Adaptable Separation of Persistence Attributes employing Computational Reflection

ABSTRACT

The separation of concerns principle is aimed at the ability to modularize separately those different parts of software that are relevant to a particular concept, goal, task or purpose. Appropriate separation of application concerns reduces software complexity, improves comprehensibility, and facilitates concerns reuse. Considering persistence as a common application concern, its separation from program's main code implies that applications can be developed without taking persistence requirements into consideration. As a result, persistence aspects may be plugged in at a later stage. This separation offers the developer handle persistence software attributes regardless the application functionality. We have analyzed different approaches to accomplish a complete separation of persistent features, appreciating that computational reflection achieves an entire transparency of persistence concerns, offering an enormous adaptability level. We present the implementation of a research-oriented prototype that illustrates how computational reflection can be used in future persistence systems to completely separate and adapt application persistence attributes at runtime.

1. Introduction

Persistence capabilities are usually granted to applications by the use of explicit access to database management systems, such as object-oriented databases or object-relational mapping products. The usual way of building applications is tangling application functional code with explicit SQL or OQL persistence statements. This tangling of the source code of different concerns causes many drawbacks: legibility, maintainability and portability of the source code, lack of persistence functionality reuse, and low adaptability of persistence attributes.

The Separation of Concerns (SoC) principle emerges in order to overcome these common drawbacks of the software development lifecycle [1]. The objective of the SoC idea is to separate crosscutting concerns, such as persistence, from the main application code. The code that addresses each concern will not be spread out over different parts of the application, separating the main application algorithms from special purpose concerns. Following this technique, persistence requirements of any application could be plugged in a later stage into the application code, once the business logic has been specified. The source code of the main program will stay unmodified regardless of its persistent features.

In this article we analyze existing approaches to obtain a full separation of the persistence concern –from dominant persistent-application development to aspect oriented programming, including the orthogonal persistence approach. The most advanced ones, as well as the one presented in this paper, are currently in a research stage. In the era in which software engineering and networking capacity are becoming ubiquitous, research into new methods of software engineering such as concern (aspect) oriented software development has appeared. This is a research paper that shows how computational reflection is a suitable technique to be applied in the research field of dynamic separation of orthogonal software properties, taking persistence as the main example.

2. Dominant Persistent Application Development

Nowadays, the dominant persistent data model in the enterprise remains the relational model, represented in practice by the SQL language. Taking the Java platform as an example, the programmer interfaces with SQL either directly or indirectly. She could use SQL directly employing JDBC or SQLJ. On the other hand she may access persistent data through some object-relational mapping software (e.g., Sun's Java Blend or Sybase's CocoBase) or by means of a framework such as the Enterprise JavaBeans architecture.

Another approach to obtain data persistence is based on file persistent storage. Following with Java as an example, this platform has included an object serialization technology. The eXtensible Markup Language (XML) has recently become popular as a common framework for file formats.

When the programmer selects an object-oriented programming language, the requirement to map to SQL or XML to make an object graph persist is an added burden both during development and deployment, causing a considerable runtime cost to the application –e.g. requiring a parser to translate XML documents into objects and the other way round. Besides this impedance mismatch [2], these approaches require the programmer to write code sentences explicitly in order to make objects persist. The main reason of this lack of transparency is that programming languages and database management systems have historically evolved separate from one another, producing substantial differences between their computational models. As we will show in this paper, performing calls to a persistence system interface inside the language's computational model will make possible both the separation and the adaptation of application's persistence attributes at runtime.

3. Orthogonal Persistence

A step forward in achieving transparent persistence has been the appearance of orthogonal persistence systems in the 90s [3]. The aim of orthogonal persistence is to provide a single, uniform, computational model for all aspects of an application that deals with long-lived data. This capability is defined by three principles:

- Type orthogonality: All data objects should be allowed the full range of persistence, irrespective of their type.
- Persistence independence. The form of a program is independent of the longevity of the data it manipulates.
- Persistence by reachability. The lifetime of each object is determined by reachability from a set of root objects.

The two first principles compose the objective of a completely transparent persistence system: the programming language should not distinguish persistent objects from the transient ones, regardless of its type.

