УДК 004.054

# Преподавание основ автоматического тестирования программного обеспечения с использованием Appium и симулятора футбола

# Teaching Automated Software Testing with Appium and Soccer Simulator

Хаустов В.А., Мозговой М.В.

Khaustov V., Mozgovoy M.

Khaustov Victor Aleksandrovich, graduate student, University of Aizu, Aizu-Wakamatsu, Japan email: m5201150@u-aizu.ac.jp

Mozgovoy Maxim Vladimirovich, Associate Professor, University of Aizu, Aizu-Wakamatsu, Japan, email: mozgovoy@u-aizu.ac.jp

Технологии автоматизированного тестирования широко применяются в современной разработке программного обеспечения, но редко изучаются в университетах. В статье обсуждается наша попытка исправить ситуацию путём внедрения разработки обязательных автоматических тестов для исследовательского проекта, создаваемого студентами в рамках дипломных работ. Мы полагаем, что автоматическое тестирование следует применять к системам, которые мы в действительности разрабатываем и поддерживаем в ежедневном режиме, и нам следует полагаться на современные технические методы, чтобы обеспечить высокую учебную ценность этого начинания. По итогам экспериментов мы заключаем, что автоматизированное тестирование несёт множество потенциальных преимуществ для студенческих проектов, не при этом не вызывая осложнений, которые часто встречаются в коммерческих компаниях.

The technologies of automated software testing are widely used in modern software development but rarely taught at the universities. We discuss our attempt to remedy the situation by introducing compulsory automated tests for the research project being developed by the students for their graduation works. We argue that testing should be applied to the systems we actually improve and maintain on the daily basis, and we should rely on the state-of-the-art methods to ensure high educational value of this endeavor. As a result of experiments, we conclude that automated testing has many potential benefits for student projects, introducing almost no perceivable burdens that are often present in commercial companies.

**Ключевые слова**: автоматизирование тестирование ПО, Appium, дымовое тестирование.

**Keywords**: automated software testing, Appium, smoke testing.

## Introduction

Automated testing is an integral element of modern software development pipeline, frequently discussed in the literature [7]. The combination of automated tests with manual quality assurance procedures is one of the central tenets of established software development methodologies, such as TDD [1] and BDD [2]. A particular emphasis is usually made on automating small-scale *unit tests*.

In practice, however, maintaining an adequate set of tests can be a challenging and time-consuming task: surveys show that most professional developers are not satisfied with their current testing suites or do no automatic testing at all, complaining that the tests are difficult to write and maintain [4]. A pragmatic approach to testing suggests prioritizing testing strategies and keeping at least the most useful tests well maintained. A number of authors suggest giving the priority to *smoke tests* that check basic functions of the whole software system. Humble and Farley [7] believe that *"smoke test, or deployment test, is probably the most important test to write"*; Mustafa et al. [10] advice to *"stick to smoke testing"* in case of severe time and cost pressure; MSDN documentation calls smoke testing *"the most cost-effective method for identifying and fixing defects in software"* after code reviews [12].

Automated software testing, however, is rarely offered as a separate university course: the students have to master the relevant skills within software engineering studies or during internship. The aim of this paper is to describe our attempt to integrate automated software testing (mainly smoke testing) into everyday activities of our laboratory. We discuss how our students use established smoke testing software tools in a practical mid-scale research project, and how these practices improve student understanding of the topic and motivate them to strive for high quality of the software they develop.

## Testing Strategies for Soccer Simulator

One challenge of teaching automated software testing as a university subject is to find a software project to be used in the course. The students need to know it thoroughly in order to write reasonable test scripts, so in most cases it is assumed that the project has to be implemented during the same course. Another popular option is to test a student project developed as a hobby project or an assignment for a different course. In both cases, such projects are typically immature and thus cannot help to master the whole range of concepts and approaches to software testing.

In our lab, we are working on several projects dedicated to the development of human-like artificial intelligence (AI) systems for computer game worlds. One of them, currently known under a working title of *Soccer Project*, is being designed as a testbed for our team-based AI system that learns from human actions and tries to reproduce the respective person's play style [9]. Soccer Project consists of numerous independent tools, assisting various stages of this process. The central instrument of this set is the game engine named *Soccer Simulator* (see Figure 1). Its purpose is to simulate a game between two soccer teams on a two-dimensional pitch. Each team consists of five or eleven players. Each player can be controlled by a human (via keyboard and mouse or via a remote client), by a simple script-based AI system, or by the AI system we are designing.

