Towards the Integration of Metaprogramming Services into Java

Ignacio Lagartos, Jose Manuel Redondo and Francisco Ortin
Computer Science Department, University of Oviedo, c/Calvo Sotelo s/n, 33007, Oviedo, Spain

Keywords: Java, Metaprogramming, Structural Intercession, Dynamic Code Evaluation, Static Typing, Early Type Error Detection.

Abstract: Dynamic languages are widely used in scenarios where runtime adaptability is a strong requirement. The metaprogramming features provided by these languages allow the dynamic adaptation of the structure of classes and objects, together with the evaluation of dynamically generated code. These features are used to build software capable of adapting to runtime changing environments. However, this flexibility is counteracted with the lack of static type checking provided by statically typed languages such as Java. Static type checking supports the earlier detection of type errors, involving a valuable tool in software development. In this position paper, we describe the steps we are following to add some runtime metaprogramming services to Java. We intend to provide the runtime flexibility of structural intercession and dynamic code evaluation provided by most dynamic languages, without losing the robustness of the compile-time type checking of Java. The metaprogramming services are provided as a library so, unlike other existing systems, any standard virtual machine and language compiler could be used.

1 INTRODUCTION

Dynamic languages have turned out to be suitable for specific scenarios such as rapid prototyping, Web development, interactive programming, dynamic aspect-oriented programming and runtime adaptive software (Redondo, 2015). Most dynamic languages provide metaprogramming services that allow treating programs like data, and modify them at runtime (Ortin, 2002). Fields and methods can be added and removed dynamically from classes and objects (structural intercession), and new pieces of code can be generated and evaluated at runtime, without stopping the application execution (Ortin, 2003). These services make it easier to develop runtime adaptable software in dynamic languages (Paulson, 2007).

In order to provide that runtime adaptability, dynamic languages commonly implement a dynamic type system, postponing type checking until runtime. One limitation of this approach is that every type error is detected at runtime. On the contrary, statically typed languages such as Java and C# commonly detect many type errors at compile time, when the programmer is writing the code. This lack has been recognized as one of the limitations of dynamically typed languages (Meijer, 2004). The absence of compile-time type information also involves fewer opportunities for compiler optimizations, and the extra runtime type checking commonly implies performance costs (Ortin, 2014b).

In previous works, we have inferred type information at compile time to provide early type error detection in dynamically typed code (Garcia, 2016; Quiroga, 2016). In this work, we aim at providing metaprogramming services to the statically typed Java language. The objective is to increase the runtime adaptability of Java, without loosing the early type error detection and runtime performace of its static type system.

The main contribution of this position paper is the description of a Java library aimed at providing metaprogrammning services, maintaining its static type system. Particularly, we intend to add structural intercession of classes and its existing objects at runtime, allowing the dynamic modification of their structure. We also provide the evaluation of dynamically generated Java code. This is a work in progress position paper.

Lagartos, I., Redondo, J. and Ortin, F.
Towards the Integration of Metaprogramming Services into Java.
DOI: 10.5220/0006355802770284
In *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2017)*, pages 277-284
ISBN: 978-989-758-250-9

2 LIBRARY INTERFACE

The metaprogramming services are provided as a library of the Java platform. We modify neither the Java Virtual Machine (JVM) nor the language implementation. Thus, unlike other approaches (Würthinger, 2013; Redondo, 2008), any standard JVM and Java compiler can be used.

To describe the interface of the library, this section presents an example that dynamically modifies the structure of the class shown in Figure 1.

At runtime, the code modifies the structure of the existing class shown in Figure 1.

```
1.  public class Dog {
2.      public void bark(){
3.          System.out.println("Woof!!");
4.      }
5.      public void shake(){
6.          System.out.println("Shakes");
7.      }
8.  }
```

Figure 1: Example Java class.

The code in Figure 2 modifies the `Dog` class at runtime using structural intercession (read and write reflection) (Ortin, 2005). We add a `name` field to every `Dog` instance at runtime, evolving the structure of the class. Besides this new field, we also add two new `getName` and `setName` methods. Moreover, the implementation of the existing `bark` and `shake` methods are modified, so that they consider the new `name` field.