The third rule specifies a mechanism to implement transparent persistence. Persistence by reachability is focused on establishing the persistence concern in the own application's source code. The transitive closure of a persistent root object involves a quite transparent mechanism. However, if we think of persistence as a common application concern that may be separated from the program's logic, persistence by reachability would be only a possibility in a range of implementations. Different criteria (such as only identifying a specific set of persistent objects) might be necessary, depending on the persistence requirements of specific programs.

There exist different examples of orthogonal persistence systems. In the Java world, PJama and PEVM are two well-known implementations of the Object Persistence Java platform (OPJ) [3]. The main drawback of existing implementations is that the persistence by reachability rule makes them not fulfill the first criterion of persistence independence [2]. Persistence is not taken into account as a completely separate concern, regarding the SoC principle.

As an example, to make a collection of objects persist in PJama/OPJ, the programmer should explicitly program the following steps in the application's source code:

1. Take the persistent store by means of the `PjavaStore.getStore()` invocation.
2. Try to get the collection of objects from the persistent store (calling the `getRoot` method), indicating its persistence identifier.
3. If the object does not exist in the store (an exception is thrown), it should be introduced. Therefore, it is created in memory and included in the storage (using the `newPRoot` method) specifying its unique identifier.

Once the collection has been made persistent, the rest of the application logic is transparent to its persistence settings. However, the retrieval and storage of root objects need to be managed explicitly in PJama/OPJ. Apart from the lack of separating the persistence concern, existing orthogonal persistence systems do not offer adaptation of different features such as security or concurrency.

4. Persistence in Aspect Oriented Programming

Aspect-Oriented Software Development (AOSD) is a promising discipline that follows the SoC principle at any stage of the software lifecycle. AOSD is an evolution of the Aspect Oriented Programming (AOP) [4]. AOP is an implementation technique that provides explicit language support for modularizing application aspects: functionality that cuts across the system in a modular way. It allows the developer to design a system out of orthogonal concerns and providing a single focus point for modifications [1].

In the AOSD literature, persistence is often described as a classical candidate for aspectization [5]. Theoretically, it should be possible to:

- Modularize persistence as an effective aspect, employing AOP techniques.
- Reutilize persistence aspects, independently of the kind of application.
- Develop programs unaware of the persistent nature of its data.

Analyzing different implementations of persistence aspects, we realize that the previous goals are not easily achieved in real world examples. As a first example, PersAJ [6] provides a prototype to store aspects in an object-oriented database. In order to keep the persistence model independent of a particular AOP approach, an aspect is used to describe the persistence representation of aspects. Its aim is to provide a model for aspect persistence, but application data and persistence code were not separated. On the other hand, Kielze and Guerraoui [7] provided an assessment of AOP based on separating concurrency control and failure handling code in a distributed system. However, they investigated a case study on aspectizating transactions, only one facet of persistence. Modularization of code dealing with storage and retrieval of application data was not dealt with in detail. Another study has been performed trying to develop a persistence system with AspectJ (an aspect-oriented extension to Java) [5]. Their conclusion was that the development of persistence aspects and applications could not be done independently one of each other. Storage and update of persistent data does not need to be accounted for, but retrieval and deletion must be explicitly considered.

Therefore, the existing aspect tools do not seem to be really suitable for developing persistence aspects, following the main aim of the SoC principle. In contrast to the techniques analyzed in this paper, we will show how reflection is a more suitable technique to transparently separate any persistence concern.

5. The nitrO Reflective System

Reflection is the capability of a computational system to reason about and act upon itself, adjusting itself to changing conditions. Its computational domain is enhanced by its own representation, offering its semantics and structure as computable data.

There exist different levels of reflection [8], many of them used to obtain persistent features. Introspection is the lowest level or reflection: it permits the access to system structure, but not its modification. Introspection is offered by many programming languages (e.g., Java, C# and, in a very limited way, C++). The introspective runtime type information of C++ was employed in the development of Texas Persistent Store, offering a compile-time introspective virtual memory. Although Texas provides high performance, its persistence settings cannot be replaced or adapted at runtime, and they are completely monolithic.

Structural reflection is the second level of reflection, where the structure of the system may be dynamically altered. Python and Smalltalk are examples of languages that offer this kind of features. These reflective capabilities have been used to serialize objects just before they are stored, and the other way around.