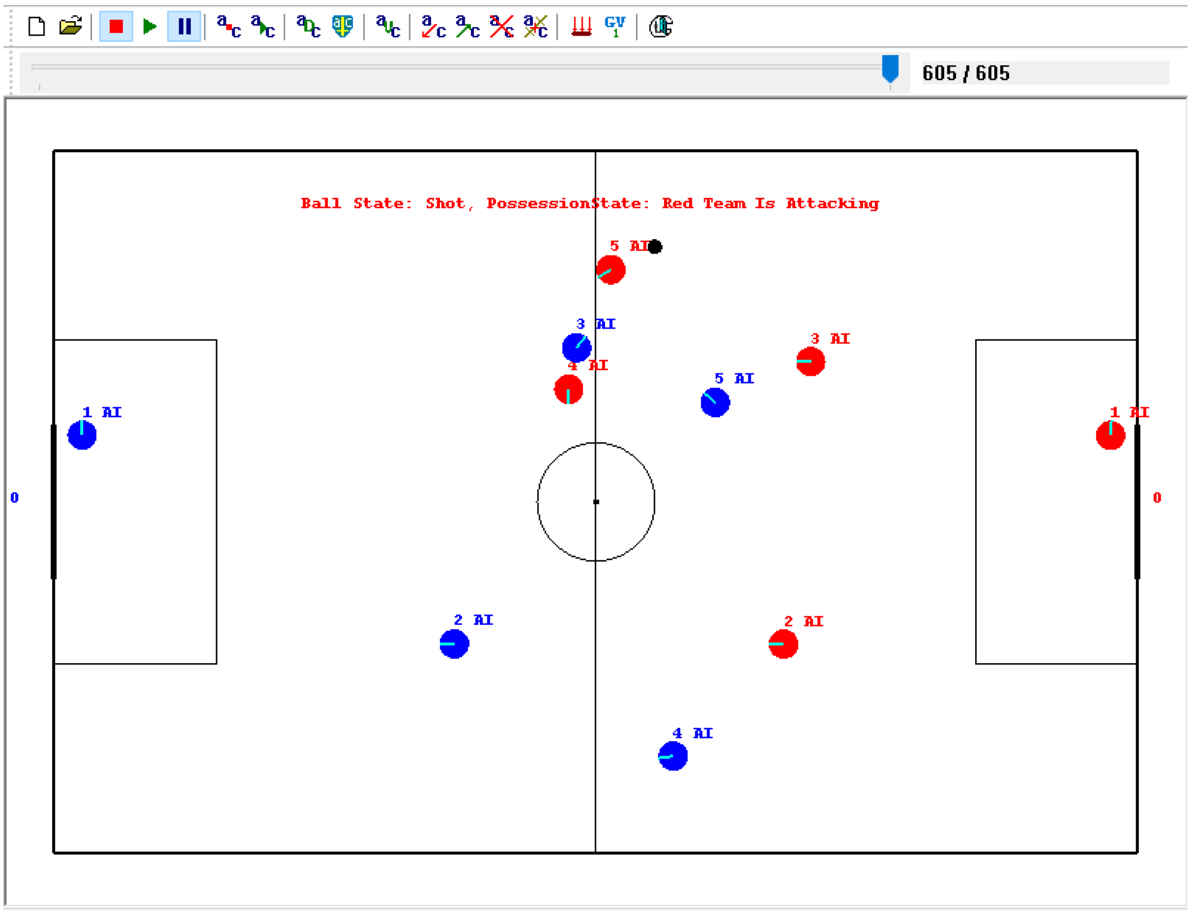

Figure 1: Soccer Simulator (main window)

We use Soccer Simulator in a variety of scenarios. First and foremost, we play soccer games to train the AI system, and let it play against a scripted-based AI to evaluate its skills. Certain experiments require repeated unattended runs of Soccer Simulator with different AI settings to find the optimal configuration or to collect various statistical data that helps to analyze the patterns of player behavior.

Currently, there are more than ten people directly involved in this project. Most of them are working on relatively independent elements of the system, and are solely responsible for the quality of code they commit to the central project repository. We rely on a continuous integration pipeline, backed by an automated build server that performs a full compilation of the project triggered by any changes in the repository. This ensures early problem detection since the project leader and the last contributor are immediately notified about build failures.

Selected modules and tools within Soccer Project are also covered with unit tests [1]. They essentially represent a part of our code base, and thus are being compiled and run during automated project builds. Unit tests are thus easy to integrate into the delivery pipeline, as they do not require any specialized tools, and can be implemented with popular and widely available libraries, such as xUnit family [5].

We encourage the students to write unit tests, but high coverage is difficult to achieve. Therefore, as suggested by Mustafa et al. [10], we put the main emphasis on smoke testing. Furthermore, smoke testing requires writing specialized test scripts and using third-party automation frameworks, and thus provides additional benefits for the learners: they have to study additional practical tools and techniques, which will help them in their future careers of software engineers.

## Smoke Testing with Appium

While a smoke test can be as simple as *“launching the application and checking to make sure that the main screen comes up with the expected content”* [7], it can also evolve into a complex suite of tests checking core application functionality. In case of Soccer Simulator, we have to test, at least, the basic sequence of actions, typically performed by the users:

1) Run the simulator, and start a new game in “player with the ball” mode (in this mode only the player currently possessing the ball is controlled by a human via keyboard and mouse, while all other players are controlled by a scripted AI system).
2) Play for at least one minute, steering the player in random directions.
3) Save the obtained game recording.
4) Use the recording to train the custom AI agent, and save its knowledge base.
5) Run the game in “AI-supported player with the ball” mode to test the behavior of the resulting AI agent, controlling the player possessing the ball instead of a human.
6) Load the agent’s knowledge base into the system.
7) Play for at least one minute to make sure the program does not crash.

