A platform for multi-language component-based software development

Carlos Edmilson da Silva Maia
Laboratório de Computação Aplicada
Universidade Federal de Santa Maria
Santa Maria, RS, Brazil
edmilsonmaia@gmail.com

Marcos Cordeiro d'Ornellas
Laboratório de Computação Aplicada
Universidade Federal de Santa Maria
Santa Maria, RS, Brazil
ornellas@inf.ufsm.br

Abstract— Creation and usage of modules with specific functionalities have been appearing in many stages of software development. The goal is that the end product turns out to be easier to maintain and that the provided functionality is easily reused in further software projects. Many times those functionalities are more easily or quickly implemented with different programming languages that might prove more adequate for the problems at hand. The goal of this work is to offer programmers a platform that allows for the use of several different scripting languages in a simple and accessible way.

Keywords-component-based software development;software engineering;design patterns.

I. INTRODUCTION

The use of modules and creation of modular systems is becoming more common within software development. Such components bring several advantages to the process, including reduction of the initial development time, lower costs for individual components, higher quality and easier and more manageable maintenance [7]. Component-based software development is generally recognized as crucial to development of dependable software systems [8][9]. When developing software this way, it might become easier to reuse code that was originally developed for other projects while standing useful for solving current problems.

According to this software development methodology, it would be handy if the programmer is able to write the modules included in the system using a set of different languages. Among the benefits of such improvement we can highlight:

- The functionality of a given module might be more easily implemented with a specific programming language;
- The programmers available for development might be limited to of have more experience with certain languages;
- It might be preferable to reuse code or to use libraries written in a specific language;
- It might be desirable to develop a prototype of the module before implementing the final version, with this prototype being written in a language that performs worse but offers a shorter development time.

This work resulted in a platform that allows for the development of a component-oriented system using independent modules written in different scripting languages.

II. TECHNOLOGY

The platform, originally development to facilitate the quick prototyping of games, allows for communication, done through function calls and event dispatching, between modules written in Java and any scripting language supported by (or that has support for it written in accordance to) the Java Specification Request 223 (JSR223), Scripting for the Java Platform. JSR223 provides several new features in the usage of scripting languages, allowing the loading and execution of script, access to Java objects from those scripts, passing of arguments from Java to the scripts and access of script objects from Java [5]. The usage of scripting languages like Python, Perl, Ruby and Groovy usually allows a higher abstraction level than static languages like C and Java, facilitating the use of their features, expediting, for the programmer, the usage of pre-existing components and functionality [4][5].

The choice for Java has been made due to the fact that it has a good flexibility and consistency in the task of dealing with scripting languages and dynamic class loading. In addition to that, the synergy between the Java platform and scripting languages provides an environment in which developers and users can collaborate to create more dynamic and useful applications [5]. It is also worth noting that Java is a language that was created for the purpose of being a object-oriented programming language, and the use of reusable software components is considered to be a natural extension of the object-oriented paradigm [3][6]. Other characteristics of the Java language include static typing, C/C++ derived syntax and compiling to a intermediate code that can be executed under any platform, as long as the Java Virtual Machine is installed.

With the discussion upon technologies and programming languages set up on a stable stage, the research moved its focus on the matter of which technology might be used to manage and allow for the communication between the modules. One of the most prominent ways of using modules with Java is found in the OSGi platform, which is a service platform for the creation of service-oriented modular programs with the Java language. It is a platform composed of four layers – the security layer, the modularization layer, the lifecycle layer and the service layer – that was created to offer to the modularization problem in Java [1]. The main problem with OSGi is exactly the abundance of services, interfaces and layers that are made available. Even though they are very useful for large projects like Eclipse, they will result in an unnecessary complexity to the platform.

During the development process, we might observed that the OSGi was the best option available till the Java 1.6. By using that version of Java, it became much more viable and simple to use the functionality provided by Java’s Service Loader API, since this API was reviewed and improved in this new version. This API allows for a simple management of classes – modules – that can be dynamically loaded and used during runtime. It was noted that this API offers enough functionality for the platform’s needs without adding the inherited complexity of the OSGi platform, which would result in an additional barrier for development and usage of the final system.

