An Online Learning Platform for Teaching, Learning, and Assessment of Programming

Philip E. Robinson
Department of Electrical & Electronic Engineering Science
University of Johannesburg
Auckland Park, South Africa
Email: philipr@uj.ac.za

Johnson Carroll
Faculty of Engineering and the Built Environment
University of Johannesburg
Auckland Park, South Africa
Email: jcarroll@uj.ac.za

***Abstract*— In this paper the use of an open-source online learning platform to aid in teaching and assessment of computer programming in large classes is discussed. The pedagogical philosophy of how the subject of computer programming is taught is presented. Based on the skills and learning processes that are identified for effective teaching of computer programming, a strategy for employing modern web technology coupled with an automated assessment capability to meet these goals is discussed. The paper describes the technology and implementation of the learning platform and new methods for automated assessment of programming assignments and exams. Finally, the application of the system to achieve the pedagogical goals and the benefits of using the system for teaching large classes is reported.**

***Keywords*—** *Online Learning Platform, Automated Assessment, Programming*

I. INTRODUCTION

Teaching large classes is a challenging exercise, they inherently limit the amount of personal attention a student can expect from the lecturer or teaching assistants. The teaching of computer programming is particularly challenging in this environment for several reasons. First, programming is a topic that requires an unusually rapid progression through the first three learning levels of Bloom's taxonomy [1] through to application. It is only at this point that the subject matter can be clearly linked to the broader context of solving engineering and computing problems. Such a rapid progression requires significantly more hands-on practice than other subjects, as developing skills and fluency allow a student to approach, deconstruct, and solve more complex programming problems. This hands-on approach to solving problems through tinkering and actively engaging in the making of an artefact that solves a problem builds the intuition needed to become a competent and adept programmer.

In large programming classes one encounters the additional problem of a scarcity of computer resources for students, which can place a hugely problematic constraint on assessment methods. Authentic and constructively aligned assessment of programming should emulate the setting in which students might write programs in the real world: on a computer with the instantaneous feedback from the programming environment, iterative submissions, and a debugging or trial-and-error approach to producing functional programs. However, many programming classes are assessed using traditional written exams despite many lecturers' discomfort with that approach [3]. Written assessments also have a significant administrative disadvantage in that the process of grading these assessments is a laborious and extremely time consuming process. The authors have noted that grading a typical written programming exam for a 400-student class can require up to 40 man-hours.

This paper describes the design and implementation of an open-source online learning platform for teaching and assessing programming. The platform allows for online completion of programming assignments using automated assessment tools, allowing both the standard compiler/interpreter feedback as well as customized contextual instructor guidance. We describe the pedagogical approach underpinning the design of the platform, as well as the technical details of how the platform is implemented. The platform was utilized for both formative and summative assessments in an introductory programming class of 400 engineering students. We describe both the benefits and challenges encountered when utilizing the platform, and describe some of the additional information available from the online system that would not be available through traditional assessment methods. This additional data provides ample scope for future research into how students learn to program.

The remainder of this paper will be structured as follows. Section II will present the pedagogical philosophy that is employed in the implementation of the online platform. Section III will describe the implementation of the system. Section IV will discuss how the system meets the pedagogical goals laid out in Section II and, finally, Section V will provide some concluding remarks.

II. PEDAGOGICAL APPROACH

The process of learning the skill of computer programming involve understanding only a handful of basic concepts. However, these concepts form only a small part of the true complexity of solving programming problems. Even the simple programming problems will contain many emergent issues. It is effectively impossible to exhaustively teach all the possible permutations of instructions and algorithmic structures that can solve a given programming problem of moderate complexity. Hence, students of programming each construct their own body of knowledge and understanding which expands when the student encounters and overcomes additional problems. Particularly in the modern, internet-connected learning

environment, the process of learning programming can be neatly described by Papert's constructionism [2], where learning occurs through creatively solving problems rather than the transmission and reception of knowledge. This construction of knowledge is most reliably achieved when the learner is experiencing the process of constructing a meaningful product rather than reproducing a rote learned series of concepts [2]. To foster these processes, a teaching and learning strategy should allow a student to tackle problems in an environment where they can explore and tinker with their solutions while receiving real-time guidance.

The method used to assess a student's level of understanding of a topic is also an important factor to consider when designing a teaching and learning strategy. We consider two key characteristics of programming assessment: authenticity and constructive alignment, ala Biggs [12]. In order to be authentic, assessment of programming ability should allow the student to tinker with a solution with the benefit of feedback from the compiler or interpreter. This is a far more authentic situation than a written test, as a programmer is rarely (if ever) asked to produce a working program on paper. Further, if the natural method of learning programming is through a constructionist process of experimentation and problem solving, then the assessment should also take place in a context which allows the same type of experimentation during the assessment. If the format of the summative assessment mirrors both the formative assessment as well as the instructional setting, the educational outcomes and assessment should be inherently and easily aligned.

