JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 8, September-October 2004

# Reflection as the Basis for Developing a Dynamic SoC Persistence System

**Benjamin Lopez, Francisco Ortin and Javier Noval**, Computer Science Department, University of Oviedo, Spain

**Abstract**

Persistence is a common application requirement that is usually taken into account when the program is being developed. Different emerging techniques following the Separation of Concerns principle are focused on detaching crosscutting concerns, like persistence, from the main application code. Although this is a profitable principle, existing tools lack two main features: runtime adaptability and language independence. This paper shows how computational reflection can be employed as a suitable technique to overcome the two previous limitations, offering dynamic adaptation of persistence features in a language independent way and achieving transparent separation of the application's persistence concern.

## 1 INTRODUCTION

Persistence capabilities are usually granted to applications by the use of explicit access to database management systems (DBMS), such as object-oriented databases or object-relational mapping products. Tangling application functional code with explicit SQL or OQL persistence statements makes up the final application.

A different approach is the support for persistent objects into object-oriented languages. Taking Java as an example, *PJava* (Persistent Java) [Atkinson 1996] provides a persistent programming environment for the Java programming language, based in an orthogonal persistent variant of the Java platform and machine. Other initiatives use persistent storage engines (like ObjectStore PSE or Jeevan Java Objects) offering an API to endow the programming language with persistence functionality.

Whichever the previous alternative we select, application development will suffer from the following drawbacks:

1. Legibility and maintainability. Since additional code not related to the application logic is tangled through the source code to have the added database functionality, legibility and maintainability of the source code suffers a fall.

Cite this article as follows: Benjamin Lopez, Francisco Ortin, Javier Noval: "Reflection as the basis for Developing a Dynamic SoC Persistence System", in *Journal of Object Technology*, vol. 3, no. 8, September-October 2004, pp. 121-145. http://www.jot.fm/issues/issue_2004_09/article2

2. Portability suffers as well. There is a direct dependence of the persistence mechanism explicitly in the application implementation. Changes in persistence issues, imply changing the application implementation.
3. Poor adaptability. Adaptation of persistence related aspects, such as adding a new indexing technique, are commonly made by changing and recompiling the source code. There is no possibility to adapt these features to runtime emerging requirements, unpredicted at design time.
4. Persistence functionality reuse. Commonly, similar fragments of code achieving the same functionality differing from data structure, implies redundant code not being refactored and reused. With the separation of persistence concerns and the use of reflection, this routines could be generic and, therefore, reusable.
5. Complexity. While the programmer is developing the application functionality, she has to make explicit calls to the introduced APIs and/or the extensions added to the programming language, not being possible to reason about application logic in isolation.

Aspect oriented software development [Kiczales 1997] is an innovative paradigm focused on obtaining Separations of Concerns (SoC) [Parnas 1972, Hürsch 1995] in software development, making possible to modularize crosscutting aspects of a system. Most existing aspect-oriented tools are language dependent and lack runtime adaptability –few offer runtime adaptation in a very limited way.

Following the SoC principle, we have developed a different approach to the task of adding persistence functionality to programming languages, which is based on the notion of employing language-neutral reflection. This means that the user does not need to take special action to make objects persist (no explicit tangled code is needed) and so, complexity, legibility and portability problems are not a concern.

We have implemented a reflective platform called nitrO [Ortin 2002] that, independently of the programming language selected by the programmer, offers a great level of runtime adaptability. Over this platform, we have developed a persistence framework that allows dynamic changes to persistence related aspects (for example, dynamic change of indexing techniques for a given application), not needing to specify it in the application's code. This application adaptation is performed at runtime, not needing to modify its functional code, and can be carried out in a programmatically way –i.e., the application itself, or another one, may change its persistence features at runtime.

The rest of the document is organized as follows. In the next section we present aspect-oriented programming and the main lacks of existing tools. Section 3 briefly describes current systems that use AOP to create persistence applications. Section 4 introduces the architecture of the nitrO platform, and the persistence system design is presented in section 5. A sample scenario demonstrating different system's capabilities is described in section 6. Finally, we analyze system benefits as well as its runtime performance (section 7), and section 8 presents the ending conclusions.

## 2 ASPECT ORIENTED SOFTWARE DEVELOPMENT

In many cases, significant concerns in software applications are not easily expressed in a modular way. Examples of such concerns are transactions, security, logging or persistence. The code that addresses these concerns is often spread out over many parts of the application. Software engineers have used the principle of SoC to manage the complexity of software development [Parnas 1972, Hürsch 1995]; it separates main application algorithms from special purpose concerns. Final programs are built by means of its main functional code plus their specific problem-domain concerns. Its main benefits are a higher level of abstraction, easier to understand the application's functionality, concern's code reuse, and the increase of application development productivity.