Based on those conclusions, the module manager, which would be the core of the system, was developed using the ServiceLoader functionality to load modules written in Java, allowing that each module has its most adequate implementation to be used [2].The functionality for loading modules written in scripting languages was developed specifically for the platform. At the end, the system offered an abstraction level that allows for the programmer to use different modules without noticing any difference between Java modules or modules developed in scripting languages. There is a single interface that is used by modules of all languages to access functionality present in other components of the system.

III. Implementation

The initial approach was the creation of well-defined interfaces that would be implemented by the modules. This implied, however, that the functionality that would be implemented on a system written on this platform would have to be statically defined beforehand. Whenever a programmer needed to add new functionality, he would need to add new interfaces to the platform so other modules could access it. Those interfaces would invariably be implemented in Java to be either included in the platform’s code or to be dynamically loaded, when their features would finally be accessible to other modules.

The problems of updating and implementing new features in existing modules have to be managed. Every time a module has to offer a new method to the rest of the system or to change the signature of an existing method, the programmer needs to include or update that functionality in the module's interface. Components that use that module need to be rebuilt in order to be compatible with the new version of the platform. The implementation has to take into account the usage and integration of different scripting languages with Java. Therefore, this integration needs to be clear to the programmer.

It is necessary to allow for the programmer to have the mentioned limitations alleviated or removed in such a way that allows the applications to be easily extended and for the communication between modules to be as straightforward as possible (from the programmer's point of view). To achieve this goal, two communication systems were defined and adopted: event dispatching and method calling. These methods were tailored for the platform in order to achieve the goals of the project.

The event dispatching present in the platform is quite simple: modules can register their interest on being notified by certain events (which includes listening to categories of events or even all events generated in the system) and can also dispatch their own events to the rest of the system. All modules that requested to be notified of the event receive information regarding it from the platform when it is dispatched. The event includes its name and an optional arguments list of variable size. This list works like an arguments list of a method call.

The direct method calls, however, are offered up by the modules, which are allowed to register them in the platform so other components can call them directly. When one of those calls is made, the platform will forward the call, including its argument list, to the providing module, and will return the method’s results to the caller. The platform, however, does not generate fatal errors if the methods invoked in the system do not exist. This is done so the programmer is allowed to call methods that have not been implemented yet or that are implemented in a module that is not available at runtime.

The platform deals with all needs of argument conversion of basic types between modules of different languages, both for event arguments and for method’s arguments and return values. The event listening registration and method offering made by the modules is done through the usage of methods of the platform specific for this task. In the case of modules implemented in Java, it is also possible to do such registrations by using Java Annotations. In case of Java modules, the platform also provides automatic functionality to dynamically load libraries made available in Java Archives (JARs) and the usage of functionality provided by native functions through the use of the Java Native Interface (JNI). The files relevant for both cases can be kept inside the module’s directory, together with any additional assets used by it, in a way that it is easier to maintain isolation and portability of the components.

It can be also noted that even though the platform only allows for direct usage of Java and scripting languages, it is also possible to use libraries written in compiled languages such as C/C++ and assembly by usage of the aforementioned JNI support. The programmer that whishes to do so will need to write a wrapper module in Java or to look for a library written in Java that already does so.

Fig. 1 shows what the directory structure looks like for an application implemented using the platform. There is a directory called "jsr223.lib" that stores the Java libraries required for handling scripting languages. The "modules" directory has a folder for each module present in the application, each with his "lib" and "jni" directories (for Java libraries and for Java Native Interface libraries, respectively) if the module requires any additional libraries. Those are loaded automatically by the platform at runtime.

The modules themselves are represented by a single JAR file (in the case of a Java module) or a file, named after the module's name, with the extension related to the language used (in the case of a script module). Each module can add more files and directories under its own directory if needed.

The example presented in Fig. 1 is a simple application that consists of a small keyboard-controlled colored circle drawn on the screen. The logic of the program is present in the SimpleGame module, which is written in JavaScript on a single file as shown in fig. 2. This module created the application window, handles user input and updates the screen by using functionality provided by other modules. An "assets" directory can also be found inside the module's directory; it contains the texture for the circle. The creation of the assets directory to hold texture files was arbitrarily decided by the module creator; besides the initial script or JAR file (mandatory) and the "jni" and "lib" folders (optional), all the other resources the programmer needs can be organized inside the module's directory as he sees fit.

