A Testing Tool for Introductory Programming Courses

Thesis A Seminar

Kyu-Sang Kim
z5208931

Supervised by Andrew Taylor (UNSW)
Assessed by John Shepherd (UNSW)

Term 1, 2022

Contents

Student Enrolments in Introductory Courses

Student numbers enrolling in introductory programming courses at UNSW have significantly increased in recent years

Enrolments/Year	**2017**	**2018**	**2019**	**2020**	**2021**
COMP1511	1351	1655	1874	1905	2381
COMP1521	715	1136	1352	1417	1633
COMP2521	378	1019	1389	1445	1551

Table 1: Course enrolments from UNSW Class Timetable (April 2022)

The trend in increasing student enrolments for these courses present two main challenges for course staff:

1. How can courses feasibly start/continue utilising practical assessments without automation tools as the staff to student ratio decreases?
2. How much of the marking process can be automated via tools?
3. Will the automation tools remain viable as assessments change?

Andrew Taylor autotest

1. Written by Andrew Taylor in 2015 (Python rewrite)
2. First introduced to UNSW in COMP2041 2015 S2
3. Sufficiently capable and in current use to perform automated marking for most UNSW introductory programming courses
4. Has some **design deficiencies** and accumulated non-trivial **technical debt** since introduction
5. Deeper exploration of architecture and implementation details in Background section of Seminar

Technical Debt

1. We approach technical debt as outlined in the framework defined within *An Exploration of Technical Debt*[1]
2. Cunningham[2], who introduced the concept of technical debt, described how *"shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite"*
3. How can we reconcile existing technical debt and ensure debt remains manageable in the future?

[1]Tom Edith, Aybüke Aurum and Richard Vidgen, 'An Exploration of Technical Debt' (2013) 86(6) *The Journal of systems and software 1498*

[2]Ward Cunningham, 'The WyCash Portfolio Management System' in *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (addendum)* (ACM, 1992) 29

Approaches to Managing Technical Debt

1. Extensive Project & Code Documentation
2. Project Management Tools - GitHub, Jira etc.
3. Modular Architectural Design
4. Modernisation of Solution - Feature extensibility etc.
5. Automated Regression Testing

Introducing all of these technical debt management solutions to the existing Andrew Taylor autotest software package has been determined to be infeasible due to the extensive refactoring required.

Thus, we propose that a new software package be created from a clean slate to minimise potential sources for both current and future technical debt from the ground up.

Thesis Statement

A user-friendly and maintainable general code testing tool is important to streamline the administration of introductory programming courses

We will:

1. Develop an extensible and easy to use software package which parses and runs pre-written tests on submitted code
2. Implement development procedures that minimise both current and future technical debt
3. Remediate known flaws in the existing autotest package
4. Maintain backwards compatibility with legacy tests written for the existing autotest package
5. Perform proving and performance tests on the new software package
6. Deprecate and replace the existing autotest used for introductory programming courses at UNSW CSE

Contents

Code Marking in Introductory Courses

1. We assume that *code marking* refers to the determination of whether submitted code when run conforms to some behaviour that is outlined in a specification or any similar resource
2. What are some potential issues with manual code marking?[3]
    - Tedious and repetitive work is prone to mistakes by human error
    - Potential inconsistencies when distributed between separate markers
    - Increasing marking load per course staff member due to increasing student numbers can exacerbate the aforementioned issues
3. How can we eliminate or reduce these issues?

[3] Jackson David, 'Using Software Tools to Automate the Assessment of Student Programs' (1991) 17(2) *Computers and education* 133

Automated Testing & Marking Approaches

1. The earliest example of automated testing on student code was published in 1960 by Jack Hollingsworth of the Rensselaer Polytechnic Institute[4]
2. **Main goal:** Autonomously verify the correctness of student submitted code in relation to pre-defined expected behaviour
3. **Main benefit:** Automation of performance/correctness testing of student submitted code
   - Time spent on manual testing by staff can be utilised in other tasks
   - Students were observed to learn programming better with an automatic grader over dedicated lab groups
   - Increasing enrolments for programming courses per teaching period becomes economically feasible

[4]Jack Hollingsworth, ‘Automatic Graders for Programming Classes’ (1960) 3(10) *Communications of the ACM 528*