When the students design their own experiments, they have to engineer other tests scripts that might include multi-agent behavior, statistical data collection, stress testing (running the software for longer time periods to ensure its stability), human/agent control switching, etc. This way, the students have to test the software they use and improve on a daily basis in their research work.

To run such test scripts, one can rely on the underlying operating system’s API to simulate user actions or use a specialized *application automation framework*. We follow the latter approach, since it represents a universal cross-platform solution, applicable in diverse domains, and thus can be of higher value for the future career of students.

Currently, we rely on a popular Appium automation framework [11]. Appium possesses a number of attractive features that make it a good choice for a university environment:

1) The support of several platforms, including mobile operating systems.
2) Extensive documentation and community support.
3) Open source code base and non-restrictive Apache license.

4) The support of test scripts, written in different languages, including Python, Java, Ruby, and C#).

Appium test scripts interact with the program almost in the same way as users do: they can press buttons, select checkboxes or radio buttons, insert text strings into edit boxes, inspect the content of labels and proceed with mouse clicking on arbitrary areas. Appium is implemented as a client-server system, where the Appium server is responsible for running the program under testing, while a client has to connect to the server and run the test scripts. This architecture allows us to set up a single server machine, and to design and improve the scripts on any remote computer. Appium scripts can also take screenshots of applications and keep action logs, which helps to find errors in test scripts and analyze the sequence of actions that cause test failures.

Such tests scripts can be, in principle, integrated into the continuous integration pipeline. We have already done it for a related mobile tennis game project [8], and are planning to do it for soccer. Currently, the tests have to be run by the students, which also has certain advantages for the educational process, as they can examine and control all steps of the testing.

## Discussion

University provides, probably, one of the most friendly environments to introduce automated testing into software development process. In commercial companies, testing is often discussed in terms of cost-benefit analysis, since the management needs to justify the costs, inevitably caused by additional processes and people involved [6]. In the universities, the primary goal of our initiatives is to educate students, thus any practical methodology, applicable to real-life scenarios, can be used with most ongoing research projects. In our case, the use of smoke testing is entirely justifiable, since the soccer AI we are developing can indeed be tested with relatively simple scripts, and exclude at least the most salient potential bugs in the code.

An important aspect of automated testing is the “safety net” feeling, often discussed in the literature [3]. Being inexperienced developers, students are often afraid to introduce modifications into a relatively large and complicated code base, fearing that their change might break some existing functionality. Automated tests mitigate this problem, providing a simple way of checking the most crucial use cases. On the other hand, tests enforce people to pay due attention to the quality of code they produce. Unfortunately, the students occasionally introduce the changes that break the build or make the system fail even on simple tests. The presence of automated build and test tools help us to reveal and resolve such issues quickly and efficiently.

Writing automated test scripts helps to learn modern tools and technologies, and can itself be a fun process. Not all use scenarios are equally easy to automate, so writing a test to check a certain functionality can be a challenging assignment. We found the results of our experiments with automated testing in the lab encouraging and can recommend to extend the use of modern software development pipeline elements in the organization of student projects.

## References

1. K. Beck. Test-Driven Development by Example, Addison-Wesley Professional, 2002, 240 p.
2. S. Bellware. Behavior-Driven Development. Code Magazine, 2008, vol. 9(3).
3. L. Crispin, J. Gregory. Agile Testing: A Practical Guide for Testers and Agile Teams. Addison-Wesley Professional, 2008, 576 p.
4. E. Daka, G. Fraser. A Survey on Unit Testing Practices and Problems. 25th IEEE International Symposium on Software Reliability Engineering (ISSRE), 2014, pp. 201-211.
5. Hamill P. Unit Test Frameworks. O'Reilly Media, 2004, 304 p.
6. D. Hoffman. Cost Benefits Analysis of Test Automation. STARWEST Software Testing Conference, 1999.
7. J. Humble, D. Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional, 2010, 512 p.

8. M. Mozgovoy, E. Pyshkin. Unity Application Testing Automation with Appium and Image Recognition. *Communications in Computer and Information Science*, 2017, vol. 779, *in press*.
9. M. Mozgovoy, I. Umarov. Believable Team Behavior: Towards Behavior Capture AI for the Game of Soccer. Proceedings of the 8th International Conference on Complex Systems, Boston, USA, 2011, pp. 1554-1564.
10. G. Mustafa, A. Shah, K. Asif, A. Ali. A Strategy for Testing of Web Based Software. Information Technology Journal, 6(1), 2007, pp. 74-81.
11. N. Verma. Mobile Test Automation with Appium. Packt Publishing, 2017, 231 p.
12. Microsoft Corp. Guidelines for Smoke Testing. MSDN Library for Visual Studio 2008.
    URL: https://msdn.microsoft.com/en-us/library/ms182613(v=vs.90).aspx