The proposed library allows performing the five operations individually. Additionally, it also provides the execution of all the operations at the same time with the concept of transaction. Figure 2 creates one transaction (line 2) with the five operations (lines 4 to 10). Then, the transaction is executed atomically in line 12. If all the operations can be executed, the program continues; otherwise, no operation is performed. For example, if the body of `setName` has a type error (line 7) a `CompilationFailedException` exception error will be thrown and none of the five operations will be executed.

If we want to invoke a newly added method (e.g., `setName`), we should provide a new mechanism, since that method is added later, when the application is running. A direct invocation to `setName` will not be compiled because that method does not exist at compilation time. For this purpose, our library provides the `getInvoker` method (line 14). It returns the standard `BiConsumer` interface added to Java 8 (Oracle, 2017a). Its `accept` method executes `setName`, which was added at runtime. Unlike the Java reflection API, we generate statically typed code (at runtime), so we expect to obtain a significant performance benefit (Ortin, 2014a).

We have just shown how the library provides structural intercession; we now describe how to obtain dynamic code evaluation (i.e., the `eval` function in Lisp, Python and JavaScript languages). Figure 3 shows this capability.

Line 21 first adds another implementation of the `bark` method. This line shows how to perform a single intercesive operation without using a transaction. It also shows that methods could be overloaded at runtime, without breaking the rules of the type system. The following statements in Figure 3 (lines 23 to 26) perform the dynamic evaluation of the string "`dog.bark(nTimes)`". It is important to notice that the code is represented as a string, and hence it can be built dynamically, depending on the runtime environment. That is to say, the code is

```
1.  // Create transaction
2.  IntercessorTransaction transaction = new IntercessorTransaction();
3.  // Add field name
4.  transaction.addField(Dog.class, String.class, "name");
5.  // Add get/set
6.  transaction.addMethod(Dog.class, "getName", MethodType.methodType(String.class),"return name;");
7.  transaction.addMethod(Dog.class, "setName", MethodType.methodType(void.class, String.class),"name = value;", "value");
8.  // Modify existing methods
9.  transaction.replaceImplementation(Dog.class, "bark", "System.out.println(this.name + \": Woof!!\");");
10. transaction.replaceImplementation(Dog.class, "shake", "System.out.println(this.name + \": Shakes\");");
11. // Execute transaction
12. transaction.commit();
13. // Get invoker for 'setName'
14. BiConsumer<Dog, String> setName = Intercessor.getInvoker(Dog.class, "setName", BiConsumer.class, Dog.class, String.class);
15. // Check name field
16. String name = readline("Name: ");
17. Dog dog = new Dog();
18. setName.accept(dog, name);
19. dog.bark();
```

```
Name: Rufus
Buddy: Woof!!
```

Figure 2: Example adaptation of the `Dog` class using a transaction collecting 5 intercesive operations.

evaluated dynamically (`dog` refers to the dynamic state of the `dog` object created in line 19, the same as `nTimes`).

```
20. // Overload 'bark' method
21. Intercessor.addMethod(Dog.class, "bark",
        MethodType.methodType(void.class, int.class),
        "for (int i = 0; i < times; i++) bark();", "times");
22. // Evaluate the call to overloaded method
23. BiConsumer<Dog, Integer> barkN =
        Evaluator.generateEvalInvoker("dog.bark(nTimes)",
        BiConsumer.class, new String[] {"dog", "nTimes" },
        Dog.class, int.class);
24. // Invoke overloaded method
25. int nTimes = readNumber("Times: ");
26. barkN.accept(dog, nTimes);
```

```
Times: 3
Buddy: Woof!!
Buddy: Woof!!
Buddy: Woof!!
```

Figure 3: Dynamic evaluation of a single expression.