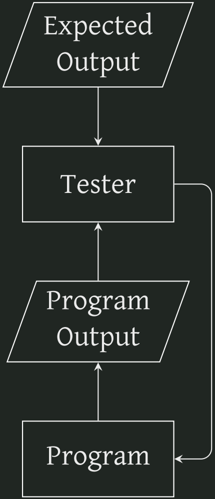
Automated Testing & Marking Approaches

1. How can we methodically determine that *"submitted code matches expected behaviour"*?
   - External Program Side-effect Comparison
   - Internal Program Unit Testing
   - Combination of above[5]

2. What are the specifics of these methodologies and what are some existing automated testing frameworks?

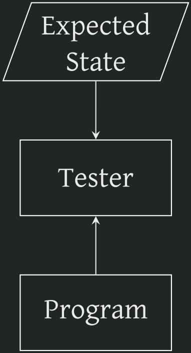
[5] Soundous Zougari, Mariam Tanana and Abdelouahid Lyhyaoui, 'Hybrid Assessment Method for Programming Assignments' in *2016 4th IEEE International Colloquium on Information Science and Technology (CiSt)* (IEEE, 2016) 564

External Program Side-effect Comparison



1. Assume program side-effects to be **observable output** from a tested program which has been stored on external resources such as the console via *stdout* or files that store logs/program state
2. This output can be **externally compared** with those pre-generated by a sample solution to determine whether expected behaviour has been achieved
3. Comparisons on side-effects after program execution allows for easier testing of complex programs with dependencies on multiple different components at the cost of storage space

Internal Program Unit Testing



1. Assume unit testing refers to the testing of **individual discrete code components** within the program by comparing internal program state to a known expected state pre-generated by a sample solution
2. Executing and collecting the results of the internal unit tests for the program can be used to determine whether expected behaviour has been achieved
3. Testing internal code components allow for more thorough determination of program correctness at the cost of difficulty in testing more complex programs which may not be able to share the same unit testing framework

Andrew Taylor autotest

1. **Author(s) & Introduction:** Andrew Taylor (University of New South Wales) - 2015
2. **Testing Methodology:** External Program Side-effect Comparison
3. **Language:** Python (+ some Shell and Perl script interfaces)
4. **Maintained/Support:** Not regularly maintained outside of rare UNSW CSE student engagement
5. Current UNSW CSE automated general code testing tool utilised for lab and assessment marking in most introductory programming courses and at times, some higher level courses (COMP Level 2+)

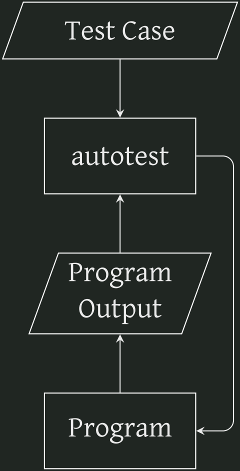
Andrew Taylor autotest implementation details

Listing 1: autotest Example Test Cases

```
files=is_prime.c

1 stdin="39" expected_stdout="39 is not prime\n"
2 stdin="42" expected_stdout="42 is not prime\n"
3 stdin="47" expected_stdout="47 is prime\n"
```

Listing 2: autotest Wrapper

```
#!/bin/sh

parameters="
default_compilers = {'c' : [['clang', '-Werror', '-
      std=gnu11', '-g', '-lm']]}
upload_url = https://example.com/autotest.cgi
"

exec <path_to_autotest>/autotest.py --
      exercise_directory <path_to_exercise_dir> --
      parameters "$parameters" "$@"
```



Andrew Taylor autotest pros & cons

1. **Pros:**
   - Proven to be *mostly* reliable at UNSW CSE
   - Can support any programming language assuming autotest can detect and compare side-effects
   - Extensive parameters exposed to manage test execution environment
   - Test results on failure provide meaningful information on the differences between actual and expected output
2. **Cons:**
   - High levels of technical debt due to outdated use of technology and architecture
   - Significant lack of documentation
   - Difficult for first-time users to create tests and manage test execution environment

Harvard CS50 check50

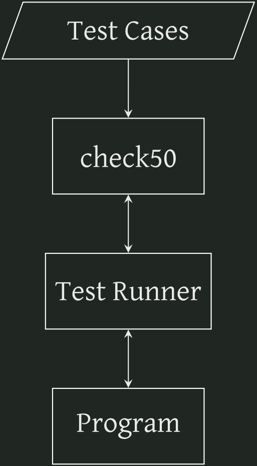
1. **Author(s) & Introduction:** Chad Sharp (Harvard University) - 2012[6]
2. **Testing Methodology:** External Program Side-effect Comparison
3. **Language:** Python
4. **Maintained/Support:** Regularly maintained with active development (in private repository)
5. Current Harvard University CS50: Introduction to Computer Science automated general code testing tool for checking correctness of practical lab exercises

