The PrimeGame Revolutions

A cloud-based collaborative environment for teaching introductory programming

Jens Dietrich
Massey University
Palmerston North,
New Zealand
j.b.dietrich@massey.ac.nz

Johannes Tandler
Technical University
of Dresden
Dresden, Germany
johannes@jtdl.de

Li Sui
Massey University
Palmerston North,
New Zealand
leesui0207@gmail.com

Manfred Meyer
Westphalian University of
Applied Sciences
Bocholt, Germany
manfred.meyer@w-hs.de

***Abstract*—The PrimeGame is an established mathematical programming game that has been used successfully in undergraduate computer science teaching since 2003. To meet the increasing demand for innovative programming tools in undergraduate tertiary and secondary education, we have created SoGaCo, a novel platform to deliver the PrimeGame and similar games to a wide audience via standard web browsers. SoGaCo is designed to have a very low total cost of ownership. This is achieved by enabling teachers to provision a customised collaborative development environment on commodity cloud computing infrastructure. Amongst the unique features of the platform are its social networking features and support for polyglot programming.**

In this paper, we describe the requirements for this system, its design and implementation. We focus on how the scalability and security challenges of an open web-based development environment are addressed. This includes a discussion of the sandboxing and verification techniques we have developed in order to safeguard server-side code execution on the Java Virtual Machine.

I. Introduction

There are widely acknowledged shortcomings in computer science education that is often seen as unable to satisfy the needs of employers, both in terms of the quality and the quantity of the graduates produced. The use of gamification seems to be a viable approach to improve the experience students have, and can improve student engagement and educational outcomes [13], [15], [18].

On the other hand, educators face additional challenges: budgets to acquire new systems are tied, and there is a general expectation that programming tools can be made available for free or at a very low cost. Moreover, teachers at highschool level have very limited technical support and often no access to servers to deploy custom educational packages. This leads in many cases to a preference for cloud-based systems that have a low overall total costs of ownership.

In this paper, we describe our experience in building such a platform. We focus on the technical problems we had to solve to achieve a sufficient level of usability, stability, security and scalability. The system described is SoGaCo (social gaming and coding). SoGaCo is a platform that supports content modules build around simple mathematical board games. Students write bots that play those games on their behalf, and share those bots to play against peers or assessment / benchmarking bots.

While we have implemented several different games with SoGaCo[1], our discussion focuses on our experience with one particular game, the PrimeGame [12], [11]. We chose the PrimeGame for the following reasons: (1) The PrimeGame has an extremely simple programming model that makes it suitable for entry level programming for both highschool and first year university level teaching. (2) There is a simple hierarchy of possible strategies with increasing complexity and increasing bot strength. This increase requires students to explore new features of the programming language used (conditionals, loops, container data structures). (3) The PrimeGame has been used successfully in tertiary education before.

The rest of the paper is organised as follows: we review related work first, followed by a detailed discussion of the design, security and scalability aspects and a brief presentation of the game user interface. A public installation of the game is available at http://sogaco.massey.ac.nz/.

II. Related Work

Several authors have reported the positive impact gamification can have on educational outcomes [13], [15], [18]. The PrimeGame was first used in 2003 at the University of Applied Sciences Gelsenkirchen and since 2009 at the Polytechnic of Namibia and some other institutions in Germany [12]. Since then, successful annual and bi-annual competitions took place, with good participation rates, although the overall success-rate depended on the cultural background of students and required careful balancing between the competitive and collaborative aspects of the game. In [11], the authors presented a version of the PrimeGame based on a client-server architecture but using a proprietary Java client.

There is large body of research on educational programming environments. For space reasons, we only discuss the projects that have directly inspired our work. Greenfoot [8] is a desktop-based development environment where students produce animations by programming actors in Java. Greenfoot can be seen as a turn-based one-player game environment. It supports some social interaction via an online gallery. GreenFoot's focus is on teaching OOP concepts, following

[1] PrimeGame, Mancala, Othello, 5 in a Row

the "objects first" philosophy. We have used GreenFoot successfully for years in a 200-level object-oriented programming course at Massey University. Robocode [14] is a turn-based multiplayer game. While robocode is client-based, it supports collaboration through battle simulators that can load robots developed by other players. Scratch [16] is an online environment for creating simple animations and games. It uses its own visual language, and targets a younger audience. For a more detailed comparison between scratch, greenfoot and the related alice environment, the reader is referred to [17].