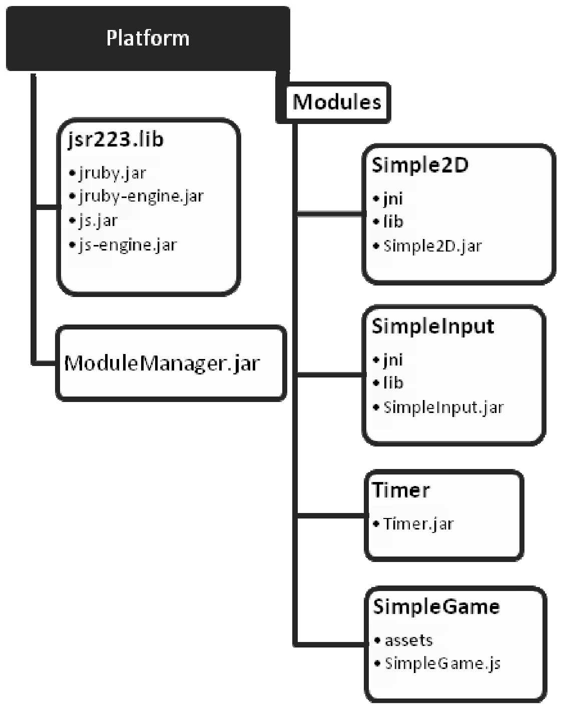

Figure 1. Directory structure of a sample application

```
var texture = Module.getDirectory()+"/assets/circle.png";
var x = 0; var y = 0; var speed = 1.5;

function onLoad() {
    // system.init is generated by the ModuleManager after
    // all modules have been loaded
    Module.addEventListener("system.init", "init");
    // timer.fixed is generated a hundred times per second
    // by the Timer module
    Module.addEventListener("timer.fixed", "update");
}

function init(event) {
    // creating a window using a method provided by the
    // Simple2D module
    Module.doMethod("2d.doWindow", 800, 600, "My Game");
}

function update(event) {
    // moving the texture on the screen based on the data
    // obtained from the Input module
    if(Module.doMethod("input.isPressed", "left")==true)
          x -= speed;
    if(Module.doMethod("input.isPressed", "right")==true)
          x += speed;
    if(Module.doMethod("input.isPressed", "up")==true)
          y -= speed;
    if(Module.doMethod("input.isPressed", "down")==true)
          y += speed;

    if(x < 0) x = 0; if(x > 800) x = 800;
    if(y < 0) y = 0; if(y > 600) y = 600;

    // drawing the texture on the screen using methods
    // provided by the Simple2D module
    Module.doMethod("2d.begin");
    Module.doMethod("2d.draw", texture, x, y);
    Module.doMethod("2d.end");
}
```

Figure 2. SimpleGame.js

The Simple2D module is a module written in Java that uses the Java OpenGL (JOGL) library to draw two-dimensional textures on the screen. The JOGL library files are placed in the "lib" directory, while the native libraries (".dll", ".so" and ".osx" files for the Windows, Linux and Mac OS X operating systems, respectively) are placed in the "jni" directory. The SimpleInput module is also written in Java and follows a similar pattern as the Simple2D module, using the JInput library to handle input from the user.

The last module needed for this application is the Timer module. It is also written in Java and the SimpleGame module uses it to update the application logic as time passes.

It is important to notice that each module is isolated inside its own folder; any module can be added or removed by moving its directory into the modules directory or out of it, respectively. Since the platform intentionally generates no fatal errors if inexistent methods are called, changes caused by addition of removal of modules in the system are noticed as soon as the platform is restarted with no need to change any of the module's code. In the case of the example application, for instance, if we removed the Timer module then the application would stay frozen in the same initial state. If we removed the SimpleInput module, the application would work as usual but ignore player input. If we removed the Simple2D module, the input would be correctly processed but the user would have no visual feedback, since no application window would even be created. Finally, if we removed the SimpleGame module, all the others module would be loaded and the platform would enter an idle state, since no active module would do any actions like create a window and update the logic of the application.

