Orthogonal Persistence in Java supported by Aspect-Oriented Programming and Reflection

Rui Humbero R. Pereira
Instituto Superior Contabilidade e Administração do Porto
ISCAP/IPP
Porto, Portugal
rhp@iscap.ipp.pt

J.Baltasar García Perez-Schofield
Departamento de Informática
Universidad de Vigo
Vigo, España
jbgarcia@uvigo.es

Abstract — **The persistence concern implemented as an aspect has been studied since the appearance of the Aspect-Oriented paradigm. Frequently, persistence is given as an example that can be aspectized, but until today no real world solution has applied that paradigm. Such solution should be able to enhance the programmer productivity and make the application less prone to errors. To test the viability of that concept, in a previous study we developed a prototype that implements Orthogonal Persistence as an aspect. This first version of the prototype was already fully functional with all Java types including arrays.**

In this work the results of our new research to overcome some limitations that we have identified on the data type abstraction and transparency in the prototype are presented. One of our goals was to avoid the Java standard idiom for genericity, based on casts, type tests and subtyping. Moreover, we also find the need to introduce some dynamic data type abilities. We consider that the Reflection is the solution to those issues. To achieve that, we have extended our prototype with a new static weaver that preprocesses the application source code in order to introduce changes to the normal behavior of the Java compiler with a new generated reflective code.

Keywords-component; aspect-oriented programming; reflection; genericity, type inference; type erasure; framework;

I. INTRODUCTION

The initial version of the aof4oop [1][2] framework was already a fully functional system capable of providing Orthogonal Persistence [3] on any Java type, including arrays, to a software application. Using this framework prototype, the application can easily and transparently manage its data objects physically stored in a database. The current version only supports DB4Objects [4] as backend database and was not developed with performance goals.

This prototype, despite its limitations, demonstrated some of the advantages of Orthogonal Persistence in terms of productivity and final code quality. It implements the persistence concern as an Aspect, maintaining the application oblivious [5] of any technical details about the interaction with the database. It was designed to be totally reusable without the need of any kind of adaption on the code. The following code fragment shows the way an application can interact with data objects.

```
CPersistentRoot psRoot;
Student student;
Student student2;
Course course;

// get a persistent root (psRoot)
psRoot=new CPersistentRoot();

//Get one Student object from the psRoot (the database)
student=psRoot.getRootObject("rui");

//Get one Course object from the psRoot (the database)
course=psRoot.getRootObject("TO");

//Associate the persistent Student object with the persistent Course
course.addStudent(student);

// Instantiates a new Student object (still transient)
student2=new Student(1234,"Student Name","Student Address");

// Turns the student2 persistent simply because it is associated to another persistent object
course.addStudent(student2);
```

This form of persistence treats orthogonally all objects, following the three principles formulated by Atkinson and Morrison [3]. Their state persistence is only dependent if they are associated with another persistent object, or not, that is, if they are reachable by another persistent object [3].

In the next section we will briefly describe the system architecture of the developed framework.

In the section III we will discuss the implementation of Genericity in the Java platform and the drawbacks of the type erasure approach adopted in the version 5.0 of this platform. We will also debate the implications of the type erasure for the parametric polymorphism and the persistence.

Sections IV and V will present the recently improvements supported on Generics to the transparency and data abstraction of the prototype. We will also describe a new extension to our prototype that allows going even further into its orthogonality.

In the section VI we will debate the drawbacks of that improvement and finish this study with some conclusions.

II. THE PROTOTYPE FRAMEWORK

The developed framework provides orthogonal persistence services to an application that can easily and transparently manage its data objects physically stored in a database. Those persistence services were implemented as an Aspect in terms of Aspect-Oriented Programming (AOP) [6].

The AOP consists on a programming technique that allows the transversal separation of concerns. In an object oriented context, a concern that is transversal to all objects can be segregated from those objects and put in a specialized object called Aspect, while the remaining concerns, which are specific to each object, maintain themselves implemented in the object class.

The persistence is a concern that is transversal to any component (objects, functions or procedures) in the majority of the software. Due to that, it is frequently considered as a good example of crosscutting concern that can be aspectized. The main goal of the prototype is to test that concept.