The online learning system presented in this paper addresses the pedagogical concerns of both the learning environment and assessment of programming. As described in the coming sections, the system emulates a development environment, allowing the student to engage with the programming tasks in a setting comparable to real-world programming environment. In addition to the built-in compiler and interpreter feedback, the system also allows the instructor to devise automated guidance to help students overcome common problems. The exact same system can be used for formal assessment, ensuring that the students are assessed in a realistic manner that is aligned with their learning process.

III. Structure of the Online Learning System

Over the years, various automated programming evaluation systems have been devised and implemented. Surveys such as [4] and [13] describe dozens of automated assessment tools and studies. As noted in [4], these projects often have a limited lifespan or restricted applicability due to the short-term or limited nature of the overall project, frequently a single class or postgraduate research field. However, as indicated in [13], the trend toward online education has renewed and expanded interest in effective, automated, online assessment of programming. Unfortunately, the technical exposition of the systems is usually lacking, making it difficult to reproduce and/or customize the system without substantial duplication of effort. Here, we present a full technical description of the new system, focusing on the non-standard features and system structures that allow the system to be utilized in both formative and formal summative assessment.

The system described in this work is based on the popular Moodle open-source learning platform. This platform was developed by Dougiamas, who decided to build a free and open-source learning platform focused on constructivist pedagogical principles [5]. Moodle is licensed under the GPL written for the GNU project [6], this protects the rights of end users to run, study, share and modify the software. This open-source principle has fostered a large and active development community surrounding the Moodle project with constant active contribution to the core code-base and many community developed plug-ins that extend the capabilities of the platform in a variety of ways.

Moodle itself is a PHP-based Course Management System (CMS) that can be deployed on a variety of operating systems, web servers and database systems. The full source code is available for free and is well documented which allows for end users to tinker with the system if they so please. However, easy one-click automated installation packages have also been created by companies like Bitnami to make deployment exceedingly simple [7]. Moodle provides all the core features one would expect from a CMS, which includes robust user management, diverse content management tools, scheduling tools, a variety of assessment tools, messaging, a grade management system, integration of plug-ins and support for e-learning module standards like SCORM [8].

This open community has led to the development of many useful plug-ins for Moodle, and the system described in this paper is based on a Moodle plug-in called the Virtual Programming Lab (VPL) developed and maintained by Rodríguez-del-Pino [9].

A. Virtual Programming Lab (VPL) Plug-In

The VPL plug-in is a system designed to present and assess programming assignments through the Moodle platform. The plug-in consists of three main elements. The first is the main plug-in module that runs on the Moodle server; the second is an editor component that allows for the editing of source code in the browser. Finally, there is the jail server that acts as a sandbox environment that executes the student's code.

The main plug-in module of VPL, which runs on the Moodle server, manages the descriptions of the assignments, the marking scripts and protocols, the grading process, scheduling settings, access restrictions, similarity checks and controls how a student's code will be executed on the jail server.

The editor is an integral part of the VPL plugin and provides a capable in-browser editor environment that supports syntax highlighting for the various supported languages and multiple file support through tabs. The editor allows students to edit the assignment source code, provides the interface to the development environment to receive feedback from compilers or interpreters and allows the student to submit assignments for automatic assessment, feedback and grading.

The final element of the VPL system are the jail servers. These are the servers that the VPL plug-in transmits a student's code to for execution. These servers are where the toolchains for the various supported languages are installed. As of the writing of this report, VPL can currently execute 27 languages

with varying levels of support for syntax highlighting, debugging and graphical user interfaces [10]. Executing student's code is a risky endeavour for a server as students are prone to producing bad code that can inadvertently compromise an operating system through memory leaks, infinite loops or system calls. Some students are also bound to test the limits of an execution environment and attempt to execute malicious code on the server. Therefore, VPL servers execute code in a chroot jail[1]. In this way, the VPL system can control the maximum allowed system resources that a given student program can use and protect the jail system from erroneous or malicious code during execution.

The VPL system supports using multiple jail servers for a single Moodle environment and manages the load balancing between these servers when many students are using the system simultaneously. When a student submits a program for execution it is transmitted to the jail server along with the execution scripts created by the instructor. These scripts are then used to execute the code using the appropriate compiler or interpreter. The output of the program, compiler or interpreter is then sent back to the student with a standard command-line interface [9]. VPL has recently added support for graphical output from programs in addition to the command line interface. This is achieved by using the VNC remote access software that is built into most modern Linux distributions. This interface then streams a basic windows environment back to the student's browser allowing them to interact with the graphical elements of the environment they are currently engaged with.

The VPL system also includes a source code similarity measurement system which is used to analyse the submissions for a given assignment and report back on their relative similarities using an easy to navigate interface. This tool makes it possible to determine which students submitted plagiarised code with a minimal time investment [9].

For use of VPL in strict testing environments, the plug-in includes the standard Moodle activity security features that provide the ability to control access to an activity using a password and to limit access by network addresses. This allows an instructor to limit access to an activity to a specified set of computers, such as those in the lab where the assessment is being conducted. VPL can also disable the ability to copy and paste text in the editor, which ensures that a student was the author of an activity [9].