We have just seen how to evaluate an expression dynamically. The proposed library also provides the evaluation of multiple statements, and even the creation of a whole class. Figure 4 shows an example of that. A new `TrainedDog` class is added at runtime (line 29). This class extends the existing `Dog` class, which was modified at runtime (Figures 2 and 3).

Line 28 in Figure 4 asks for the code to be evaluated. The user dynamically writes the code with gray background color, which generates the new `TrainedDog` class. This class implements the `train method` that receives a function as a parameter (the standard `Consumer` Java 8 interface allows passing lambda expressions as arguments). Those functions can later be asked to the trained dog with the `order` method.

Line 36 creates an instance of a trained dog, trains it with the "shake" order (line 42) and orders it to shake (line 43). The output in Figure 4 shows how the actions of the dog depend on its training. It also shows how a newly added class can extend another class defined statically, which in turn was modified dynamically.

All the metaprogramming operations are statically typed. If the code has a type error, the library dynamically throws a `CompilationFailedException` describing the compiler error. Besides, the dynamically generated code does not use reflection, so we avoid its runtime performance cost (Conde, 2014).

3 ELEMENTS OF THE LIBRARY

3.1 Metaprogramming Services

After presenting an example, we detail the functionalities of the proposed library. Regarding structural intercession, we provide:

- Adding, deleting and updating fields of classes and, thus, of all their running instances. The update action means changing the field type.
- Replacing method implementations. Without modifying their signature, the body of methods (their code) is replaced with a new one.
- Adding, deleting and updating methods (including their implementations). As with fields, updating means changing the method signature. Adding methods include overloading their implementation (as in Figure 3).
- Additional methods to provide the access to new fields and methods. It is similar to the reflection API, but aimed at accessing the members added at runtime.

These metaprogramming services are applied to classes. Evolving a class implies the dynamic adaptation of its instances. Since Java is a class-based language (Redondo, 2013), we do not provide the dynamic adaptation of a single object. That possibility is not included in the Java type system and, as mentioned, we want to take advantage of the benefits of its static type system.

Regarding dynamic code evaluation, our library provides the following services:

- Dynamic evaluation of expressions. This is the traditional `eval` functionality provided by most dynamic languages. Only one single expression is evaluated, and its value is returned. The expression may access any element of the running application.
- Dynamic execution of Java code. We provide the execution of either a sequence of statements or the contents of a Java file. As before, the code may depend on the runtime environment.

3.2 Runtime Adaptation

To describe the elements of the library, we explain how the system behaves at runtime, when the example in Section 2 is executed. Figure 5 shows the runtime steps of our example.

One of the issues when implementing the proposed library is that the JVM does not allow reloading classes dynamically (Pukall, 2008). Once a class is loaded into memory, its code cannot be changed. The only exception is the capability of modifying method implementations, added in Java 5 with the `instrument` package (HotSwap). For this reason, we propose a system based on creating new class versions at runtime.

Every time a class is modified with our library, a new class version is created and loaded at runtime.

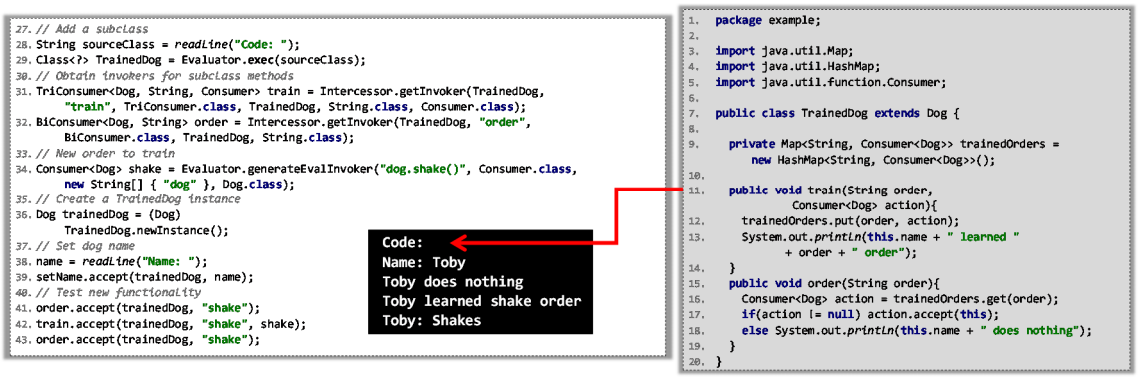

Figure 4: Dynamic evaluation of a Java file.