There are a number of web-based development environments that have inspired our work, in particular the cloud9 IDE[2] and the Python Tutor [6].

III. Design

A. Requirements

The design of SoGaCo is based on the following requirements elicited from interviews with teachers.

REQ1 Programming Language Support. There have been significant changes in the use of programming languages for highschool and entry-level undergraduate computer science courses in recent years, with Python and Java being the most widely used languages at the moment [7]. It is very likely that there will be more changes. Therefore, our platform is to support multiple languages.

REQ2 Cloud deployability. Cloud-based deployment is necessary to build and deliver a platform that provides low total cost of ownership for users. In particular, with this approach on-site installation can be avoided.

REQ3 Security. Web-based deployment means that code is submitted to the server for compilation and execution. This makes the system extremely vulnerable to injection attacks, including "unintentional" attacks caused by careless programming. This must be addressed.

REQ4 Multiple Games. It cannot be expected that a single game can satisfy the needs of different courses and student cohorts with diverse educational and cultural backgrounds. Therefore, the platform must be separated from the actual content (the games).

REQ5 Authentication. Whenever possible, providers should not have the responsibility to manage user account information. Instead, existing user accounts (in-house or social) should be re-used.

B. Abstractions

The design of the system has to address the requirements discussed above. At the core of the design are some abstractions that describe services or subsystems. While implementations of these abstractions are necessary to deploy an actual instance of SoGaCo, there is a significant amount of freedom how these abstractions are provided.

One of the main design decisions was to use the Java Virtual Machine (JVM) [9] as a platform. The JVM itself provides core abstractions, for instance the ability to execute code written in different languages (REQ1), including explicit support for Java and Python. The JVM with its ecosystem of high quality low cost application servers is also a good platform for cloud-based deployment (REQ2), and libraries like Shiro provide the abstractions for third-party in-house (LDAP) and social (OpenAuth) authentication (REQ5).

C. Services

The system is designed around a set of HTTP services following the REST approach [3]. Data transferred between client and server is generally JSON-encoded. State is avoided whenever possible, however, HTTP sessions are used to track user information such as authentication status. This is imposed by the Shiro framework[3] used to provide authentication services. This design supports the implementation of alternative clients, such as mobile applications. Core services are listed in Table I.

TABLE I
Core Services

URL Pattern	Method	Description
`/bots/`	POST	build and save a new bot
`/bots/<botId>`	PUT	build and save an existing bot
`/delete/<botId>`	DELETE	delete a bot
`/bots-src/<botId>`	GET	fetch the source code of a bot
`/bot-metadata/<botId>`	GET	fetch the meta data for a bot
`/userbots/<userId>`	GET	get the bot ids of the bots owned by a user
`/creategame_b2b`	POST	create a bot against bot game
`/games/<gameId>`	GET	fetch a recorded game
`/template/<language>`	GET	get the bot code template for the respective language

D. Build Process

At the core of SoGaCo is a builder. The purpose of the builder is to turn source code into bot objects that can participate in games, while enforcing a strict security regime and giving detailed feedback to users when the build process fails. Providing support for multiple languages while addressing the unique security issues of cloud-based systems is non-trivial. Java code can be compiled programmatically through the Java Compiler interface `javax.tools.JavaCompiler`. However, code written in many other languages can only be interpreted, either via proprietary interfaces or via the JSR223 [5] standard interface. The build process we devised consists of a number of operations that are performed sequentially.

For compiled languages like Java, the following steps are performed.

COMPILE Source code is compiled into byte code using the respective language compiler.

BYTE_CHECK The generated byte code is checked for violations of security rules. Checks are performed on a byte code model generated with ASM [2].

[2]https://c9.io/

[3]http://shiro.apache.org/

BYTE_INSTR The generated byte code is instrumented. Code to monitor bots for resource usage, to react to timeout requests and to generate trace statements is injected.

LOADING The byte code is used to load a Java class.

INSTANTIATION The class is instantiated.

TESTING A JUnit [1] acceptance test suite is executed to check the semantics and the quality of service characteristics of the bot. In particular, the correctness of generated game moves (post conditions) and resource allocation (timeouts, memory usage) are tested. A custom JUnit runner that allows to inject the object under test is used.