The following diagram presents the system architecture.

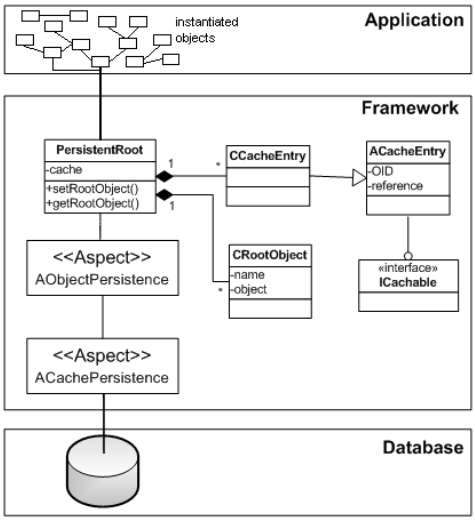

Figure 1- System architecture

III. GENERITICY

Since the objects obtained from persistent root `psRoot` can be from any type, the method `getRootObject(String rootName)` must return an object from the class `Object`. But when that return value is assigned to a Student, a Course or any other type of reference, the static type checking rules of the compiler requires a cast from `Object` to the actual type of the persistent object obtained from database.

Generics support came with the 5.0 version of the Java platform. This new feature was introduced to solve similar problems, such as type checking on data collections. Before support for generics appeared, the object references when put on a collection (an `ArrayList` for instance) were considered as `Object`, the top level class in the hierarchy of the Java classes. The types of those objects were not tested, so any type could be added to that collection. A cast is mandatory in the opposite phase, when any object is retrieved from that data structure. The compiler does not allow for the assigning of an object pertaining to a super class to a reference of a subclass.

This Java generic idiom [7], supported by the standard libraries based on casts, type tests and the `Object` class as a generic type, allows the programmer to deal with the data type generiticity, but the expressiveness and the type safety of the language is very compromised. Consequently, the presented framework in this work is also compromised.

A. *Parametric Polimorphism and type erasure*

With the Parametric Polymorphism [7] present in the Java 5.0, classes can be generic through types parameters. The instantiation code statement of a collection should use a parameter with a type but, as a result, those problems described above are solved. The compiler does consider that all objects on that collection pertain to a single type, the class specified in the type parameter. It will only accept that class of objects and when objects are retrieved it will not require any cast, because the compiler will insert it automatically. This new approach is based on a type erasure idiom [8] and frees the programmer of all those concerns. When the generic class is instantiated the compiler statically replaces (erases) that parametric data type by a raw data type, typically an `Object`. The compiler also introduces the necessary type checks at compile-time and uses bridge methods to ensure the type security of the retrieved objects.

The authors [7] of this approach justify their option, of using raw types, because serves two important purposes: the support of interfacing with legacy code, retrofitting all existing and used libraries in many production applications; and they support writing code in those few situations where it is necessary to downcast from an unparameterized type (like `Object`) to a parameterized type (like `ArrayList<A>`), and one cannot determine the value of the type parameter.

The adopted solution in the Java platform 5.0, as already described, allows a normal coexistence of non-generic and generic code. That it is achieved by the compatibility of

the binary class files that represents each class. That compatibility it a complex issue to solve that results of the multiple versions of structure that a polymorphic class may have, depending of the polymorphic parameters. That multiple representations of the same class are not suitable of being represented within a pure homogeneous translation [9][10], applied in the previous versions of the Java platform. Another considered alternative was the heterogeneous translation [9][10]. This one maps a parameterised class into a separate class for each instantiation. For example, the heterogeneous translation of the Pizza class `Hashtable<Key,Value>` replaces the instance `Hashtable<String,Class>` by the class `Hashtable$_String_$_Class_$` [9]. But, this other type of translation, besides of obvious disadvantages such as the extra needs of disk and memory space, is incompatible with class files structure used in the previous versions of the Java platform.