The new class version holds all the changes made to the previous version. If those changes are collected in one transaction, only one new version is created regardless the number of class modifications.

Every class should provide a link to its last updated version, so we add a `_newVersion` private field to all the classes (Figure 5). To perform this field addition transparently to the user, we implement a new Java `ClassLoader` and modify all the classes, using the Java Agents API added to Java 5 (Oracle, 2017b). This process is done at load time, so there is no runtime performance penalty when the JVM reaches a steady state (Georges, 2007).

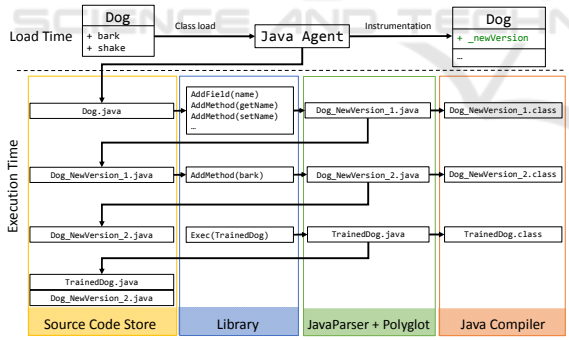

Figure 5: Runtime steps for adapting the `Dog` class.

The first prototype of our library requires the source code of the applications (once it is mature enough, we will work at the JVM binary code level). Figure 5 shows how the source code of every class version is stored. Using this source code storage, changes to the classes are implemented by changing the source code, recompiling and loading them into memory.

When the user modifies the `Dog` class, a new version `Dog_NewVersion_1` is generated. This new class holds the last version of the original `Dog` class. The `_newVersion` field of `Dog` instances will be updated at runtime. This field update is performed lazily, when the object is first accessed after class adaptation. In that moment, the `_newVersion` reference is updated, and the object state is transferred to the new class version (`Dog_NewVersion_1`). This process consumes extra execution time, but it is performed only once per instance.

One issue is how we manage to replace the existing code accessing `Dog` fields with code that accesses the corresponding fields in the last class version. This is done by using the `invokedynamic` bytecode added to Java 7 (oracle, 2017c). Our `ClassLoader` replaces all the field access bytecodes with `invokedynamic`. Therefore, we can change the functionality of field access with Java 7 `MutableCallSites`. We use the JINDY API to utilize `invokedynamic` from the Java language, getting rid of writing JVM assembly code (Conde, 2014).

Another issue is how we manage to replace method invocations with invocations to the new class version (recall that the last version holds the actual state of the objects, i.e. the appropriate `this`). We first modify the implementation of every method in `Dog`, using the `instrument` Java 5 package. The new code will simply invoke another method in `Dog_NewVersion_1`: `bark` calls `_bark_invoker`, `shake` calls `_shake_invoker`, and so on (see Figure 6). The purpose of those `invoker` methods is to implement the lazy object state transfer and `_newVersion` update described above. After doing this update (only once per instance), the last method version (e.g., `bark` and `shake`) is called in `Dog_NewVersion_1`. In this way, if a method in an updated class is called, it will call the corresponding `invoker` in the last class version; if necessary, object