For interpreted languages like Python, the following steps are performed:

SRC_CHECK Source code is checked for violations of security rules. Usually, source code is parsed into an abstract syntax tree to perform this step.

SRC_INSTR Source code is instrumented. Code to monitor bots for resource usage, enforce timeouts and generate trace statements is injected. Usually, source code is parsed into an abstract syntax tree to perform this step.

INSTANTIATION A Java wrapper object is created. Internally, this object uses the interpreter and the submitted source code to interpret method invocations during game play.

TESTING A JUnit acceptance test suite is executed to check the bot. See above for details.

IV. Security

As mentioned before, the security of any system that allows the execution of code submitted from an open web-client on the server is critical. We briefly describe the main challenges, and how they are addressed.

A. Usage of critical system functionality

Bots should not have access to parts of the platform API that would allow them to create or manipulate critical resources. In particular, this includes access to I/O classes, `java.lang.System` and the threading API. Access to the reflection API is also prohibited as this could be used to bypass verification. This is enforced via API white lists during the static analysis steps in the build process. If access to APIs not on the white list is detected, a descriptive error is sent back to the client. On the client, a marker pointing to the respective source code and displaying the error message is generated.

This approach allows us to detect problems early at build time and communicate them back to the user. This is an inherent advantage over the existing class loader / security manager-based sandboxing techniques built into Java and used for instance in the applet API. Here, security violations will only be detected late, at runtime.

B. Responsiveness to Timeouts

For performance as well as for security reasons, bots are executed in (pooled) threads using the Java executor framework. This allows some thread-based sandboxing. In particularly, long-running tasks can be interrupted, and therefore certain types of denial-of-service attacks can be prevented.

When a task is submitted, a timeout can be set that will interrupt a task that runs too long. However, since the Java threading model is collaborative [4], a task cannot be directly terminated. Instead, the collaboration of a task is required to self-terminate. Since students cannot be expected to support this in their code by frequently consulting the interrupted state and terminating computation when it is set to true, the code for these checks must be transparently injected. This is done during the instrumentation phase of the build process.

C. Memory Allocation Quota

The injection-based monitoring described above can also be used to enforce memory allocation limits. For each method invocation in bot code, a call to a monitoring interface is injected. With this monitor, method invocation quota can be enforced, and therefore stack overflow errors can be prevented. We can also control the heap memory allocated from the current thread at this point [4], and force the bots to terminate if a certain quota is exceeded.

D. Security for Python User Code

Unfortunately, the approach described in sections IV-B and IV-C can not be easily ported to Jython (i.e., Python on the JVM). For Jython, we have used the built-in Jython debugger utility to wrap and execute Python code. The debugger can then make the equivalent calls to the monitoring interface that checks the interrupted state and the heap memory allocation.

V. Scalability

A. Cache Design

The sophisticated build process with several additional verification steps has a significant impact on performance. In particular, when a compiled language like Java is used, new classes are created and loaded at the end of each successful build. This process has to be carefully managed in order to avoid memory leaks[5]. But even when memory leaks are avoided, garbage collecting classloaders is potentially slow, depending on the JVM being used.

While caching provides a possible solution, in particular with respective to frequently used built-in "benchmarking bots", caching bots directly is not possible or practical for a number of reasons. The same bot might participate in multiple games running in multiple threads at the same time. This becomes a problem if bots have state. For educational reasons, we do not want to restrict the use of fields in bots. But even for stateless bots there is a problem, as we add state through the instrumentation and the use of controllers. While the use of `ThreadLocal` could provide a solution to the problem, we opted to use unique `BotFactories` for each bot that is being built. These factories consist of class / class loader

[4]Note that this requires the presence of the `com.sun.management.ThreadMXBean` MBean that is available in Oracles JDK, but not in the OpenJDK as of version 8u40

[5]Manifested in `java.lang.OutOfMemory: PermGen` errors)

combination that can be safely cached, and unique short-lived bot instances can be created for each game that is being played. Therefore, all bots are automatically thread-local. The only drawback is that "static state" (static fields in Java) cannot be permitted.

While this works well for compiled languages, a different strategy must be chosen for interpreted languages like Python. In this case, we use the pre-compilation mechanism that is part of JSR223 [5] to create cachable artefacts.

B. Cache Implementation

The cache is implemented as a concurrent map with a maximum size and automatic eviction of entries after a configurable maximum lifetime. For this purpose, the cache utility class from the Google guava library[6] is used.