Despite of the advantages of the adopted solution, many authors are very critic about this implementation option arguing that compromises the type safety, the type orthogonality and others important characteristics of the Java language [11][12][13]. These criticisms are specially harsh in regard to implementation decisions (based in the type erasure idiom [8]) while the chosen syntax has been well received [11].

In the concern the persistence, the adopted approach compromises the implementation of Java orthogonal persistent systems [12][13][14][15]. The erasure process, which consists of eliminating the type parameters at the end of compile-time, affects Reflection on the parametric polymorphic classes since the type information is not available at run-time. As well known, the Reflection it is particular important to persistence mechanisms and database systems, moreover, the incorrect run-time type information also affects the Reachability of the objects [12]. If an object contain references to a `Collection<Person>` there is a risk of all that person objects to be stored as pertaining to class `Object`. And if a query is applied to those collection elements, looking for Person objects, it obtains an empty wrong result set, beside of require a casts.

B. Generic Methods

Parametric polymorphism is also applied on methods to provide them with Genericity. A given method is identified as generic if it declares one or more type variables (using the same syntax, as in parametric polymorphic classes with type parameters [8]). A type parameterized method has the ability of inference their return type value in same scenarios. Those characteristics have been explored in our prototype framework and it explained in detail in the next section.

IV. STEP FORWARD INTO TRANSPARENCY AND DATA ABSTRACTION

Recently, we have applied Genericity to the `CPersistentRoot` class of the developed prototyped. The `getRootObject(String rootName)` now it is a generic method that returns a generic type. That gives the opportunity to the compiler to decide, in each case, what kind of class the method effectively returns by doing type inference [16]. The underling process of calculating persistence closures continues to be the same one, based on subtyping. The added value is given by the support for generics. The following listing presents the signature of that method.

```
public <T extends Object> T
getRootObject(String rootName) {...}
```

This type of return value has a great importance, since it enhances the level of data type abstraction and transparency of the framework turning unnecessary any cast of data type.

That transparency is evident in the two lines of code below, where the same method of same object instance returns two distinct classes of objects.

```
...
student=psRoot.getRootObject("rui");
...
course=psRoot.getRootObject("TO");
...
```

This is achieved by the generic type being replaced by the correspondent type at compile-time. An implicit cast is actually taking place through type inference, by means of the generic return type to a specific type of object reference. This is happening in the example with the reference to a Student (student) and a reference to a Course (course). This process is similar to the one above with the generic classes. The return type of the method is erased and replaced by a raw type, commonly an `Object` class. In our prototype that type it is explicitly made by declaring `T` as a subtype of `Object`. At the end, the inference process does an automatic cast to the corresponding type of variable.

The prototype already provides a considerable level of transparency and orthogonal data type treatment, by freeing the programmer of doing casts and systematically data type tests, improving the data abstraction. With the obtained object reference the programmer can call all its methods or access all its properties. For instance:

```
student=psRoot.getRootObject("rui");

System.out.println("Street:"+student.getAddress().g
etStreet());
```

The object is pointed by a reference of the appropriate type (Student), thus all class structure is available to the compiler or the IDE allowing, for instance, auto-completion facilities.

A common procedure for any programmer is to avoid the splitting this code in two lines. Unfortunately, in this case everything changes, and type inference is not really possible.

```
System.out.println("Street:"+psRoot.getRootObject("
rui").getAddress().getStreet());
```

For a good understanding of what follows, this case will be identified as Case A.

As already explained above, the compiler infers the obtained reference through the program variable chosen by the programmer. But in this new situation, the compiler has no way to apply the inference algorithm and find what class the generic value pertains to. It is actually impossible at all to obtain that information at compile time. Only at run-time the system will be able to determine what is the class of the objects activated from the database. Since the generic method returns a generic type the compiler does subtyping and assumes that the result is an `Object` instance, where the `getAddress()` method does not exist. Because of that the result it is an illegal source Java code. The compiler gets an error while parses that source line.