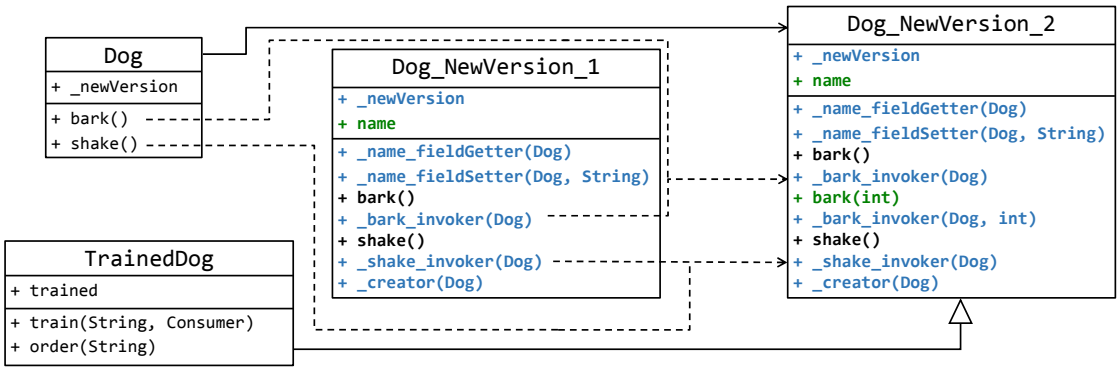

Figure 6: Runtime structure of the existing class versions.

state is transferred; and then the last method version is called.

When the programmer adapts an already adapted class (e.g., Figure 3) a new `Dog_NewVersion_2` is created, compiled and loaded (Figure 5). The `_newVersion` of both the original `Dog` and `Dog_NewVersion_1` will be lazily updated to the last class version. Similarly, all the method bodies will be replaced with direct invocations to the invokers in the last class version. The purpose is that, once instance states have been transferred, the runtime performance cost does not depend on the number of class versions. Figure 6 shows the runtime structure of classes after performing the two class modifications in Figures 2 and 3. After updating all the instances of the first version, `Dog_NewVersion_1` is useless –that is why it is now shown in Figure 6.

3.3 Dynamic Code Evaluation

Figures 3 and 4 show how our library provides dynamic code evaluation. If we just need the dynamic evaluation of an expression (Figure 3), the library creates a temporary class, with a method that implements that expression. We need to provide a mechanism to execute that dynamically generated code, following the Java type system. For this purpose, we make the dynamically generated class to implement one of the "functional" interfaces added in Java 8 (`function` package) (Oracle, 2017a). In this way, the interface provides the specific type of the expression to be evaluated.

For evaluating a sequence of statements, we follow a similar approach: the method body is the code provided by the user, and `void` is the returned type. For a whole class (Figure 4), we just place the code in a Java source file and compile it.

As mentioned, class adaptation is achieved by modifying the application source code. However, code manipulation is not an easy task. To distinguish the elements in a program, code should be represented with tree- or graph-based data structures such as AST (Abstract Syntax Trees) (Ortin, 2007).

To manipulate classes (add, remove or update fields and methods) we used the JavaParser tool (Figure 5) (JavaParser, 2017). It allows us to take Java code, obtain its AST, modify it, and regenerate the output Java code. Then, we simply call the `JavaCompiler` class added in Java 6.

In the dynamic evaluation of code, there is an important issue that should be considered. When programmers are writing code to be evaluated dynamically, they are not aware of the different class versions. Our library provides programmers the abstraction that the `Dog` class is being dynamically changed. For example, the programmer may be interesting in running the code `dog.setName("Rufus")`. However, if this code is evaluated, it will prompt a type error since `Dog` has no `setName` method (`Dog_NewVersion_1` does).

Therefore, we need to perform some changes in the code to be evaluated at runtime. Those changes are related to the types: if the code is accessing a new member added to a type, its last version must be used instead. We, thus, need to know the type of every expression to be evaluated dynamically (e.g., `dog` in our example). At the implementation level, we just replace `setName` with `setName_invoker`, since the latter method always class the last version.

To perform these changes to dynamically evaluated Java code, we use the Polyglot front end compiler for building Java language extensions (Polyglot, 2017). Following the Visitor design pattern (Gamma, 1994), we traverse the AST and replace the method invocations which types have evolved. Finally, we generate the modified code, compile it and load it into memory.

4 RELATED WORK

There are different works aimed at adding some metaprogramming features to Java. Most of them are based on modifying the implementation of the JVM.

