarded “simple” errors. A system that runs student code for them and gives them compiler feedback may already provide answers for some students, reducing questions.
- Automated assessment is scalable, which means more students can get feedback than they could get through waiting for teaching assistants.
- The system may provide an overview for the teacher on the progress of students. This gives them diagnostic information of their course.

Major arguments against automated assessment:

- More complex exercises are hard, if not impossible to assess well.
- Much time goes into developing an environment that can assess students’ code. Especially developing tests that test code for correctness takes a lot of time to develop.
- Existing systems are hard to use. They provide bad, if any, user interfaces.
- Automated assessment systems may become a system that itself must be debugged often if it is unstable, or breaks incoherently.

Important feedback generation techniques and feedback types are:

- Through the use of static code analysis, student code may become much more readable. This conveniences student assistants when helping students.
- The possibility of using secret unit tests is something that should be considered. The main purpose of such secret tests are the prevention of fraud with the exercises or tests.
- Unit tests that simply compare values are considered to simple. More abstract code analysis is preferable. However, this can only be considered for small exercises. More complex exercises are unsuitable for this purpose.
- For smaller exercises, constraint-based modelling is a nice feature. Constraint-based modelling checks code against modelling constraints like the presence of a `for` loop or assignment of certain values.
- Program transformations, in which code is reordered to fit a model solution for assessment, should also be considered.

5. TOOLCHAIN

A toolchain is presented that consists of a version control system on a web service, a “plug-in” website that hooks into the aforementioned system to distribute an exercise framework and a Build Server that executes unit tests and code analysis tools to generate feedback (Figure 5).

5.1 Version Control

A web service to host both the exercise code and the solutions from the students makes the exchange of exercises and their solution possible. The solution uses Git to host the files. Git is considered a good choice, since it’s use is almost ubiquitous for new projects.

The Git repository consists of exercise code, Unit Tests and a Build tool definition file. The exercise code is code that is used as a skeleton for the exercises. In a separate folder one can find the unit tests that can both be used by the students and the educators to get feedback on their code. A build tool configuration file instructs the Build Server to run the unit tests and report their success or failure, and code inspection tests that assess the code style of the student.

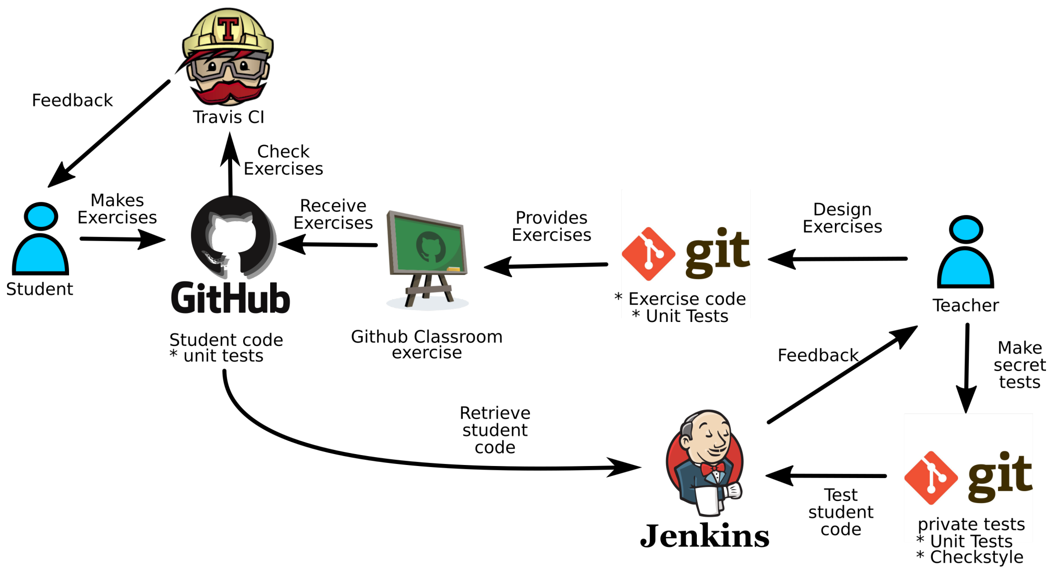

Figure 1. Toolchain

5.2 Build Server

The tool used for the assessment of students' exercises is a Build Server. The Build Server should be supplied with a series of unit tests, which test the correct functionality of the exercises made by the students. This can be done through supplying the unit tests as part of the repository, like previously mentioned. This additionaly gives the students the possibility to run the tests themselves and see whether they pass. To encourage this, a Build Server configuration file for an online build tool like TravisCI can be supplied that executes these tests for them. One could consider adding tests that test for code style as well.