We consider that the use of Reflection and a change on the normal behavior of the Java compiler, the best way to deal with this issue. An alternative reflective code must be generated at compile-time to serve as replacement code. This technique allows the access to the internal data class of the activated objects and the invocation of all it methods at run-time. That can be achieved by very different approaches, but all of them share the common goal of generate an alternative version of the code, this one already is legal from the point view of the compiler. For the given example, of this Case A, that alternative code could be similar to the following:

```
Object o1=psRoot.getRootObject("rui");

Object
o2=o1.getClass().getMethod("getStudent").invoke(o1,
(Object[])null);

Object
o3=o2.getClass().getMethod("getAddress").invoke(o2,
(Object[])null);

System.out.println(“Street:”+o3);
```

Type inference also does not work in a second kind of situations that we identify as Case B. This case prevents the method overloading to work properly at compile-time. Supposing that we a have a method called `printPersonalData(Student student)` that prints to the screen the student personal data. When using this method as presented below the compiler consider the argument an `Object` class.

```
printPersonalData(psRoot.getRootObject("rui"));
```

In those cases the compiler does not accept an `Object` in the place of a `Student` class, neither the overloading works correctly if exists another method with the signature `printPersonalData(Teacher teacher)`. Cabana, Alagic and Faulkner, have identified a very similar problem result of the type erasure on parametric classes [13]. As happens with Case A, in this case the code is also illegal to the Java compiler.

We also consider that this second problem also can be solved with Reflection. The process it is very simple and somewhat similar to the previous one, because at compile-time all this code situations are also replaced for another version of code. This alternative code test the generic object returned, determining if there is any method with correct signature for the corresponding argument class. As already explained, only at run-time it is possible to know what is the class of the generic object, so the method overloading must occur at that moment.

Our proposal to meet a solution to those two cases is a preprocessor extension to the Java compiler and the prototype, applying code manipulations in order to turn possible the compilation, extending the Genericity of the language and the framework prototype in the direction to dynamic typing.

V. EXTENDING THE PROTOTYPE

Our most recent research was an extension to the Java compiler thorough a preprocessor that parses the source code identifying all statements where the data type of the retrieved object could not be inferred at compile-time. The applied techniques on the preprocessor are presented in this section.

A. Method Genericity (Case A)

For the case A described above, we consider that can be easily achieved by a searching in the source code for any direct method call from the `CPersistentRoot` instance object. For each point on the code we replace the nested method calls for a special method that accepts the generic object as an argument and invokes all each methods in a Reflective way. This replacement process already was explained above. That special method is rendered at compile-time and stays as a private method of class where the occurrence happens. The following fragment of code shows how the problem identified it is handled by our framework.

```
System.out.println(“Street:”+_aof4oop$0$getAddress$
getStreet(psRoot.getRootObject("rui"));
```

The name of this method it is obtained by the result of the concatenation of all nested method names. If a same method sequence occurs again, with different arguments, another method with a different version number it is created.

B. Method overloading(Case B)

At present time this preprocessor already handle with the first described case (A) and we are working on new developments to resolve the second kind of situations (B). The algorithm that is being applied is similar and already was described with some detail above. For the given example the actual called method will be the following one:

```
private void _aof4oop$printPersonalData(Object arg1)
throws Exception
{
   if(arg1==null)
     throw new NullPointerException();
   else if(arg1 instanceof Student)
     printPersonalData((Student)arg1);
   else if(arg1 instanceof Teacher)
    printPersonalData((Teacher)arg1)
   else
     throw new ClassCastException(“No such method”);
}
```

Besides of the simple working principle, this case B raises some complex implementation requirements. The

algorithm of overloading method inference must be able to deal type all variety of method signatures requiring a very sophisticated parsing process. In this example the implementation it is quite simplistic, but in other studied examples it is not.

C. Static Weaver

Analyzing both implementations, A and B cases, we can conclude that they follow the same basic principles and techniques of the Aspect-Oriented Programming (AOP) turning this new extension of the prototype, as all the rest of the system, an aspect-oriented component. The rendered method it can be considered as an Advice [6] and the locals in the code where they are invoked are Pointcuts [6]. Due to that, the preprocessor works as a static weaver [6] that necessary applies the dynamic data type mechanisms at run-time. With this new static weaver (the preprocessor) we have bridged a gap that exists in all studied AOP frameworks. In any one of them we do not find any syntax of Pointcut Expression [6] capable of answer to all requirements of the presented problem. Because the advantages to dynamic weaving, this gap it is exacerbated by the actual tendency in all AOP frameworks of apply the aspects at load-time or run-time in the byte-code after the compilation. It must be noted that in our two cases, where the reflective code must be injected, the source code at beginning is not even legal for the compiler. As final result, the prototype now has two weavers: a static one that modifies the type checking rules of the Java compiler; and another one that provides the application with persistence services.