B. Automated Assessment using VPL

The real power of VPL as a tool for teaching and assessing programming assignments is in its automated execution and assessment capabilities. VPL runs a student's code using a BASH script to prepare the source files, compile the code (or send it to the relevant interpreter) and then execute the code. The standard output stream of this process is then provided to the student via a console window in their browser where a student can interact with the running program. When a program is being assessed, a different BASH script is used to compile and execute the program. This assessment script will then provide automated input to the program and evaluate its behaviour for grading purposes [9].

Out of the box VPL includes default run, evaluation and debugging scripts for all the supported languages. A standard format for defining basic evaluation test cases is also provided. Using the built-in evaluation scripts involves defining a series of test cases where the input provided to the program is defined along with the expected output that the program should print to the standard output stream. Grading conditions are provided for each test case. VPL includes a C++ grading program which takes in the specified test cases and uses them to evaluate a student's submission and automatically grade the submission [9]. While this capability provides an easy way to quickly produce automatically assessed programming assignments it is very limited. Input can only be provided via the standard input stream and the output is naively assessed as a single output numeric or direct string comparison. This means that for a student to be graded as correct their program's output must conform exactly to the specified solution output with no extra spaces or new-line characters. This can often lead to confusion: defining the exact form of the required output can be challenging, and students will often not understand the formatting issues created by spaces and new-line characters which are difficult to spot in a text console.

Unfortunately, the execution script side of VPL is not very well documented and the only real source of information provided comes in the form of the default run and evaluation BASH scripts provided with the VPL source code. Luckily, Thiébaut has recently published some of his work on using this feature of VPL and the website accompanying this work has provided several tutorial examples of how the execution scripts used by VPL can be employed for custom evaluation of activities [11].

In VPL each activity has three default scripts created to manage the various modes of execution

- **vpl_run.sh** and **vpl_debug.sh** are called when the student clicks Run or Debug in the editor. These scripts should prepare the submitted source code and execute it using the appropriate toolchain. These execution modes allow the student to interact with the program using a console.
- **vpl_evaluate.sh** is called when the student clicks the Evaluate button which indicates they are ready to grade their submission. This script should execute the submission and evaluate its behaviour to produce a grade that is recorded in Moodle.

For custom evaluation to be conducted one must edit the **vpl_evaluate.sh** BASH script. The execution scripts work by producing an executable script called vpl_execution that VPL will execute in the chroot jail. The vpl_execution executable will contain the BASH commands to prepare the submitted source code for execution, execute it and then evaluate its output. This script must then print comments and the final grade to the standard output stream. The comments are presented to the student in the browser and VPL will parse the grade which is recorded in Moodle. Thiébaut tackles the evaluation of interpreted languages and compiled languages

[1] A chroot jail is a feature in the Unix operating system that allows one to isolate a process and limit its access to the file system and system resources [9].

using different approaches [11]. For interpreted scripting languages, Python in Thiébaut's case, it is very easy to write a custom evaluation script in the same language and then import and execute the submitted code within the evaluation script. This is convenient because using the same programming environment enables the evaluation script to natively examine variables in memory or other mechanics of the submitted code. When dealing with a compiled language, Java in Thiébaut's case, he conducts all his evaluation using the BASH script by compiling the submission code, running the resulting executable and piping the program's output into a text file. Because BASH scripts do not have access to the internal variables of a program or the standard input and output streams of a program the BASH script cannot directly interact with a running program. This means that the input to the program must be provided as command line arguments or in the form of a text file. The output of the program is piped into a text file and Linux commands like `diff` are used to compare the output to a model output file to determine the resulting grade. This method of grading is effective, but relying solely on the BASH script means that all evaluation logic and interaction with the submitted program must be done using the command-line interface in Linux, making it quite clumsy for complex evaluation.

In this work we build on the techniques proposed by Thiébaut and present an approach to running and evaluating both compiled submissions and interpreted submissions using a more capable scripting environment that allows for the implementation of more complex evaluation logic. This is done by including additional evaluation scripts for VPL to use during the evaluation process. VPL allows for the inclusion of additional files for evaluation, but one must just tell VPL which of these files must be sent to the jail server during execution in the *Advanced Settings* menu.

The course being presented using this system covers the Octave language and the C language which are an interpreted language and a compiled language respectively. First the evaluation of an Octave activity is presented as this is the simplest case for a custom evaluation script and can be seen in Code Snippet 1 and Code Snippet 2.