[6] Chad Sharp et al, 'An Open-Source, API-Based Framework for Assessing the Correctness of Code in CS50' in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (ACM, 2020) 487

Harvard CS50 check50 Implementation Details

Listing 3: check50 tests

```
import check50
import check50.c

@check50.check()
def exists():
"""hello.c exists"""
check50.exists("hello.c")

@check50.check(exists)
def compiles():
"""hello.c compiles"""
check50.c.compile("hello.c", lcs50=True)

@check50.check(compiles)
def emma():
"""responds to name Emma"""
check50.run("./hello").stdin("Emma").stdout("Emma").
     exit()
```



Harvard CS50 check50 Pros & Cons

1. **Pros:**
   - Tests are very simple to create as a chain of functions
   - Documentation is very extensive
   - Testing results can be rendered to HTML via a module for easier viewing
   - Supports running of tests on both local and remote machines (PaaS Support)
   - Concurrent running of tests is supported
2. **Cons:**
   - As a result of design choice for simplicity, testing of complex programs can be challenging (may require Harnessing)
   - No official support or implementations for programming languages outside of C and Python (Flask Supported)

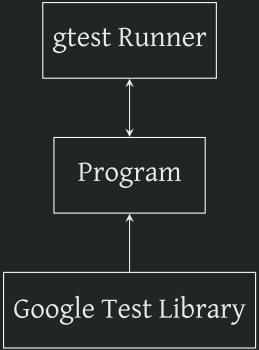
Google gtest

1. **Author(s) & Introduction:** Google - v1.0.0 released in 2008
2. **Testing Methodology:** Internal Program Unit Testing (Technically Supports External Program Side-effect Comparison)
3. **Language:** C++
4. **Maintained/Support:** Regularly maintained by Google and Open Source Community
5. Originally created by Google for internal use but has become one of the most popular C++ unit testing frameworks within the xUnit family of testing frameworks

Google gtest Implementation Details

Listing 4: gtest test cases

```
TEST(FactorialTest, Positive) {
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(2, Factorial(2));
    EXPECT_EQ(6, Factorial(3));
    EXPECT_EQ(40320, Factorial(8));
}

TEST(IsPrimeTest, Positive) {
    EXPECT_FALSE(IsPrime(4));
    EXPECT_TRUE(IsPrime(5));
    EXPECT_FALSE(IsPrime(6));
    EXPECT_TRUE(IsPrime(23));
}
```



Google gtest Pros & Cons

1. **Pros:**
   - Documentation is extensive with a large community
   - Concurrent execution of tests is supported
   - Performance benefits of C++
2. **Cons:**
   - Testing of complex systems can be difficult as per standard with Internal Program Unit Testing methodology
   - Setting up test execution environment is not common for most users and can be difficult to configure based on testing needs

JUnit

1. **Author(s) & Introduction:** Kent Beck, Erich Gamma, David Saff, Kris Vasudevan - Initial Prototype in 1997[7]
2. **Testing Methodology:** Internal Program Unit Testing (Technically Supports External Program Side-effect Comparison)
3. **Language:** Java
4. **Maintained/Support:** Regularly maintained by JUnit team and Open Source Community
5. A very popular open source unit testing framework for Java applications within the xUnit family of testing frameworks

[7]Martin Fowler, "Bliki: Xunit", *martinfowler.com* (Webpage, 2022) <https://martinfowler.com/bliki/Xunit.html>

BAGS - Basser Automatic Grading Scheme

1. **Author(s) & Introduction:** J Hext and J Winings (University of Sydney) - Earliest Documented in 1968[8]
2. **Testing Methodology:** Primitive External Program Side-effect Comparison
3. **Language:** Unknown (one of the KDF9 Operating System languages)
4. **Maintained/Support:** Unknown (Assumed to be abandoned due to age)

[8]J Hext and J Winings, ‘An Automatic Grading Scheme for Simple Programming Exercises’ (1969) 12(5) *Communications of the ACM* 272

Kassandra

1. **Author(s) & Introduction:** Urs von Matt (ETH Zürich) - Earliest Documented in Winter term 1992/1993[9]
2. **Testing Methodology:** External Program Side-effect Comparison
3. **Language:** Bash Shell & MatLab & Maple
4. **Maintained/Support:** Unknown (Assumed to be abandoned due to age)