Finally, computational reflection means dynamic customization of system structure and semantics. An example is the modification of the message-passing mechanism and objects lifetime at runtime, in order to update objects in a database every time their state is modified. Java has introduced in its version 1.3 a new dynamic proxy API inside its reflection package. This API offers a limited computational reflection service (that must be specified at compile time): the modification of single-class method invocation. When the programmer creates an object in a specific way, Java funnels all its method calls to an invocation handler. Applying this facility and Java introspection, the Hibernate Object/Relational mapping library translates the use of persistent objects into the underlying relational database model –a XML document must specify the mapping. Although Hibernate offers an efficient persistence layer between the application and the database, it does not reach the transparent separation of the persistence concern achieved with orthogonal persistent.

Meta-Object Protocols (MOPs) is the most extended mechanism employed to obtain runtime computational reflection. However, they basically have two drawbacks: they offer a too limited set of primitives to develop highly adaptable systems, and all of them use a fixed programming language. That was the reason why we developed nitrO, a non-restrictive computational-reflective system [8]. It offers more adaptability than existing MOP systems and it is language neutral –i.e. it can be programmed in any programming language. It was developed in the Python 2.2 programming language.

The nitrO reflective platform was designed following the theoretical definition of reflection [8]. This definition considers that a reflective computation is a computation about the computation, i.e. a computation that accesses the interpreter (what is call *reification*). Therefore, applications running over nitrO can access its interpreter at runtime, modifying their structure and customizing their language semantics. In this way, we have developed a generic interpreter capable of interpreting any programming language by previously reading its specification. The generic interpreter is language-independent: its inputs are both the user application and the language specification.

Programming languages are detailed in nitrO with language specification files, employing a top down parsing mechanism similar to the one used by the JavaCC tool. The lexical and syntactic features are expressed by means of context-free grammar rules; the semantics with Python code action routines, placed at the end of each rule [8]. We have specified Python, ECMAScript and a subset of the Java programming language. Although our first implementation requires the source code of each application, we are currently specifying new intermediate language grammars such as JVM bytecodes or MS.NET PE formats.

At runtime, any application may access language specifications by using the whole expressiveness of a meta-language: the Python programming language. Opposite to conventional reflective platforms, there are no previously specified restrictions imposed by a meta-object protocol –any feature can be adapted. Runtime changes to language specifications are automatically reflected on the application execution, because the generic interpreter relies on the language specification while the application is running.

As a simple example of a reflective application in the nitrO platform, we show a Java program where an `Author` object is created (Figure 1.a). In an infinite loop, the author's name is displayed on the console. Every application identifies its programming language previously to its source code (second line of Figure 1.a). When the application is about to be executed, its respective language specification file is analyzed and translated into an object representation in memory. Then, the generic interpreter, following the language specification, will execute the application.

```
Application = "Author"
Language = "Java"

class Author {
  String firstName, surname;
  Author(String firstName,String surname) {
    this.firstName = firstName;
    this.surname = surname;
    }
  String getFirstName() {return firstName;}
  String getSurname() {return surname;}
  void show(Console console) {
    console.print( "Author: “ +
            this.getFirstName() );
    console.print(" " + this.getSurname());
    console.println('.');
  }
  static void main(String[]args) {
    Author author = new Author(
            "Oscar", "Wilde" );
    Console console = new Console();
    while (true)
      author.show(console);
  }
}
```

a) Author Java program source code

```
reify <#

from objs import *
authorApp = nitrO.apps['Author']
authorInterpreter = authorApp.
      applicationGlobalContext['theInterpreter']
symbolTable = authorInterpreter.getSymbolTable()
authorInstance = symbolTable.getVar('author').
      getInstance()
authorClass = authorInstance.getClass()

nameInstance=authorInstance.fields[authorClass]
      ['firstName'].instance.setValue('Edgar')
surnameInstance=authorInstance.fields[authorClass]
      ['surname'].instance.setValue('Poe')

authorClass.addField(Jfield
      ('middleName',stringClass,''))
stringObject = stringClass.newInstance()
stringObject.setValue('Alan')
authorInstance.fields[authorClass]
      ['middleName'] = stringObject
#>
```

b) Reflective code modifying the Author program

Figure 1. Two nitrO example applications.

Another program may customize author application properties, by running reflective code. This feature is offered by the `reify` statement that the generic interpreter automatically recognizes. Regardless of the programming language, the reflective system identifies the `reify` statement, obtains the Python code located inside of it, and evaluates it at the same level as the generic interpreter. The reflective application may access both the internal structure of the authors program and its language semantics, achieving the dynamic adaptation of the application by means of computational reflection.