In Code Snippet 1 the BASH script that is executed is shown. For an interpreted language the custom evaluation script is launched using the relevant interpreter. The snippet also demonstrates how an environment variable is passed, containing the current student's ID, to the evaluation script. These variables are made available by VPL by including the **common_script.sh** script. The custom evaluation script for this activity is shown in Code Snippet 2. This activity required the student to generate the first ten Fibonacci numbers and place them in an array called *result*. With an interpreted language it is possible to execute the students script within the evaluation script. A try-catch clause could be used to gracefully catch any errors but if an error occurs the error stream will be presented to the student in the editor. This script contains a model answer that is directly compared to the students answer and feedback is provide by using the **Comment** tags. The final grade is reported using a **Grade** tag which VPL then records in Moodle. This approach is very flexible and it is possible to test the submitted code in a variety of ways and incorporate intelligent feedback and a variety of grading strategies.

When it comes to a compiled language like C, the submission must first be compiled before evaluation. In Code Snippet 3 an example of this process is shown. The program is compiled using the GCC compiler and the output of the compiler is piped into a dummy text file. This text file is then printed on the standard output stream as feedback for the student. The BASH script then checks if the compilation succeeded by confirming that the executable exists. For grading the BASH script then calls a custom Python script that will execute the compiled program.

```
#! /bin/bash
 . common_script.sh

cat > vpl_execution <<EOF
#! /bin/bash
octave -q customEval.m $MOODLE_USER_ID
EOF

chmod +x vpl_execution
```

Code Snippet 1: Example of vpl_evaluate.sh for an Octave submission

```
%Call the students submission script
main

%Model answer, calculate the first 10
%Fibonacci numbers
ansResult = [0, 1];
for i = 3:10
   ansResult(i) = ansResult(i-1)+ \
                          ansResult(i-2);
end

%Perform grading
grade = 0;
if exist('result')
  if isequal(ansResult,result)
    printf('Comment :=>> Your answer
is correct!\n');
    grade = 100;
  else
    printf('Comment :=>> Your result array
is not correct!\n');
  end
else
  printf('Comment :=>> The result does not
exist!\n');
end
printf('Grade :=>> %d\n',grade);
```

Code Snippet 2: Example of customEval.m for an Octave assignment.

```
#! /bin/bash
 . common_script.sh

cat > vpl_execution <<EOF
#! /bin/bash
gcc main.c -o testProg 2> dummy.out
cat dummy.out
if [ -e testProg ]; then
  echo "No Compile Error"
  python3.4 customEval.py $MOODLE_USER_ID
fi
EOF

chmod +x vpl_execution
```

Code Snippet 3: Example of vpl_evaluate.sh for a C submission

```
import sys
import subprocess
#Model answer to calculate first 10
#Fibonacci Numbers
ansFibonacci = [0, 1]
for i in range(2,10):
  ansFibonacci.append(ansFibonacci[i-1] \
  +ansFibonacci[i-2])

p = subprocess.Popen(["./testProg"], \
stdout=subprocess.PIPE,stdin=        \
subprocess.PIPE,universal_newlines = \
True);
output = p.communicate(input='10')[0];
#The following lines split the output
#stream using \n as a delimiter and
#removes any empty terms
terms = output.split('\n')
terms = [i for i in terms if i != '']
terms = [int(i) for i in terms]
if terms == ansFibonacci:
  print("Comment :=>> Correct answer!")
  print("Grade :=>> 100")
else:
  print("Comment :=>> Incorrect!")
  print("Grade :=>> 0")
```

Code Snippet 4: Example of a custom Python evaluation script (customEval.py) for a compiled C activity

In Code Snippet 4 an example of a Python script being used to evaluate a compiled C program is shown. Firstly, the model answer is calculated and then the **Subprocess** Python Module is used to execute the student's compiled program while providing input arguments and reading the output from the student's program. The output is then parsed, cleaned up and analysed for grading purposes. The example shows a very basic clean up and parsing of output stream. The stream is split by new lines, any empty terms are removed and the strings are then converted to integers. This parsing process can be far more robust: it can validate the output stream's contents and can be made more resilient to variations in formatting and incorrect data types.

In the next example we will demonstrate a more complicated evaluation script, which will demonstrate passing standard input to the submitted program and inserting diagnostic code into the submitted source code before compilation. Unlike interpreted languages that can be executed directly by the evaluation scripts (and which therefore have direct access to the variables in memory), gaining access to the data stored in variables in compiled programs is more difficult. Generally, when a script runs a compiled executable, the only interface between the script and the running program is via command line arguments passed when the program is executed or using the standard input and output streams. Being able to examine the state of internal variables in the program is not directly possible. However, the evaluation scripts are able to modify the submitted source code before compiling it and executing it. This means that one can programmatically insert diagnostic code into the student's submitted code to aid in the evaluation process.

```
#include <stdio.h>
int main() {
    //Do not change any of the
    //following code!
    srand(time(NULL));
    int data[20],index=0;
    while(index<20) {
        data[index++] = rand()%199+1;
    }

    //Provide your code below
}
```

Code Snippet 5: Starting code provided for second example array question in C language.

```
#! /bin/bash
 . common_script.sh

cat > vpl_execution <<EEOOFF
#! /bin/bash
#Check that the students code compiles
gcc -w main.c -o testProg 2> dummy.out
cat dummy.out
if [ -e testProg ]; then
  echo "No Compile Error"
  #Run script to insert diagnostic code
  python3.4 parseCode.py
  #Compile modified code
  gcc -w newMain.c -o testProg
  #Run evaluation script
  python3.4 customEval.py
fi

EEOOFF
chmod +x vpl_execution
```

Code Snippet 6: Example of vpl_evaluate.sh for a C submission which test compiles the submitted code, runs a python script to insert diagnostic code into the submitted code and then compiles and evaluates the modified submission.