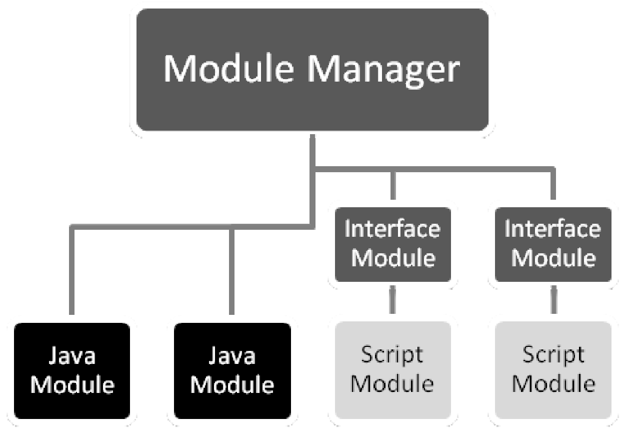

Figure 3. General overview of the platform structure

Fig. 3 shows a general overview of how the platform handles modules. The module manager and the interface modules are part of the system and are written in Java. The interface modules are created and handled automatically by the platform as needed during runtime to manage modules written in scripting languages. Those components are responsible for handling argument translation, method forwarding and event dispatching related to the script module they manage. The interface modules extend the same functionality as the Java modules; this allows the system to provide the same access to both kinds of components with no difference.

IV. RESULTS

To create a new module using a scripting language, the programmer only needs to create a directory for the component containing a file, written in the chosen language, with a single initialization method. The integration of the newly-written module to the rest of the system is done automatically by the platform during runtime, when it finds out which modules are available to load and initialize.

There are some additional tasks that need to be performed to implement a new module in Java, such as including the platform's libraries, extending a specific interface and including meta-information for the ServiceLoader. Besides those additional tasks, however, the integration to the rest of the system has no further differences from a module written in a scripting language.

Modules can be included and removed from the target system simply by handling their directories accordingly, as any module found in the modules path is automatically loaded by the platform on the next execution. All that is needed to get the system running is to start the JAR that serves as a basis for the platform. The initialization dispatches events that allow for a given module to perform the initial tasks of the program by using functionality provided by other components. As a result, the programmer that wishes to write a program that runs over this platform should select the modules he needs (or develop them) and write a special module that acts as the main application itself. The platform makes no distinction between this specific module and the others.

With regards to performance, this system adds a noticeable overhead to the final software, reducing yield by 10% to 15% in relation to the same application written in a single language without the need for a central component to translate the method calls. With addition to that, several serious performance problems related to garbage collection were noticed during development. It was later found that in applications with continuous execution, like games, where you have repeated and frequent method calls every second, the allocation and deallocation of memory was done constantly by the part of the system that forwards method calls to scripting languages. This caused the Java Virtual Machine to trigger its garbage collection routines more frequently, generating pauses and delays during execution. In the case of games, this resulted and frequent pauses in the graphics processing. This problem was solved by configuring Java to use the low interruption concurrent garbage collector (Concurrent Mark Sweep Garbage Collector, CMS) and the Parallel Young Generation collector for the tenured generation and young generation garbage collections, respectively.

V. CONCLUSION

The developed platform proved to be good enough in accordance with the original goals that were set for the project. The creation of new modules has the simplicity expected by the original project.

This work promoted the idea of quick prototyping and systems development, providing functionality for easier and more maintainable software development. It should be pointed that performance is not a primary concern and its impact on the final system may be ignored in face of the features in the platform.

ACKNOWLEDGMENT

The authors would like to thank the Laboratório de Computação Aplicada (LaCA) of Universidade Federal de Santa Maria (UFSM) for the provided infrastructure.

REFERENCES

[1] D. Chappell and K. Kand, "Universal middleware: what's happening with OSGi and why you should care," unpublished, 2009.

[2] M. Fowler, "Inversion of control containers and the dependency injection pattern, " unpublished, 2004.

[3] J. Hopkins, "Component primer," Communications of the ACM, 2000.

[4] J. K. Ousterhout, "Scripting: higher level programming for the 21st century," IEEE Computer, 1997.

[5] J. O'Conner, "Scripting for the java platform," unpublished, 2006.

[6] C. Szyperski, "Component software: beyond object-oriented programming," Addison Wesley Longman Ltd, 1998.

[7] P. Vitharana, "Risks and challenges of component-based software development," Communications of the ACM, 2003.

[8] G. Blair, T. Coupaye and J. Stefani, "Component-based architecture: the fractal initiative," Annals of Telecommunications, 2009.

[9] L. Bass, P. Clements and R. Kazman, "Software architecture in practice," SEI Series in Software Engineering 2nd ed., 2003.