Figure 1.b shows an example of part of a reflective application that modifies the structure of the author program. Python code inside the `reify` statement may access any application running in the system, using the `nitrO` global object. This object is the system's Facade. Figure 1.b shows how we could obtain the author application, its symbol table, the author instance, and its class. Then we modify its two attributes (`firstName` and `surname`).

Finally, the reflective application modifies the structure of the authors program: a new attribute `middleName` is added to both the author class and its single object. The structures of the author object and class have been customized. In order to keep the example compact, the reflective application does not modify the `show` method displaying the new `middleName` attribute. This can be easily done, because statements in a method are treated as string data at the meta-level.

Applications are launched and inspected by means of the nitrO shell: a graphic window that interprets a reduced command language based on Python. When a program is executed, nitrO creates a new graphic window. After having executed the author program (Figure 1.a), we could adapt it by running the `reify` statement of the second program shown in Figure 1.b. When this second program is launched, the former will be dynamically adapted by the latter without changing its source code –in our example, the description of a new author is shown in its graphic window.

A common issue to take into account in reflective systems is security. Modifying one application's structure from another program requires security control. Currently, there exist runtime security models (e.g., Java security policy or .NET code-access security) that offer a rich set of permissions to configure many policy levels, including reflection. Following versions of our system will include the .NET code-access security system to grant reflective permissions to nitrO applications.

6. The nitrO Persistence System

Employing the reflective capabilities of nitrO, we have developed a persistence system in order to obtain a complete separation of the persistence concern. The system design is composed of three main subsystems:

1. Application. This package offers the representation of every running program (its classes, methods, objects and so on). It can be reused independently of the language selected, whenever the language to be supported is an object-oriented one. Changing its structure at runtime implies structural reflection of the program being executed.
2. Interpreter. It is responsible for performing the contextual analysis and application execution. The interpreter developed was a subset of Java Programming Language –the main simplification was the elimination of primitive types, in order to simplify the implementation. Its dynamic customization involves computational (semantics) reflection.
3. Persistence. This is the main package that offers the language neutral persistence system. Using reflection, this subsystem gives the programmer the ability to dynamically customize the reflective features of any application in a transparent way. Its design has been performed taking into account that different storages, indexing mechanisms and update policies could be used and dynamically replaced.

The system has been developed at the same level as the generic interpreter –i.e., at the meta-level, using Python. Its code employs the reflective capabilities of the system, adapting the semantics and structure of running programs. It can make a program persistent without changing its source code.

6.1. Interpreter Subsystem

The nitrO system takes the specification of the Java Programming Language and automatically generates the parse tree of the application to be executed. Then, nitrO executes (following the Command design pattern) the semantic rule specified at the end of the first syntactic production. This process returns the program's Abstract Syntax Tree (AST), a simplification of its parse tree.

The interpreter takes the program's AST and performs its interpretation. The interpretation mechanism is based on performing different decorations of the AST, following the Visitor design pattern. The `parse` method takes an AST, analyzes the node structure and calls the appropriate `visit_xxx` method –there are as many `visit` methods as syntactic constructions in the Java language. Following this scheme, semantic analysis, application representation, and program execution is performed. Therefore, the system may obtain computational reflection by modifying the `visit` methods of the execution visitor.

6.2. Application Subsystem

This package contains the classes that represent a Java application at runtime. Classes (`JClass`) are made up of fields (`JField`), methods (`JMethod`) and constructors (`JConstructor`); the two last elements are grouped by `JMethodGroup` instances. `JRef` denotes a reference to an instance.

One important thing of this module is that it has been designed indicating the interface that should be implemented to make an element persist. Implementing the `Instance` interface, any object could be persistent. The methods located in this interface (`makePersistent`, `makeTransient`, `store`, `restore` and `getID`) are the ones that the persistence subsystem employs to manage object persistence. In our design, only objects are persistent because classes (code) are directly stored in the file system.

6.2.1. Persistence ID

The creation of a unique persistence identifier (ID) of every element to be stored is a common issue that persistence systems have to deal with. As application objects are going to survive to program execution, references to them (their memory addresses) will not be valid. Therefore, we must assign a unique global ID to any object.