Code Snippet 5 shows the code given to the students for an assignment. In this question the students are provided with the code to generate a 20-element array of randomly generated

integers. The assignment then asks the students to accept as input from the standard input stream an integer between one and nine. They must then write the code to count how many multiples of that number are present in the array and print out the resulting count. Providing the code to generate the array that the student needs to work with gives the student an insight into how the data is structured and stored, and making the data randomly generated means they cannot hardcode a solution to suit a static data set. The evaluation of this problem will present two new challenges. Firstly, when the evaluation Python script executes the compiled submission it must provide the random number via the standard input stream. Secondly, to calculate the correct answer for the current randomly generated data set the evaluation script needs to access the data in the array.

To gain access to the contents of variables in a compiled program we will need to insert some diagnostic code into the submitted code which writes the data to a file which the evaluation script can then open and read in. Code Snippet 6 shows the BASH script for this example. Firstly, the BASH script compiles the unmodified code to confirm there are no errors present. If the code compiles, the BASH script runs **parseCode.py** which is the script that inserts our diagnostic code. Once the code is modified the BASH script compiles the new code and then calls our evaluation Python script (**customEval.py)** to run the result and perform the evaluation.

```
import sys
#Open the students source code
f = open('main.c','r')
#Read it in and split it by newlines
codeMain = str(f.read())
mainLines = codeMain.split('\n');
f.close();
#Parse the code and find the last
#closing brace
lastLine = 0;
for i in range(0,len(mainLines)):
    if(mainLines[i].strip() == '}'):
        lastLine = i
#Insert the code to write the contents
# of the data array to a text file
mainLines.insert(lastLine-1,'FILE
*out;out = fopen("data.txt","w");
int ixx;for(ixx=0;ixx<20;ixx++) {
fprintf(out,"%d\\n",data[ixx]);}
fclose(out);')
#Write the modified source code to a
new file
newMain = open('newMain.c','w')
for l in mainLines:
    newMain.write(l+'\n')
newMain.close()
```

Code Snippet 7: Python script (parseCode.py) that inserts diagnostic code into the submitted code. The code is inserted at the end of the main() function and writes the contents of the **data** array to a file for the evaluation script to examine.

Code Snippet 7 shows the script that inserts the diagnostic code. This script opens the student's submission source code file, reads it in and separates it into lines. It then searches through the code to find the last closing brace which will indicate the very end of the program. The script then inserts the diagnostic code, which opens a new text file and writes all the values data array values into the text file. The code includes a strangely named integer variable which is so named as to make it unlikely to clash with any variables declared by the student. When the modified code is compiled and executed the contents of the data array variable will be written to a text file which the evaluation script can access.

Code Snippet 8 (shown on full page) shows the evaluation script for this example. The first thing this script does is generate a random integer between one and nine to pass to the student's program when it is run. The **Subprocess** Python module is used to run the compiled executable and the **communicate()** function includes the ability to transmit data via the standard input stream as is shown here. The next step is for the script to calculate the model answer for this randomly generated input and the values in the data array. The script opens and reads the contents of the text file generated by the diagnostic code. It then calculates the model answer for this data set. The script then reads in the values output by the submitted program. The script confirms that only one value was output by the program (otherwise the script returns a message to the student and assigns a grade of 0). The script then compares the student's answer to the model answer, and an appropriate message and grade are provided.

The final example presented in this paper is for an Octave assignment. The example will demonstrate how to use a Python script to parse the Octave code to validate certain elements of the submission, insert diagnostic code and communicate these results to the Octave based custom evaluation script.

```
%Provided random matrix, do not tamper\
with this line!
R = round(rand(1,15)*100+1);
%Only modify the values in the data\
array. Do not modify R.
data = R;
```

Code Snippet 9: Starting code provided for third example which is an Octave based sorting question.