Another option is to have private tests that only exist on the Build Server. These can then be used to more thoroughly check the supplied answer programs. Once again, code inspection tests could be added that check for code style.

5.3 Code inspection

Code style checks are used to make an assessment of how organised and clean the code of a student is. It allows for testing an additional dimension of code. Code must not only be correct, it must also be written in a manner that is acceptable in collaborative environments.

5.4 Test Setup

To test the feasibility and gauge the opinion of educators on the toolchain explained above, an implementation has been made using Github and Github Classroom (version control), TravisCI (Build Server for students) and Jenkins (Build Server for educator).

5.4.1 Github and Github Classroom

Github is a source control web service that is the most popular of it's kind. This means the integration with other software is mature, which is the main reason it was chosen for the test setup. It offers the possibility to group student code in an organization, in which the code is private to other students. Students can upload a Git repository in it that contains the exercise code, unit tests and Build Server definition file. This organization is later used by the build servers to access all the code in one run.

Github Classroom is a tool that automates much of this process. The teacher can provide a repository to that system, for which Github then creates an organization. Any student who then "starts" the assignment, forks that repository in that organization after which they can start working. The build server for teachers will automatically detect the students' fork and run the configured tests.

```
language: python
python:
  - "2.7"
install:
  - pip install flake8
env:
  - $EXERCISE=test_ex_1
  - $EXERCISE=test_ex_2
  - $EXERCISE=test_ex_3
  - ...
script:
  - tests $EXERCISE
```

Figure 2. Example `.travis.yml`

5.4.2 TravisCI

In order to let students get quick feedback on their code, the student can be encouraged to let their code be checked by TravisCI. The syntax between Travis and Jenkins are fairly different, with Travis being easier. However, Travis does not offer the functionality of running tests that are not inside the repository.

5.4.3 Jenkins

Jenkins is used as a build server for the educator. In order for this to work, one must install the Github Organization plugin. After it is installed, the organization made by Github Classroom can be added to Jenkins, after which it will automatically build all the student repositories.

One can also provide a "pipeline plugin" during adding this organisation, which should point to a private repository that contains secret tests from the educator. This private repository uses the "pipeline plugin" syntax. An example is given in Figure 5.4.4. This file is put in a `/vars/` directory inside the repository. Any additional files and tests may be placed in a folder alongside the `/vars/` directory/

```
# This file is put into the public repository. A
# student should not remove it.
@Library('secrettests')

node {
    # Points to /vars/secrettests.groovy in
    # private repository.
    secrettests()

    # Tricks Jenkins into only showing failed
    # tests in red.
    currentBuild.rawBuild.@result =
        hudson.model.Result.SUCCESS
}
```

Figure 3. Example `Jenkinsfile`

```
def call(Map pipelineParams){
    node {
        catchError{
            stage('checkstyle') {
                sh 'pip install flake8'
                sh 'flake8 --config=
                  jenkins/flake8.ini search'
            }
        }
        catchError{
            stage('test') {
                sh 'export DISPLAY=:0 &&
                cd search &&
                python tests.py'
            }
        }
    }
}
```

Figure 4. Example `/vars/secretexercises.groovy`

5.4.4 Exercises

To be able to demonstrate the toolchain a set of test exercises was defined in Python. The toolchain therefore used the `unittest` library to run unit tests, along with `flake8` (an external library) for code style assessment.

An exercise set for implementing A* search in Pacman, developed at the Computer Science department at Duke university was chosen to demonstrate the feature set of the test suite[17]. It is also suitable to demonstrate inspection of code style.

6. RECOMMENDATIONS AND GUIDELINES

6.1 Unit tests

The value of assessment with unit tests is entirely dependent on the quality and extent of testing of these unit tests. Writing unit tests for assessment is a trade-off between the time available for course development and extensiveness of testing. One should therefore consider writing tests a continuous process, better tests can be written by improving on the tests over successive teachings of a course.

The easiest form of assessment is testing input-output pairs, in which a predefined method is fed inputs, and is tested for returning the correct output.

6.2 Course setup

To set up a new course, one needs to set up two Github repositories:

- A public repository, containing:
  - A manual for the student, containing information on how to use the code.
  - Exercises.
  - Unit tests for the student.
  - A `Jenkinsfile` as shown in Figure 5.4.4.
  - Optional: A `.travis.yml` file as shown in Figure 5.4.2
- A private repository, containing:
  - A `.groovy` file, inside a `/vars/` directory, containing the secret tests.
  - The student' unit tests.
  - The secret unit tests.