We have designed the persistence system requiring that any persistent element should return its ID at its `getID` method invocation. The `JInstance` implementation returns the concatenation of the following values: the IP address, the PID of the process, the UID of the user, the TID of the active thread, and milliseconds went by from January the 1st 1970. We have implemented a large persistence ID trying to avoid any possible collision, taking into account that different storages and applications might be running.

6.3. Persistence Subsystem

Figure 2 shows the persistence subsystem. The `Manager` class is the Facade of the module and it has been implemented with a Singleton instance. It will offer persistence facilities to the programmer. The behavior of the persistence system could be adapted at runtime by the selection of specific `Storage` and `StoragePolicy` instances.

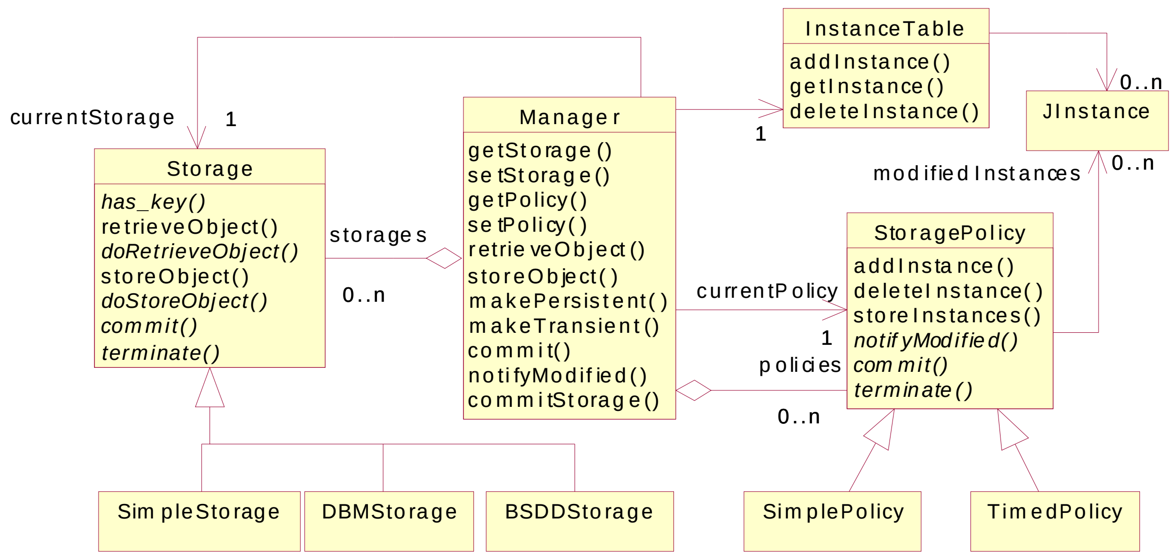


Figure 2: Persistence subsystem class diagram.

Different update policies and storage systems can be employed in the platform. The `Storage` and `StoragePolicy` abstract classes are partial implementations offered by the framework, facilitating the addition of new elements. Storages are different ways to keep information persistently in addition to its indexing mechanisms; policies specify the way objects should be updated into the storage selected. Runtime selection and swap of this two variables could be performed in a programmatically way.

We have implemented three reference storages: a dictionary (`SimpleStorage`); linear hash, B+tree and variable-length record storages, offered by the Berkeley DB Library (`BSDDBStorage`); and a Unix-based `(n)dbm` library whose objects behaves like mappings (`DBMStorage`).

Two different policies to update persistence objects (calling the storage `commit` method) have been developed: whenever a persistent object is modified a specified number of times (`SimplePolicy`), and every time a timer reaches a configurable number of seconds (`TimedPolicy`). Each of these parameters can be modified at runtime depending on runtime requirements –as well as exchanging the policy being used.

6.3.1. Instance Storing

In the storages implemented, we have used the reflective `pickle` Python module to serialize objects, i.e. converting any object to a stream of bytes and vice versa. Although this module marshals any Python object, it does not handle the issue of naming persistence objects. So, we have defined our own system of persistent object IDs (Section 6.2.1). The process of converting persistent object IDs to memory references is called *pointer swizzling*; the converse operation is sometimes termed *unswizzling*.

The persistence `Manager` implements a lazy (un)swizzling mechanism. In the unswizzling case, the reference translation is performed when the object is about to be stored. If the object has references to other persistent objects, these will be also translated following the same recursive scheme. This process is performed in parallel with object serialization.