Würthinger *et al.* modify the JVM to allow the dynamic addition and deletion of class members (Würthinger, 2010). They also support changing the class hierarchy at runtime. They ensure the type rules of the Java type system, and they also verify the correct state of the program execution. After the adaptation, runtime performance is penalized by 15%, but this value converges to 3% when the JVM reaches a steady state (Würthinger, 2013). This is currently the reference implementation of the Hot Swap functionality included in JSR 292, which was not finally included in the standard platform (Oracle, 2011).

JVOLVE is another implementation of the JVM to support evolving Java applications to fix bugs and add features (Subramanian, 2009). JVOLVE allows adding, deleting and replacing fields and methods anywhere within the class hierarchy. They modify the class loader, JIT compiler and garbage collector of the JVM to provide those services. To adapt the running applications, JVOLVE stops program execution in a safe point and then performs the update. Class adaptation is controlled by *transformer* functions that can be customized by the user.

Iguana/J extends the JVM to provide behavioral reflection at runtime (Redmond, 2002). The programmer may intercept some Java operations such as object creation, method invocation and field access. The new behavior is specified by the user, and a Meta-Object Protocol (MOP) adapts the application execution at runtime. When a MOP is associated to an object, it handles the operations against that object and provides the services to adapt its execution. Each modifiable operation is represented with one MOP class that the programmer has to extend to define the expected runtime adaptations. The MOP classes and objects are compiled following the Java type system.

Java Distributed Runtime Update Management System (JDRUMS) is a client-server system that allows changing a runtime program and adding more functionality to it (Andersson, 2000a). Servers provide the update services to the clients, which run in the JDRUMS virtual machine. That virtual machine is a JVM extension that provides distributed dynamic updates (Andersson, 2000b). Those updates modify the existing classes distributed as a deployment kit. For each updated class, a new version is created. Every time an instance of an old version is used, a new instance of the new version is created, its state is transferred to the new object, and the reference is updated. Object migration is controlled by a class that is included in the deployment kit.

In (Malabarba, 2000), class structures are dynamically modified, by changing the implementation of the JVM and creating a new `ClassLoader`. That new class loader provides the dynamic loading of modified classes, replacing the existing ones (a functionality that is not included in the standard JVM). The instances of the adapted classes can evolve in three different ways: no instance is modified, some of them are (depending on user-defined criteria), and all of them are adapted.

The following works provide some runtime adaptability with frameworks, without modifying the JVM. Pukall *et al.* propose unanticipated runtime adaptation, adapting running programs depending on unpredictable requirements (Pukall, 2008). They propose a system based on class wrappers and two roles: *caller* (service clients) and *callee* (service providers). A callee and class wrapper that provides runtime adaptation. They provide services to access the original class. The implementation of those services are changed using the `instrument` Java 5 package. The callers are aimed at replacing invocations to an object with invocations to the appropriate callee wrapper.

DUSC (Dynamic Updating through Swapping of Classes) is a technique is based on the use of proxy classes, requiring no modification of the runtime system (Orso, 2002). As in the previous paragraph, the main Java technology used is HotSwap to change method implementation at runtime. DUSC performs the static modification of classes to allow its later adaptation (making them *swapping-enabled*). They allow adding and deleting classes, but modified ones must maintain their interface (`private` methods and fields can be modified). Another noteworthy limitation is that non-public fields cannot be accessed from outside the class.

Rubah is another framework for the dynamic adaptation of Java applications (Pina, 2013). When a new dynamic update is available, they load the new versions of added or changed classes at runtime, and perform a full garbage collection (GC) of the program to modify the running instances. The JVM is not modified. Instead, they implement an application-level GC traversal using reflection and some class-level rewriting. To update an application with Rubah, the programmer has to specify the update points, write the control flow migration, and detail the program state migration.