Additionally, one needs to set up the Github Classroom exercise. As a start repository, add the public repository. From this, a github organisation results, which will be used to set up Jenkins.

In Jenkins, add a Github organisation, pointing to the organisation made by Github Classroom. Add the private repository as a "pipeline library". Make sure the `Jenkinsfile` in the public repository points to this library by referencing the name of the `.groovy` file in the `/vars/` subdirectory.

If private tests are used, it is strongly advisable to add student unit tests too. It will direct the students to write the code in such a way that the private tests actually run on them.

6.3 Code style

In order to inspect code style, a code style inspection tool is used. In this toolchain, flake8 was used. flake8 provides a configuration file, with which one can change the criteria the code style tool tests for. Caution must however be kept, since any change made by the educator steers the style away from what is standard in the industry.

6.4 Additional assessment tools

Along with unit tests, testing was shown with code style checks. One could consider adding additional tests, such as code coverage testing, language feature inspection and adding functionality from other educational tools. This functionality may however require significant rewriting of those tools, along with possibly having to redefine the assignment testing in those tools' formats.

7. EVALUATION

To evaluate the proposed toolchain, an evaluation is done of both the educational needs of educators and of the usability of the proposed toolchain.

7.1 Usability

Usability is an important part of software. In order to gauge the opinion of educators on usability an interview was conducted with educators, during which the toolchain was presented. A demo was given with the software and an explanation was given on the tasks that a teacher had to conduct to use this in a course.

Positive notes on usability:

- Jenkins:
  - It is considered very convenient to be able to check all the students' code in one go.
  - You can immediately see if a student has problems, as the build failure of all students can be seen in one page.
  - Interviewees proposed ideas for high-level tests like pylint and coverage tests, which indicate that they understand how they can extend the system to more accurately assess students.
- Github + Travis:
  - Interviewees liked to see Github, as learning to use it is considered a learning goal by itself.
  - The integration with Travis is maintenance-free for the student. The student only has to mark their repository for testing.

Negative notes on usability:

- Jenkins:
  - Detailed build results are only visible per student, not in one overview.
  - The toolchain requires a self-hosted Jenkins instance.
- Github + Travis:
  - The students can (unknowingly) sabotage the Jenkins and/or Travis build by deleting the files from their repository.

In summary, the educators are positive of the proposed system. They do see a few possible flaws, that need to be fixed or taken into account when using this in teaching.

7.2 Educational needs

The most important educational needs that were identified are the presence of both static and automated code analysis. Static code analysis is present in the toolchain in the form of code style tests. Dynamic code analysis is done through unit tests, which must be defined by the educator.

The system has support for secret exercises, which was something that was considered important.

The system does not fulfil the requirement of utilizing constraint-based modelling or program transformations. These types of feedback generation techniques do not have a parallel function in industry applications, which results in their absence.

7.3 Limitations

7.3.1 Github and Github Classroom

The toolchain is using Github and Github education. There are two main reasons why this is the case. The first is the fact that Github Classroom is a service that is unique in it's implementation. The functionality is unfortunately limited to Gitlab repositories, which means one cannot use another Version Control webservice like for example Gitlab or Bitbucket. One could replicate the method it uses behind-the-scenes, but this is outside the scope of this research.

Secondly, the plugin used to let Jenkins scan all the student repositories only exists for Github. This once again means alternatives to Github cannot be used.

Github itself also lacks features that are available in competitor platforms. Github also does not integrate TravisCI immediately, which means the student needs to set it up himself, which is an inconvenience. Competitor platforms like Bitbucket and Gitlab already integrate these services.

The displayed features also require an academic license. This means a dependency on the framework that could render the toolchain useless over time.

7.3.2 Jenkins

Jenkins is a self-hosted solution, which has both it's advantages and disadvantages. To self-host Jenkins, one needs a server to host it on, and a system administrator to maintain it.

Compared to hosted solutions it offers less integration with the rest of the toolchain, and it's user interface is quite dated.

8. CONCLUSION

(1) To better understand the needs of educators of computer science educators a literature study and an interview was conducted. (2) To find out which industry software can be used to fulfil these requirements an exploratory search was conducted in the areas of version control systems, continuous integration and code inspection. A toolchain was constructed with the results from (1) and (2). This toolchain was measured against the requirements found at (2), and another interview that focused on usability.

RQ 1.a The main needs of educators are the presence of a scalable solution that can assess student code both statically and dynamically. It should support exercises that are exclusive to the educator, because of fraud concerns. It must be easy to use. It should contain more advanced static code analysis tools like constraint-based modelling and program transformations.