The assignment given for this last example consists of provided code that generates an array of random integers, which can be seen in Code Snippet 9. The students are then tasked with writing the code to sort that array in descending order using the "Insertion Sort" algorithm. If the student sorts the array using any other algorithm they will receive half marks. The students are forbidden from using the built in **sort()** function. Code Snippet 10 (shown on full page) shows the Python script used to enforce the conditions of the question and insert the diagnostic code. First the script reads in the submitted source code and checks if the critical line has been modified or not. Next the script looks for any occurrence of a call to the **sort()** function. Finally, the results of these checks are written to **mat** file which can be easily opened in the custom Octave evaluation script.

```
import sys, subprocess, random
#Run the modified source code passing a
#random value via standard input stream
in_val = random.randint(1,9)
p = subprocess.Popen(["./testProg"],\
stdout=subprocess.PIPE,stdin= \
subprocess.PIPE,universal_newlines \
= True)
output = p.communicate\
      (input=str(in_val))[0]
#Read in the array data from text file
data_file = open('data.txt','r');
data = str(data_file.read())
data = data.split('\n')
#Separate out the integer terms
terms = [i for i in data if i != '']
terms= [int(i) for i in terms]
#Calculate the correct answer
multiple_count = 0
for i in range(0,len(terms)):
  if terms[i]%in_val == 0:
    multiple_count = multiple_count + 1
#Print the output from the submission
#for reference by the student
output = output.split('\n')
output = [i for i in \
            output if i != '']
print("Comment :=>> You Printed:");
for n in output:
  print("Comment :=>> " + n)
print("Comment :=>> ---------------")
#Convert input strings to integers
for i in range(0,len(output)):
    output[i] = int(output[i])
#Check only a single number was output
if len(output) > 1:
    print("Comment :=>> Printed too
many numbers....question says to only
print the number of multiples in the
array.");
    print("Grade :=>> 0")
elif len(output) <= 0:
    print("Comment :=>> You didn't
print any numbers.")
    print("Grade :=>> 0")
else:
    if output[0] == multiple_count:
        print("Comment :=>>
Correct!")
        print("Grade :=>> 100")
    else:
        print("Comment :=>> Printed
answer not correct")
        print("Grade :=>> 0")
```

Code Snippet 8: Custom Python evaluation script (customEval.py) that reads in the array data that was written to file by the diagnostic code inserted into the student's submission to calculate the correct answer for grading purposes.

```
import sys
#Open the students source code
fMain = open('main.m','r')
#Read it in and split it by newlines
codeMain = str(fMain.read());
mainLines = codeMain.split('\n')
fMain.close();
#Test to see if the provided code was
#tampered with
codeTamper = 0; flag = 0;
if len(mainLines) > 1:
  if mainLines[1].strip() != "R =
round(rand(1,15)*100+1);":
      codeTamper = 1
#Check if sort() function was used
usedSort = 0;
#Clear out all spaces, tabs, newlines
noSpace = codeMain.replace(" ","");
noSpace = noSpace.replace("\t","");
noSpace = noSpace.replace("\n","")
sortPos = noSpace.find('sort(')
if sortPos >= 0:
  usedSort = 1

#Find the line containing the last END
#statement, which is where we will
#insert the diagnostic code
TooManyEnds = 0; count = 0; index = 0
for i in range(0,len(mainLines)):
  if mainLines[i].strip() == 'end':
        count = count + 1
        index = i

if count!=3:
  TooManyEnds = 1
else:
  #Insert diagnostic code
  mainLines.insert(3,'swapMatrix=[];')
  mainLines.insert(index+1,'swapMatrix
= [swapMatrix; R==data];')

#Write modified code to new file
fMain = open('newMain.m','w')
for l in mainLines:
  fMain.write(l+'\n')
fMain.close()

#Write the parsing flags to file
fOut = open('parse.out','w')
fOut.write(str(codeTamper) + '\n' + \
      str(TooManyEnds) + '\n' +  \
      str(usedSort))
fOut.close()
```

Code Snippet 10: Python script to parse a student's source code submission to check if the provided code was tampered with, insert diagnostic code to allow for characterising what type of sort algorithm was used and to confirm that the student did not use the built-in sort() function as was specified in the question.

```
load parse.out %Load the output of the parseCode.py script
if parse(1) == 1
  printf('Comment :=>> You tampered with the top lines of provided code!\n');
  printf('Grade :=>> %d\n',0);
elseif parse(2) == 1
  printf('Comment :=>> Your code is not in the correct form for an Insertion Sort
algorithm. The algorithm consists of two loops (nested) and an If statement\n');
  printf('Grade :=>> %d\n',0);
elseif parse(3) == 1
  printf('Comment :=>> Do not call the sort() function!\n');
  printf('Grade :=>> %d\n',0);
else
  newMain  %Execute and Evaluate submission
  %Save the current random matrix, rerun student code, load old random matrix
  R_old = R;  save r_old.mat R_old; clear all; newMain; load r_old.mat
  %Students sometimes declare variables with core function names
  clear sum prod floor
  if(isequal(R_old,R))
    printf('Comment :=>> You have changed the code that generates Random Matrix! Do
not change the values generated for the random matrix!\n');
    printf('Grade :=>> %d\n',0);
  else
    %This is the model answer which also builds the swap signature for the current
    %data
    myData = R;
    mySwapMatrix = [];
    for i = 2:length(myData)
      for j = i:-1:2
        if myData(j) > myData(j-1)
          temp = myData(j-1);
          myData(j-1) = myData(j);
          myData(j) = temp;
        else
          break;
        end
      end
      mySwapMatrix = [mySwapMatrix; R==myData];
    end

    grade = 0;
    if(isequal(mySwapMatrix,swapMatrix))
      printf('Comment :=>> Array was correctly sorted using Insertion Sort!\n');
      grade = 100
    elseif(isequal(myData,data))
      printf('Comment :=>> Array was correctly sorted using another form of sort!
Partial marks awarded\n');
      grade = 50
    else
      printf('Comment :=>> Array is not being sorted in descending order using the
Insertion Sort algorithm! This could be because you are sorting in ascending order
OR that you are not using the Insertion sort algorithm\n');
    end
    printf('Grade :=>> %d\n',grade);
```