JRebel is a tool to skip the time-consuming build and redeploy steps in the Java development process,

allowing programmers to see the result of code changes instantly, without stopping application execution (JRebel, 2017). Modified classes are recompiled and reloaded in the running application. JRebel allows changes in the structure of classes. Classes are instrumented with a native Java agent using the JVM Tool Interface, and a particular class loader. Each class is changed to a master class and different support anonymous classes that are dynamically JIT compiled (Kabanov, 2017). JRebel does not check that the whole application has no type errors. Thus, application execution crashes when changes in a class imply errors in a program (e.g., a method is removed and it is later invoked).

MetaML is a statically typed programming language that supports program manipulation (Taha, 2000). It allows the programmer to construct, combine and execute code fragments in a type safe manner. In this way, dynamically evaluated programs do not produce type errors. MetaML does not support the manipulation of dynamically evaluated code; i.e., evaluation of code represented as a string, unknown at compile time. Therefore, its metaprogramming features cannot be used to adapt applications to new requirements emerged after their execution.

5 CONCLUSIONS

The proposed library shows how, with the existing standard Java elements, it is possible to include structural intercession and dynamic code evaluation services in the standard Java platform and language, modifying neither of them. Although the runtime adaptation mechanism proposed has an execution performance penalty, the system has been designed to reduce it when the JVM reaches a steady state after application adaptation. These metaprogramming services bring Java closer to the runtime adaptability of dynamic languages, without losing the benefits of its static type system.

We have a running proof-of-concept prototype, which successfully executed all the examples shown in this article. Currently, it requires the use of Java source code, and its runtime performance has not been heavily optimized.

We plan to added services for allowing the runtime adaptability of class hierarchies. Then, apply heavy optimizations to make its steady state execution time close to Java. The last step of the project is to allow the dynamic adaptation of running applications that have been modified and recompiled. The objective of this last step is not only to adapt single classes but also whole applications.

ACKNOWLEDGEMENTS

This work has been funded by the European Union, through the European Regional Development Funds (ERDF); and the Principality of Asturias, through its Science, Technology and Innovation Plan (grant GRUPIN14-100). We have also received funds from the Banco Santander through its support to the Campus of International Excellence.

REFERENCES

Andersson, J., Ritzau, T., 2000. *Dynamic code update in JDrums*. In Proceedings of the ICSE'00 Workshop on Software Engineering for Wearable and Pervasive Computing.

Andersson, J., 2000. *A deployment system for pervasive computing*. In International Conference on Software Maintenance. Proceedings, pp. 262–270.

Conde, P., Ortin, F., 2014. *Jindy: a Java library to support invokedynamic*. Computer Science and Information Systems 11(1), pp. 47-68.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Professional.

Garcia, M., Ortin, F., Quiroga, J., 2016. *Design and implementation of an efficient hybrid dynamic and static typing language*. Software: Practice and Experience 46(2), pp. 199-226.

Georges, A., Buytaert, D., Eeckhout, L., 2007. *Statistically rigorous Java performance evaluation*. In Object-Oriented Programming Systems and Applications, OOPSLA '07, NY, USA, pp. 57–76.

JavaParser, 2017. *Process Java code programmatically*. http://javaparser.org

JRebel, 2017. *Zero Turnaround JRebel, Reload code changes instantly*. https://zeroturnaround.com/software/jrebel

Kabanov, J., 2017. *Reloading Java Classes 401: HotSwap and JRebel — Behind the Scenes*. Zero Turnaround. https://zeroturnaround.com/rebellabs/reloading_java_classes_401_hotswap_jrebel

Malabarba, S., Pandey, R., Gragg, J., Barr, E., Barnes, J.F., 2000. *Runtime Support for Type-Safe Dynamic Java Classes*. In Proceedings of the 14th European Conference on Object-Oriented Programming, London, UK, pp. 337–361.

Meijer, E., Drayton, P., 2004. *Dynamic Typing When Needed: The End of the Cold War Between Programming Languages*. In Proceedings of the OOPSLA Workshop on Revival of Dynamic Languages.