RQ 1.b A toolchain can be constructed with Github and Github classroom for exercise distribution and code hosting. Students can check their code using teacher-supplied unit tests through TravisCI. Teacher can also track the progress of students through Jenkins, which additionally offers the benefit of using secret tests to better assess the students.

RQ 1.c The toolchain is considered convenient to use, it easily identifies students who have problems. Students can easily test code themselves, they do not need to set up anything aside from Travis. However, student can sabotage the system, both intentional and accidentally, by altering or removing crucial files from the repository.

The toolchain supports most of the features requested by educators, but is vitally missing constraint-based modelling and program transformations. It also commits the user to a significant amount of effort due to the amount of unit tests that need to be written.

RQ 1 Industry software like Github, Jenkins and Travis can have an assistive role as automated assessment system, but they come with a few shortcomings. Some lack competitive platforms, which could theoretically pose a problem. Other miss functionality which limits the user to certain platforms.

9. ACKNOWLEDGEMENTS

I would like to express my gratitude towards my supervisor, without whom I would not have gotten so far. I would furthermore like to thank my reviewers, for giving very valuable advice.

10. REFERENCES

[1] Hoger onderwijs; eerste- en ouderejaarsstudenten, studierichting. Technical report, Centraal Bureau Statistiek, 2019.

[2] Chen, M. et al. Design and applications of an algorithm benchmark system in a computational problem solving environment. *SIGCSE Bull.*, 38(3):123–127, June 2006.

[3] Jurado F. et al. elearning standards and automatic assessment in a distributed eclipse based environment for computer programming learning. *Computer Applications in Engineering Education*, 22:774–787, December 2014.

[4] J. Hollingsworth. Automatic graders for programming classes. *Communications of the ACM*, 3:528–529, October 1960.

[5] J. Hong. Guided programming and automated error analysis in an intelligent prolog tutor. *International Journal of Human-Computer Studies*, 61(4):505 – 534, 2004.

[6] P. et al. Ihantola. Review of recent systems for automatic assessment of programming assignments. January 2010.

[7] Peter C. Isaacson and Terry A. Scott. Automating the execution of student programs. *SIGCSE Bull.*, 21(2):15–22, June 1989.

[8] W. Lewis Johnson and E. Soloway. Proust: Knowledge-based program understanding. In C. Rich and R.C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, pages 443 – 451. Morgan Kaufmann, 1986.

[9] F. Kalkhoven. Factsheet arbeidsmarkt ict. Technical report, Uitvoeringsinstituut Werknemersverzekeringen, April 2018.

[10] H. et al. Keuning. A systematic literature review of automated feedback generation for programming exercises. *ACM Trans. Comput. Educ.*, 19(1):3:1–3:43, September 2018.

[11] W. et al. Mccracken. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin*, 33:125–180, December 2001.

[12] J. McGrath and J. Behan. A comparison of shortage and surplus occupations based on analyses of data from the european public employment services and labour force surveys. Technical report, ICON Institute, February 2017.

[13] K.A. et al. Naudé. Marking student programs using graph similarity. *Computers and Education*, 54(2):545 – 561, 2010.

[14] A. Nguyen, C. Piech, J. Huang, and L. Guibas. Codewebs: Scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 491–502, New York, NY, USA, 2014. ACM.

[15] E. Odekirk-Hash and J.L. Zachary. Automated feedback on programs means students need less help from teachers. *SIGCSE Bull.*, 33(1):55–59, February 2001.

[16] C. et al. Ott. Translating principles of effective feedback for students into the cs1 context. *ACM Trans. Comput. Educ.*, 16(1):1:1–1:27, January 2016.

[17] R. Parr and D. Klein. Search in pacman. `https://www2.cs.duke.edu/courses/spring16/compsci270/hw1/`. Accessed: 2019-06-23.

[18] K.A. Reek. The try system -or- how to avoid testing student programs. *SIGCSE Bull.*, 21(1):112–116, February 1989.

[19] V.J. Shute. Focus on formative feedback. *Review of Educational Research*, 78(1):153–189, 2008.

[20] B. et al. Vesin. Protus 2.0: Ontology-based semantic recommendation in programming tutoring system. *Expert Syst. Appl.*, 39(15):12229–12246, November 2012.

[21] P. Xu and Y. Chee. Transformation-based diagnosis of student programs for programming tutoring systems. *Software Engineering, IEEE Transactions on*, 29:360– 384, May 2003.