Code Snippet 11: The Octave based custom evaluation script for the third example. This script reads in the results of the Python script that parsed the submitted code and provides appropriate messages where needed. It then confirms that the data set is indeed random by running it twice and checking that the output is different. It then uses the data set to build a model signature of the sorting process and compares it to the signature produced by the diagnostic code. IF the correct sorting algorithm was used the student receives full marks. If the array is sorted with a different algorithm they will receive half marks.

In order to detect which sorting algorithm was employed some diagnostic code will be inserted at the end of the outer **For** loop. The script counts how many **End** clauses there are in the code; for an insertion sort algorithm there should be exactly three such clauses. The inserted diagnostic code records the changes in the array for each iteration. This provides a signature which is unique to the sorting algorithm used and can be used to determine if the insertion sort was employed. The modified code is written to file to be executed by the custom evaluation script which can be seen in Code Snippet 11 (shown on full page).

The first thing the custom evaluation does is load the file which contains the flags produced by the validation process. If any of the three validation conditions have been detected the appropriate message is given to the student and they receive a zero grade. Next, the script will run the student's submission. In order to confirm that the random data set is indeed random, another phase of validation is performed. The data set is recorded to file. The student's submission is run a second time and the new data set is compared to first data set. If the data set is truly random these arrays' will be different. If they are not, then it indicates that the student has manipulated the contents of the dataset. They receive an appropriate error message and grade. The script then calculates the model sorting signature for the random data set. If the student's submission matches this sorting signature, then they receive full marks. If they sorted the algorithm using a different algorithm but the array is sorted correctly, they will receive half credit. Otherwise they have not met the question specifications and will receive a zero grade. This example demonstrates a number of further validation possibilities and the capability of providing partial credit to students who meet certain sub-objectives of a question.

IV. Application of Learning System to Pedagogical Goals

This section will discuss how the learning system supports the pedagogical philosophy described in Section II. The first major benefit of the system is that it lives in the cloud and is accessible from any decently sized device that can run a modern web browser. This means that students have access to a code editor and the relevant toolchains from any internet connected computer without needing to install any specialized software. This has greatly increased the access to these technologies, especially for students who have limited resources at their disposal.

The system was built to support a constructionist pedagogical philosophy and does so far better than traditional assessment of programming. The automated assessment tools can provide real-time feedback from the compiler/interpreter and built-in clues and guidance from the instructor, meaning that students can work at their own pace to solve programming problems but are given appropriate feedback every step of the way. An instructor can write scripts that detect common mistakes and provide clues on how to address these common errors to the student. In this way, the student can work at their own pace and work towards a solution on their own terms and when they have solved a problem the system will inform them of the fact. This is important because often students are not able to identify when a problem is adequately solved and by having the system automatically inform them they get the satisfaction of having solved a problem on their own without having to be told days later that they were successful through a manually graded assignment.

The system also provides an appealing alternative to traditional partial credit grading. Paper-based assessments are inherently submitted once, and the submitted work must be evaluated as is. With the system presented here, students can be expected to correct small mistakes based on the compiler or instructor feedback and produce (and test) a working program. This expectation of a working solution fosters learning to develop an understanding rather than incentivizing memorization to earn partial credit. Note that the system does not preclude partial credit: an instructor can build assessments that assess levels of completion of a programming task, or allow for common mistakes (as in the sort example). This flexibility caters for all levels of student performance but still ensures that each student is producing at least basic complete programs. The automation also means that it becomes possible to very quickly and consistently assess the level of capability that a student has demonstrated through their performance. If deemed appropriate, grading schemes can also be adapted to include number of incorrect submissions or compilation attempts, code efficiency, submission time, and more; the combination of the course management system and scripting environment is remarkably flexible.

The system also provides a trove of data about student participation and progress, keeping track of almost every action a student takes. This data can definitively answer questions about student participation and progress which might traditionally rely on self-reporting from students. For example, how does the number of revision activities completed relate to the student's performance in a test situation? Fig. 1 shows this data extracted for the first test for the C language outcome. There is a clear trend showing that the more a student has participated and completed the revision activities the better they will perform on the test. This kind of data allows an instructor to seek objective feedback on the teaching strategies they employ.