D. Side effects

Our approaches, to solve the case A and B, apparently have two drawbacks. The first is the disturbance on the error exception handling. If an exception occurs within the code that was replaced (the Advice) the stack information it not correct, because will give information of code that not exists from the programmer point of view. But, this problem it is already known in aspect-oriented environments, because happens every time that strange code (the aspect code in the Advice) is also injected in an application. The second drawback is the performance penalty introduced with the code generated by the framework. This second one it is inevitable and must be considered as necessary consequence of the needs of dynamic type behavior.

VI. Potential Risk of the Total Data Type Abstraction

Besides the two drawbacks already exposed, we consider the proposed level of data type abstraction and dynamic type achieved by the compiler extension reduces the type safety granted by the Java language by the absence of static type checking mechanisms. By requiring an explicit cast of returned value the programmer takes full consciences that object is of some specific type. And most important, the correct method invocation syntax can be statically checked at compile-time. That can anticipate numerous possible run-time errors to the compiler-time.

Considering the facts, it is questionable if that level abstraction, enabling the dynamic typing in Java, by changing the normal behavior of the compiler, is actually desirable. Naturally, we argue that should be the programmer to decide, as happens in some other program languages, if should apply the type inference and if it is decidable or undecidable.

VII. Related Research

In the early versions of the Java Language the lack of parametric polymorphism led to intensive research [7][9][14] [15][17][18] to find solutions to that problem. But the adopted solution [7][8] was not consensual. Several researchers have studied the same problem and the pros and cons of the solution adopted based in type erasure.

Cabana *et al.* [13] have studied the limitations of the type erasure and have find several pitfalls: violations of Java Type System; violations on subtyping rules; problems on method overloading and on the Java Core Reflection (JCR). To address those problems they proposed a technique mainly based on the representation of the parametric class or interface in the standard class file format with some subtle changes on: Java class files – introducing optional fields without affecting the compatibility with older versions, since those are ignored on a legacy JVM; extending the JCR to be able of obtain information about the type parameters; Modifications on the class loading process.

The relevance of the above work and others [11][12] [15] is about the concern of orthogonal persistence on parametric polymorphic classes that compromises our future work. This problem was already described above.

On specific concern of the persistence implemented as aspects, other research works also have applied AOP to provide applications with persistence. Soares et al [19] present their experience while refactoring a web application, a Health Watcher system, modularizing all code related with distribution and persistence concerns in AspectJ aspects. On our opinion, this work was limited to apply commons persistence design patterns with AOP.

Rashid *et al.* [20] has an interesting work that really present the persistence concern as an aspect, describing an aspect-oriented framework for persistence. This solution, by using the Reflection capabilities and a specialized aspect for translation Object-Relational, frees the programmer from doing any mapping from objects in memory to their related tables on the relational data base.

Kienzle and Guerraoui [21] made a detailed study about the aspectazition of transactions and failures, within persistence context, classifying that goal in three levels of different ambition of aspectization.

VIII. Future Work

Our prototype aims to treat the persistence in an orthogonal form. Currently, two of the three Atkinson principles are compromised, since the parametric classes are

not correctly stored in a database, breaking the Type Orthogonality and, consequently, the reachability.

This work presents the `CPersistentRoot` object that provides persistence services on the prototype. Those services, at current version, do not include any transaction capabilities. Future work will use important Kienzle and Guerraoui [21] work results.

As already referred above, our prototype use an object oriented data base. Considering the actual importance of the relational databases at the performance level and because they have a considerable market share, the prototype must be able to use them as information repository in order apply our framework on a real life production system.