[9]Urs von Matt, ‘Kassandra’ (1994) 22(1) *SIGCUE bulletin* 26>

TRY

1. **Author(s) & Introduction:** Kenneth A Reek (Rochester Institute of Technlogy) - 1989 at Earliest[10]
2. **Testing Methodology:** External Program Side-effect Comparison
3. **Language:** C
4. **Maintained/Support:** Unknown (Assumed to be abandoned due to age)

[10] Kenneth A Reek, 'The TRY System -or- How to Avoid Testing Student Programs' (1989) 21(1) *ACM SIGCSE Bulletin* 112

Contents

Design Requirements

The design requirements for the solution will emphasise the following properties in accordance with the thesis goals:

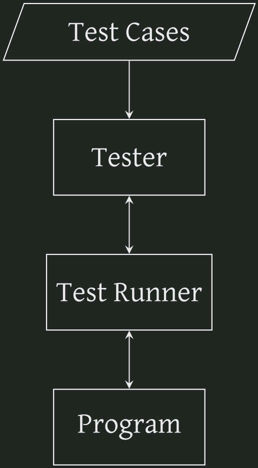
1. *Accessible* - Users of the new software package should have the easiest possible experience in integrating automatic testing and grading into their courses
2. *Familiar* - Users who have previously utilised Andrew Taylor autotest should not feel the new software package to be completely independent of the former
3. *Performant/Efficient* - The new software package should have the same, if not better performance than the original Andrew Taylor autotest. Benchmarking will be performed to verify this property
4. *Maintainable* - The new software package should be adequately documented and support ease of maintainability and possible extension of features via the appropriate architectural decisions

Exclusions

1. *Security* - We assume security to be out of scope as this is not a consideration in the existing Andrew Taylor autotest software package
2. Security considerations will greatly increase the complexity of the software and delivery of the software package before the conclusion of Thesis C is likely to be infeasible
3. *Novelty* - We assume novelty to be out of scope as introducing a novel user experience is in conflict with the aforementioned design requirements
4. Users of the existing Andrew Taylor autotest may elect to not utilise the new software package if the transition is less than convenient

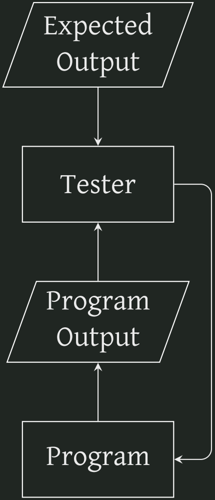
Approach 1

1. Architectural approach similar to Harvard University check50
2. check50 can be utilised as a “baseline” for both performance and correctness testing of the solution
3. Future Work and other relevant sections from the check50 paper can be utilised to inform implementation decisions for the solution
4. **Main methodology to determine correctness of tested program:** External Program Side-effect Comparison



Approach 2

1. Architectural approach similar to existing Andrew Taylor autotest which has been widely accepted for introductory programming courses at UNSW
2. autotest can be utilised as a “baseline” for both performance and correctness testing of the solution
3. **Main methodology to determine correctness of tested program:** External Program Side-effect Comparison



Chosen Approach

1. We select **Approach 2** over the alternative as it more compatible with the outlined design requirements
2. We note that solutions implementing Approach 1 are more feature rich than Approach 2 but it must be considered that uptake of the solution is of higher importance than the features it provides
3. Features present from alternative approaches can also be added or removed in the interest of time but Approach 2 will provide a good starting point
4. Regardless of approach, we will implement all aforementioned technical debt management procedures

Contents

Schedule

1. Rest of thesis A:
   - Continue inspection of Andrew Taylor autotest implementation
   - Complete Draft Design Document
   - Collect feedback on Design Document and make adjustments as necessary
2. Thesis B:
   - Implement Core Main Module
   - Implement Core Testcase Parser Module
   - Implement Core Testcase Runner Module
   - Run correctness and performance testing on Parser and Runner
3. Thesis C:
   - Implement Core Testcase Program Correctness Module
   - Implement any extensions that have been deemed necessary by the Design Document
   - Run correctness and performance testing on complete package and make final adjustments

Summary

We have covered:

- Student enrolments in Introductory Programming courses at UNSW, Andrew Taylor autotest, Technical Debt
- The thesis statement
- Code Marking in Introductory Courses
- Approaches to Automated testing and marking
- Existing and historic solutions for Automated testing and marking, deeper look into the most relevant solutions
- Design requirements and exclusions
- Approaches to the architecture of the solution
- Development schedule of the solution over all thesis periods

Thank you for attending! *Questions?*