The final benefit that the authors would like to report is the time saved administering assessments. In the authors experience, grading a traditional paper-based exam for 400 students could take between 30 and 50 man-hours of tedious work (plus additional time to set the test paper). Due to the size of available computer labs, the same 400-student class was assessed in three separate sessions, but even setting up three unique assessments with automated assessment only required around 10 man-hours of work. Additionally, once test questions are prepared on the system they can be reused either as revision questions or assembled into a database from which to randomly draw future assessments.

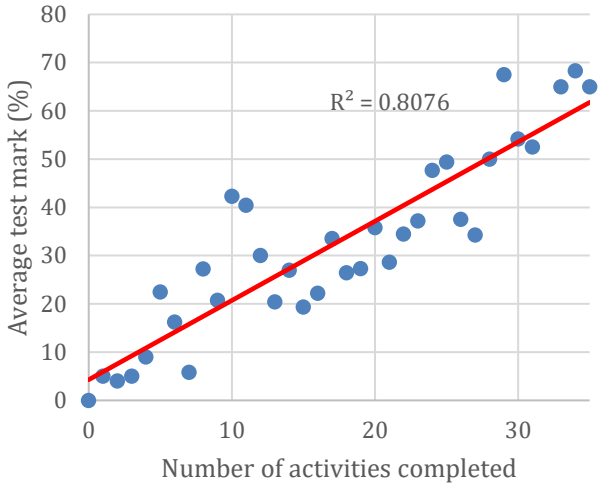

Figure 1: A plot of number of revision problem completed vs the average test mark achieved; a red line indicates the linear regression fit.

V. Conclusion

This article describes the implementation of an online learning system that enables the automated teaching and assessment of computer programming tasks for large classes. The pedagogical philosophy employed by the authors in teaching programming is described and how the implemented system addresses the goals of this philosophy is discussed.

The learning system is implemented using the open source learning platform Moodle and an open source plug-in, VPL, which supports the execution and assessment of a large variety of programming languages. The author describes the VPL system and how the scripting engine used by VPL can be used to build flexible and robust automatically assessed programming activities. A number of new techniques for building these evaluation scripts are presented in detail, allowing future efforts to reproduce and expand upon this work.

The benefits of the system for use in a large class are also discussed. Firstly, the system increases the amount of real-time feedback the students receive which is significantly higher than in the traditional model where a single lecturer and a handful of tutors can only provide limited attention to the large number of students in a class. The students can learn and experiment with the work at their own pace and in their own way. The amount of administration work is significantly reduced for the lecturer and content produced can be reused in many ways. Finally, the system provides a trove of data that can be used to ask interesting questions about the class to aid in self-reflection and course execution.

References

[1] B.S. Bloom, M.D. Engelhart, E.J. Furst, W.H. Hill, D.R. Krathwohl, *Taxonomy of education objectives: The classification of educational goals. Handbook I: Cognitive domain*. New York: David McKay Company, 1956.

[2] S. Papert, "Constructionism: A New Opportunity for Elementary Science Education," Massachusetts Institute of Technology, Media Laboratory, Epistemology and Learning Group: National Science Foundation, Award 8751190, 1986.

[3] J. Sheard, Simon, A. Carbone, D. D'Souza, and M. Hamilton. "Assessment of programming: pedagogical foundations of exams," in *Proc. of the 18th ACM conference on Innovation and Technology in Computer Science Education (ITiCSE '13)*, 2013, pp. 141-146.

[4] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. "Review of recent systems for automatic assessment of programming assignments," in *Proc. of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10)*, 2010, pp. 86-93.

[5] E. Costello, "Opening up to open source: looking at how Moodle was adopted in higher education," in *Open Learning: The Journal of Open, Distance and e-Learning*, Vol. 28, Issue 3, 2013.

[6] (2016) *The GNU Licenses Website.* [Online]. Available: http://www.gnu.org/licenses/licenses.en.html

[7] (2016) *Bitnami Website*. [Online] Available: https://bitnami.com/

[8] (2016) SCORM Website. [Online] Available: http://www.adlnet.gov/adl-research/scorm/

[9] J.C. Rodríguez-del-Pino, R-R. Enrique, and H-F. Zenón, "A Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features," in *Proceedings of the 2012 International Conference on e-Learning, e-Business, Enterprise Information Systems, & e-Government*, 2012.

[10] (2016) *VPL Plug-In website.* [Online] Available: http://vpl.dis.ulpgc.es/

[11] D. Thiébaut, "Automatic evaluation of computer programs using Moodle's virtual programming lab (VPL) plug-in," *Journal of Computing Sciences in Colleges*, Vol. 20, Issue 6, 2015.

[12] J.B. Biggs, C. Tang, Teaching for quality learning at university: What the student does. McGraw-Hill Education (UK), 2011.

[13] T. Staubitz, H. Klement, J. Renz, R. Teusner and C. Meinel, "Towards practical programming exercises and automated assessment in Massive Open Online Courses," Teaching, Assessment, and Learning for Engineering (TALE), 2015 IEEE International Conference on, Zhuhai, 2015, pp. 23-30.