IX. CONCLUSIONS

The prototype Aspect-Oriented Framework for Orthogonal Persistence (aof4oop) presents a high level of data type abstraction and access transparency, and reduces the database impedance mismatch with programming language.

Those characteristics were enhanced through the introduction of changes on the normal compiler behavior. To achieve that goal, a static weaver was developed based on a preprocessor, and is now part of the prototype.

The generics in Java 5.0, despite all the limitations universally acknowledged, have contributed to the enforcement of the type safety on our prototype, avoiding the use of the Java standard generic idiom. However, the Java parametric polymorphism does not provides a satisfactory solution to the issue of orthogonal persistence [12]. As a result, our prototype suffers from the consequences of the fact that type erasure does not allow a fully type orthogonality in the concern of persistence of parametric class instances.

REFERENCES

[1] . http://www.iscap.ipp.pt/~rhp/aof4oop/.

[2] Pereira R & Perez-Schofield J. An aspect-oriented framework for orthogonal persistence. In *Information systems and technologies (cisti), 2010 5th iberian conference on*. 2010.

[3] Atkinson M & Morrison R. Orthogonally persistent object systems. *The VLDB Journal* (1995) **4**: pp. 319-402.

[4] Paterson J, Edlich S, Hörning H & Hörning R. The Definitive Guide to db4o. . Apress, 2006.

[5] Filman R & Friedman D. Aspect-Oriented Programming is Quantification and Obliviousness. (2000) : .

[6] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J & Irwin J. Aspect-oriented programming. In *Ecoop*. 1997.

[7] Bracha G, Odersky M, Stoutamire D & Wadler P. Making the future safe for the past: adding genericity to the Java programming language. *SIGPLAN Not.* (1998) **33**: pp. 183-200.

[8] Gosling J, Joy B, Steele G & Bracha G. Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)). . Addison-Wesley Professional, 2005.

[9] Odersky M, Runne E & Wadler P. Two Ways to Bake Your Pizza - Translating Parameterised Types into Java. In *Selected papers from the international seminar on generic programming*. 2000.

[10] Odersky M & Wadler P. Pizza into Java: translating theory into practice. In *Proceedings of the 24th acm sigplan-sigact symposium on principles of programming languages*. 1997.

[11] Radenski A, Furlong J & Zanev V. The Java 5 generics compromise orthogonality to keep compatibility. *J. Syst. Softw.* (2008) **81**: pp. 2069-2078.

[12] Alagić S & Royer M. Genericity in Java: persistent and database systems implications. *The VLDB Journal* (2008) **17**: pp. 847-878.

[13] Cabana B, Alagić S & Faulkner J. Parametric polymorphism for Java: is there any hope in sight?. *SIGPLAN Not.* (2004) **39**: pp. 22-31.

[14] Solorzano JH & Alagić S. Parametric polymorphism for Java: a reflective solution. *SIGPLAN Not.* (1998) **33**: pp. 216-225.

[15] Alagić S & Nguyen T. Parametric Polymorphism and Orthogonal Persistence. In *Proceedings of the international symposium on objects and databases*. 2001.

[16] Milner R. A theory of type polymorphism in programming.. *Journal of Computer and System Sciences* (1978) **17**: pp. 348-375.

[17] Agesen O, Freund SN & Mitchell JC. Adding type parameterization to the Java language. *SIGPLAN Not.* (1997) **32**: pp. 49-65.

[18] Bank JA, Myers AC & Liskov B. Parameterized types for Java. In *Proceedings of the 24th acm sigplan-sigact symposium on principles of programming languages*. 1997.

[19] Soares S, Borba P & Laureano E. Distribution and persistence as aspects. *Softw. Pract. Exper.* (2006) **36**: pp. 711-759.

[20] Rashid A & Chitchyan R. Persistence as an aspect. In *Proceedings of the 2nd international conference on aspect-oriented software development*. 2003.

[21] Kienzle J & Guerraoui R. **AOP: Does It Make Sense? The Case of Concurrency and Failures**. In Lecture Notes in Computer Science. Magnusson B (Ed.). Vol. 2374.2006.