C. Experimental Validation

To assess the impact of instrumentation and caching on performance, we conducted a number of experiments. For this purpose, we created five bots implementing the game strategies discussed in [12]. First, we build all bots 1000 times to establish whether instrumentation has an impact on build time. We found that this is not the case, instrumentation adds less than 10% build time overhead. Then we instrumented the bots and execute 500 games of the bots playing (other instances of) themselves to measure runtime overhead. Table II shows the impact of the instrumentation of Java bots on runtime performance.

Note that the different strategies correspond to increasing code complexity in the sense of cyclomatic complexity [10]. As described above, the monitoring utility is used to check for interrupted state and heap memory allocation. The latter is relatively expensive, and it is reasonable to perform these checks only after a certain number of invocations. The dependency of the runtime on these check intervals is also reported in Table II, columns 3–5[7].

TABLE II
JAVA BOT PERFORMANCE DEPENDING ON INSTRUMENTATION

program	run time (ms)			
	no	1	10	100
anxious	63	67	64	59
greedy	33	36	33	30
prime number	50	573	132	86
no factors left	64	598	149	77
best advantage	254	2021	581	393

For the simple bots, there is almost no measurable runtime performance penalty caused by instrumentation either. However, the situation is different for the more complex bots, where instrumentation increases runtime by a factor of up to 8 for the best advantage strategy. This can be managed by increasing the interval size when memory consumption and interrupt status are checked.

[6]https://code.google.com/p/guava-libraries/

[7]E.g., the 10 in column 4, row 2 means that the interrupted state and the heap space allocated are checked after every 10 method invocations

Table III reports the equivalent data for bots written in Python. This shows some unexpected results – instrumentation actually speeds up execution! This is due to the different, faster dispatch method used internally by the Jython debugger.

TABLE III
PYTHON BOT PERFORMANCE DEPENDING ON INSTRUMENTATION

program	run time (ms)			
	no	1	10	100
anxious	130	244	243	253
greedy	75	258	243	267
prime number	82	267	295	306
no factors left	77	287	321	309
best advantage	13139	812	746	703

TABLE IV
JAVA AND PYTHON BOT PERFORMANCE DEPENDING ON CACHING

language	without cache	with cache
java	7302	163
python	5240	776

We also assessed the impact caching has using the following experimental setup. We used a round robin tournament for the Java and Python versions of the bots from [12], and measured the runtime of 5 tournaments with and without cache. The results are reported in Table IV. It is apparent that caching can significantly improve performance, and is crucial for achieving sufficient throughput.

VI. USER INTERFACE

Figures 1 – 2 show the browser-based user interface. The editor (Figure 1) provides a simple web-based IDE based on the widely used ACE component[8]. The number of available functions has been minimised to retain the simplicity necessary for educational programming environments. The editor supports syntax highlighting and formatting. Support for auto-completion is planned but not yet available.

Note the sharing feature in the main menu. This flags the bot as shared, and produces a URL that can be shared on social networks to invite other users to play against this bot.

Bots are submitted to the server for storing and building. As discussed above, the server processes the bot in a build pipeline. Errors that occur at the various stages are displayed to the user in the console panel underneath the code editor. Also, a marker is set to highlight the critical code causing the problem. This includes compilation errors. In addition, violations for the several verification rules are displayed. An example can also be seen in Figure 1. Here the user has attempted to store a bot that forces the JVM to exit (`System.exit(0)`). This is discovered during the static bytecode verification step (BYTE_CHECK), the respective error message is displayed and the line of code is highlighted.

Games can be played in the test environment shown in Figure 2. While the code editor is generic, the test environment is game specific. The games are executed on the server, the