The reverse mechanism (swizzling) is performed in two steps. The object demanded is searched in the storage at first, using its persistence ID. In this step, the stream of bytes is retrieved and converted into a Python object. Afterwards, the reference swizzling is performed, recovering memory links between objects.

This process is achieved by means of the `InstanceTable` shown in Figure 2. This table is a Python's weak dictionary that establishes a mapping between persistence IDs and their respective memory references. Any time an object is set as persistent, an entry is assigned in this table. Therefore, acting as a cache, if a persistent object is needed and it has an entry in this table, its associated instance will be used.

Notice that this table uses weak references: if the persistent object is no more referenced, the garbage collector might discard it. When a persistent object is reclaimed and it has not an entry on the `InstanceTable`, the `Manager` will recover it from the storage registered.

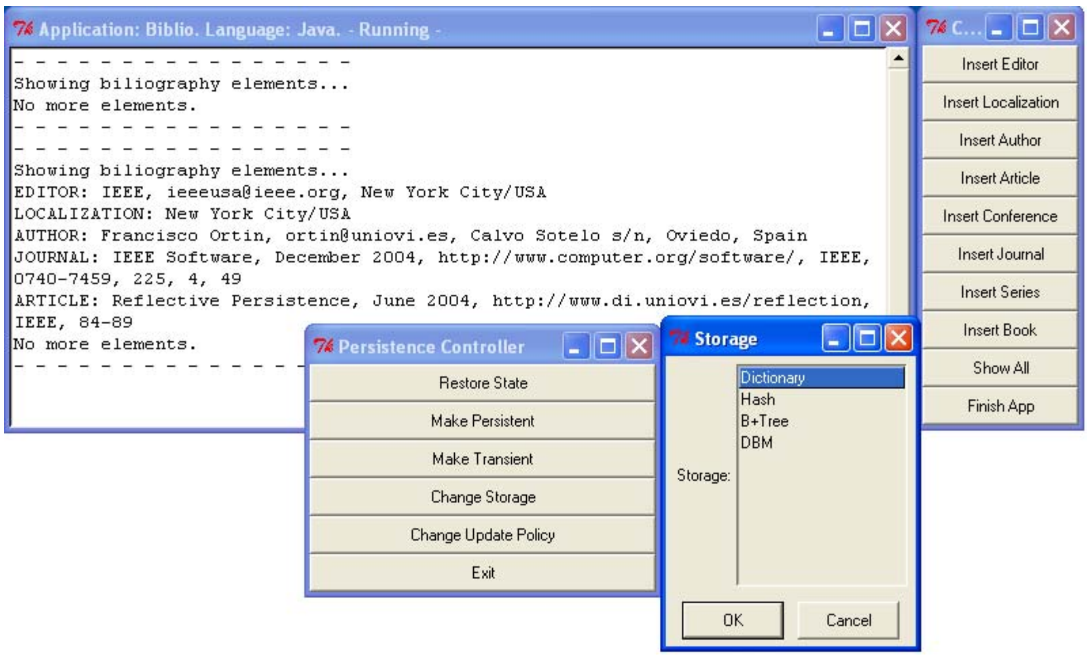
7. A Sample Bibliography Application

We have developed an example bibliography application derived from information stored on the DBLP server (http://dblp.uni-trier.de/). The program manages a set of bibliography items (journals, series, conferences, books and articles), publishers, locations, authors and editors. It has been implemented in Java and it is no persistent at all –once the application finishes, the object collections are released.

Apart from the bibliography application, we have developed a reflective persistence controller separating the application's persistence concern. This second program assigns and modifies the persistence features of the bibliography application, using reflection. It has been developed at the meta-level (using Python code inside `reify` statements). By means of a graphic menu, if offers the user the ability to make the bibliography program persistent or transient. Moreover, it permits the modification of the persistence storage, update policy and indexing mechanism employed. The reflective code accesses the persistence manager of the `Biblio` application the same way as shown in Section 5. Then, it achieves the customization of the program's persistence settings by invoking methods of the persistence manager.

Figure 3 shows an example scenario where bibliography objects have been recovered from a previous execution. The two upper windows in Figure 3 show the bibliography application execution with its corresponding graphic menu. The first display of existing bibliography elements shows that the collections are empty (upper left window).