Oracle, 2011. *JSR 292, supporting dynamically typed languages on the Java platform*. https://www.jcp.org/en/jsr/detail?id=292

Oracle, 2017. *function package, Java Platform SE 8.* https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Oracle, 2017. *instrument package, Java Platform SE 8.* https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html

Oracle, 2017. *Java Virtual Machine Support for Non-Java Languages.* http://docs.oracle.com/javase/7/docs/technotes/guides/vm/multiple-language-support.html

Orso, A., Rao, A., Harrold, M.J., 2002. *A technique for dynamic updating of Java software*. In Proceedings of the International Conference on Software Maintenance, pp. 649–658.

Ortin, F., Cueva, J.M., 2002. *Implementing a Real Computational-Environment Jump in order to Develop a Runtime-Adaptable Reflective Platform*. ACM SIGPLAN Notices 37(8), pp. 35-44.

Ortin, F., Cueva, J.M., 2003. *Non-restrictive computational reflection*. Computer Standards & Interfaces 25(3), pp. 241-251.

Ortin, F., Diez, D., 2005. *Designing an Adaptable Heterogeneous Abstract Machine by means of Reflection*. Information and Software Technology 47(2), pp. 81-94.

Ortin, F., Zapico, D., Cueva, J.M., 2007. *Design Patterns for Teaching Type Checking in a Compiler Construction Course*. IEEE Transactions on Education 50 (3), pp. 273-283.

Ortin, F., Conde, P., Fernandez-Lanvin, D., Izquierdo, R., 2014. *The Runtime Performance of invokedynamic: an Evaluation with a Java Library*. IEEE Software 31 (4), pp. 82-90.

Ortin, F., Labrador, M.A., Redondo, J.M., 2014. *A hybrid class- and prototype-based object model to support language-neutral structural intercession*. Information and Software Technology 56(2), pp. 199-219.

Paulson, L.D., 2007. *Developers Shift to Dynamic Programming Languages*. IEEE Computer 40(2), pp. 12–15.

Pina L., Hicks, M., 2013. *Rubah: Efficient, General-purpose Dynamic Software Updating for Java*. In the 5th Workshop on Hot Topics in Software Upgrades.

Polyglot, 2017. *A compiler front end framework for building Java language extensions*. https://www.cs.cornell.edu/projects/polyglot

Pukall, M., Kästner, C., Saake, G., 2008. *Towards Unanticipated Runtime Adaptation of Java Applications*. In 15th Asia-Pacific Software Engineering Conference, pp. 85–92.

Quiroga, J., Ortin, F., Llewellyn-Jones, D., Garcia, M., 2016. *Optimizing Runtime Performance of Hybrid Dynamically and Statically Typed Languages for the .Net Platform*. Journal of Systems and Software 113, pp. 114-129.

Redmond, B., Cahill, V., 2002. *Supporting Unanticipated Dynamic Adaptation of Application Behaviour*. In Proceedings of the 16th European Conference on Object-Oriented Programming, London, UK, pp. 205–230.

Redondo, J.M., Ortin, F., Cueva, J.M., 2008. *Optimizing Reflective Primitives of Dynamic Languages*. International Journal of Software Engineering and Knowledge Engineering 18(6), pp. 759-783.

Redondo, J.M., Ortin, F., 2013. *Efficient support of dynamic inheritance for class- and prototype-based languages*. Journal of Systems and Software 86(2), pp. 278-301.

Redondo, J.M., Ortin, F., 2015. *A Comprehensive Evaluation of Widespread Python Implementations*. IEEE Software 32(4), pp. 76-84.

Subramanian, S. Hicks, M., McKinley, K.S., 2009. *Dynamic Software Updates: A VM-centric Approach*. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, NY, USA, pp. 1–12.

Taha, W., Sheard, T., 2000. *MetaML and multi-stage programming with explicit annotations*. Theoretical Computer Science 248(1-2), pp. 211-242.

Würthinger, T., Wimmer, C., Stadler, L., 2010. *Dynamic Code Evolution for Java*. In Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, NY, USA, pp. 10–19.

Würthinger, T., Wimmer, C., Stadler, L., 2013. *Unrestricted and safe dynamic code evolution for Java*. Science in Computer Programming 78(5), pp. 481–498.