[8]http://ace.c9.io/

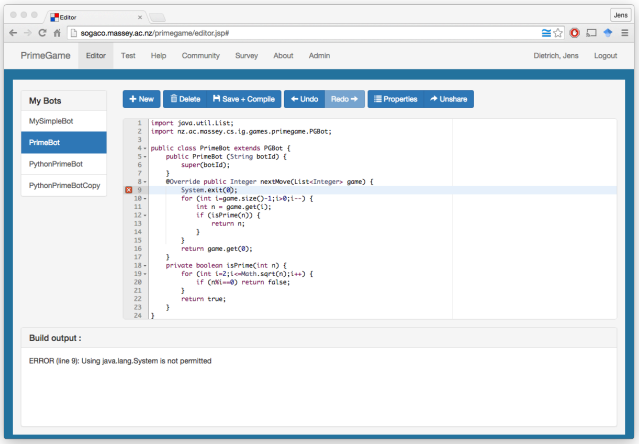

Fig. 1. SoGaCo code editor reporting a verification error

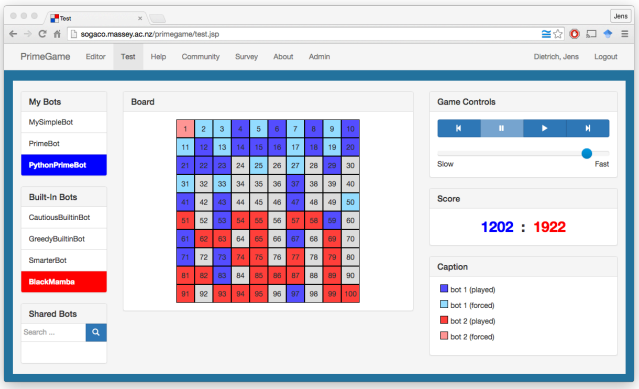

Fig. 2. SoGaCo PrimeGame test environment

results are recorded, (JSON-) encoded and returned to the client for animated replay. The animation controls are on the right side of the screen. The left side contains a list of bots, two bots must be selected from the list to play a game. These bots are colour-coded upon selection (red and blue, respectively). Note that bots shared by other users can be looked up and selected as well. When a shared bot URL is loaded, the test page is loaded with the shared bot preselected as opponent.

In the game shown in Figure 2, a simple cautious bot plays against a more sophisticated (blue) bot that plays the largest prime number available.

VII. Conclusion

We have presented SoGaCo, a browser-based educational environment for teaching programming and algorithms and data structures. While we have not yet systematically evaluated SoGaCo, evidence from end user testing with a group of 25 students suggests that this environment can promote engagement and participation. More work on validation is planned with several universities and highschools in the near future.

There are several technical features we plan to integrate, such as traceability and visualisation of code execution, similar to the respective feature of the Python tutor [6], the provision of a console for bots, support for code autocompletion, and support to organise tournaments.

References

[1] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[2] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.

[3] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[4] Brian Goetz and Tim Peierls. *Java concurrency in practice*. Pearson Education, 2006.

[5] Mike Grogan. JSR-223 Scripting for the Java ™ Platform. *Final Draft Specification, version*, 1, 2006.

[6] Philip Guo. Online Python Tutor: Embeddable web-based program visualization for CS education. In *Proceedings SIGCSE '13*, New York, USA, 2013. ACM.

[7] Philip Guo. Python is now the most popular introductory teaching language at top us universities. *BLOG@ CACM, July*, 2014.

[8] Poul Henriksen and Michael Kölling. Greenfoot: combining object visualisation with interaction. In *Proceedings OOPSLA'04*, pages 73–82. ACM, 2004.

[9] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[10] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[11] Manfred Meyer. The PrimeGame reloaded: Finding the right balance between cooperation and competition in undergraduate computer science classes. In *Proceedings EDULEARN11*. IATED, 2011.

[12] Manfred Meyer and Jens Fendler. The PrimeGame: Combining skills in undergraduate computer science programmes. In *Proceedings INTED'10*. IATED, 2010.

[13] Mathieu Muratet, Patrice Torguet, Jean-Pierre Jessel, and Fabienne Viallet. Towards a serious game to help students learn computer programming. *International Journal of Computer Games Technology*, 2009:3, 2009.

[14] Mathew Nelson and Flemming N Larsen. Robocode. *IBM Advanced Technologies*, 2001.

[15] Yolanda Rankin, Amy Gooch, and Bruce Gooch. The impact of game design on students' interest in cs. In *Proceedings GDCSE'08*, pages 31–35. ACM, 2008.

[16] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[17] Ian Utting, Stephen Cooper, Michael Kölling, John Maloney, and Mitchel Resnick. Alice, greenfoot, and scratch–a discussion. *ACM Transactions on Computing Education (TOCE)*, 10(4):17, 2010.

[18] Alf Inge Wang and Bian Wu. An application of a game development framework in higher education. *International Journal of Computer Games Technology*, 2009:6, 2009.