Using the nitrO shell, the Persistence Controller program is launched (two lower windows in Figure 3), managing the persistence features of the bibliography system. Selecting the *Restore State* option of the persistence controller, the main application indirectly retrieves a set of persistent objects from a previous execution. Then, if the user asks for existing bibliography items, the list of elements in Figure 3 is shown. In addition, we can make the application persistent with the controller, obtaining a transparent synchronization between the program's data and the persistent storage. Note that the bibliography application has no specific code to manage the retrieval and storage of data; that task is performed by the persistent system. Figure 3 also shows how storages (lower right window) and updating policies can be dynamically changed.



```
- - - - - - - - - - - - - - - -
Showing biliography elements...
No more elements.
- - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - -
Showing biliography elements...
EDITOR: IEEE, ieeeusa@ieee.org, New York City/USA
LOCALIZATION: New York City/USA
AUTHOR: Francisco Ortin, ortin@uniovi.es, Calvo Sotelo s/n, Oviedo, Spain
JOURNAL: IEEE Software, December 2004, http://www.computer.org/software/, IEEE,
0740-7459, 225, 4, 49
ARTICLE: Reflective Persistence, June 2004, http://www.di.uniovi.es/reflection,
IEEE, 84-89
No more elements.
- - - - - - - - - - - - - - -
```

Figure 3: The bibliography application and its separate persistence controller execution.

The example presented shows how a user may adapt application features at runtime, without modifying the application's source code in a single statement. We have achieved a separation of the persistence concern, employing reflective techniques. The persistence system is also adaptive: it could be customized in a programmatical way, depending on variables such as system load, persistence level, the number of connected users, or even the structure of running applications.

The complete separation of application logic from persistence concerns brings new possibilities to develop different kinds of persistence applications. Persistence facilities could be offered to the user in different ways: as a graphical browser, as a domain-specific programming environment, or as a complete transparent orthogonal persistence system. For instance, following the orthogonal persistence point of view, new optional items describing the application persistent attributes (storage, indexing mechanism and update policy) could be specified in addition to the initial indications of programming language and application identifier, offering a fully transparent orthogonal persistence.

8. Runtime Performance

The main disadvantage of dynamic application adaptation is runtime performance. The basic performance limitation of our reflective platform is caused by the interpretation of every programming language. Nowadays, many interpreted languages are commercially employed –e.g. Java, Python or C#– due to optimization techniques such as just-in-time (JIT) compilation or adaptable native-code generation. In the following versions of the nitrO platform, these code generation techniques will be used to optimize the generic-interpreter implementation. As we always translate any language into Python code, a way of speeding up application execution is using the interface of a Python JIT-compiler implementation.

9. Conclusions

This research paper presents a prototype implementation illustrating how computational reflection represents a suitable technique to achieve dynamic and transparent adaptation of application's persistence attributes, following the SoC principle. The use of reflection to obtain transparent management of persistent features has increased over time. We have shown how the highest level of reflection could be applied in future systems to obtain transparent and dynamically adaptive persistence. Since computational reflection offers the runtime customization of language semantics, it allows the combination of programming language and database management systems into one single computational model.

The Python platform, the persistence system, and the example code presented are available at http://www.di.uniovi.es/reflection/lab.

10. References

[1] W.L. Hürsch and C.V. Lopes: Separation of Concerns, Technical Report UN-CCS-95-03, Northeastern University, 1995.

[2] J. B. García Perez-Schofield, E. García, Tim Cooper, M. Pérez-Cota. Managing schema evolution in a container-based persistent system. Software, Practice & Expererience 32(14), 2002.

[3] M. Atkinson, L. Daynès, M. Jordan, T. Printezis and S. Spence: 'An Orthogonally Persistent Java', SIGMOD Record 25(4), 1996.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier and J. Irwin: 'Aspect Oriented Programming', European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1241, 1997.

[5] A. Rashid, R. Chitchyan: 'Persistence as an Aspect', International Conference on Aspect-Oriented Software Development, 2003.

[6] A. Rashid: 'On to Aspect Persistence', GCSE Symposium, Springer-Verlag LNCS 2177, 2000.

[7] J. Kielze and R. Guerraoui: 'AOP: Does it Make Sense? The Case of Concurrency and Failures', European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 2374, 2002.

[8] F. Ortin and J.M. Cueva: 'Implementing a Real Computational-Environment Jump in order to Develop a Runtime-Adaptable Reflective Platform'. ACM SIGPLAN Notices (37)8, 2002.