This principle has been performed following several approaches such as Composition Filters [Bergmans 1994], Multi-Dimensional Separation of Concerns [Tarr 1999]) or, the most extended and advanced one, Aspect Oriented Software Development (AOSD).

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) [Kiczales 1997].

AOP is an implementation technique that provides explicit language support for modularizing application concerns that crosscut the application functional code. Aspects express functionality that cuts across the system in a modular way, thereby allowing the developer to design a system out of orthogonal concerns and providing a single focus point for modifications. By separating the application functional code from its crosscutting aspects, the application source code would not be tangled, being easy to debug, maintain and modify [Parnas 1972].

### Dynamic Weaving

Most current AOP implementations are largely based on static weaving: compile-time modification of application source code, inserting calls to specific aspect routines. The places where these calls are inserted are called *join points*. The *aspect weaver* is the program that integrates aspects into the main application code. AspectJ [Kiczales 2001] is an example of a static-weaving aspect-oriented tool: a general-purpose aspect-oriented extension to Java that supports aspect-oriented programming.

It is commonly accepted to have preprocessor-like aspects weavers to interconnect functional code and aspect code. However, sometimes it is desirable to postpone the decision about whether aspect information is to be added to an application or not until runtime. For instance, one may have a huge resource-consuming image processing algorithm as part of an application and, depending on system load and available computing nodes, a trade-off between data distribution, the memory allocation scheme, and the utilization of computing power at runtime, has to be made [Schult 2002]. Both

memory allocation and calculation distribution are crosscutting concerns, but the selection must be performed at runtime by the application in a programmatically way.

Our previous example identifies a weakness of traditional approaches to aspect oriented programming. Typically, one has to decide at compile time whether an aspect should be interwoven or not. Besides, at runtime, one can neither unweave the aspect nor interweave another aspect with the application.

In order to overcome the static-weaving weaknesses, different dynamic-weaving approaches have emerged: AOP/ST [Böllert 1999], PROSE [Popovici 2001], Dynamic Aspect-Oriented Platform (DAOP) [Pinto 2001], Java Aspects Components JAC [Pawlak 2001], CLAW [Lam 2002] or LOOM.NET [Schult 2002] are different examples. These systems give the programmer the ability to dynamically modify the aspect code assigned to application join points. However, they offer a limited set of language join-points, restricting the amount of application features an aspect can adapt. For instance, PROSE cannot implement a post-condition-like aspect, since its join-point interface does not allow accessing the value returned by a method upon exit [Popovici 2001]. They have been used to develop different AOP programs, but the limited set of join-points they offer do not make them suitable for real-world persistence scenarios [Rashid 2003].

### Language Neutrality

Both static and dynamic weaving AOP tools do not offer the implementation of crosscutting concerns, regardless of the language the programmer might use. They employ fixed-language techniques to achieve separation of concerns.

We have identified computational reflection [Maes 1987] as the best technique to overcome the two previously mentioned limitations. In this paper, we present a reflective approach to develop a language-neutral dynamic-weaving persistence system.

## 3 THE BACKGROUND TO PERSISTENCE AS AN ASPECT

In the AOSD literature, persistence is often described as a classical candidate for aspectization [Mens 1997, Suzuki 1999]. 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 [Rashid 2000] 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 is not separated. On the other hand, Kielze and Guerraoui [Kielze 2002] 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 aspectizing 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 [Rashid 2003]. 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 Separation of Concerns principle. Apart from that, really flexible aspect tools that offer dynamic weaving are not available and all of them are language dependent. We will show how reflection is a more suitable technique for these purposes.

### Dynamic Adaptation of Persistence Issues

Apart from being able to dynamically make objects persist and give them back to its non-persistent state, it is interesting to adapt their persistent features in a programmatically way. Based on conditions arisen at runtime, an application could customize features such as the indexing mechanism or update policy employed.

Object-oriented persistence systems have special features to take into account: the existence of inheritance and aggregation hierarchies and the potential presence of method invocations. Thus, different indexing mechanisms are needed to allow an efficient processing of persistent data under these circumstances.

Many indexing techniques for object-oriented models have been proposed, which can be classified into structural and behavioral [Bertino 1995]. Depending on the most frequent type of query on a given class (or class hierarchy), some indexing techniques are more efficient than others. Therefore, the persistence system will allow the use of different indexing mechanisms as well as the dynamic selection of a specific one deemed as the most appropriate depending on the type of the class (or class hierarchy) in question.

Another important persistence variable is the update frequency. It is a common trade-off between safety and performance: the higher the update frequency, the lower the loss of data plus the worse performance –and vice versa. So, depending on situations detected at runtime, our persistence system could chose between system safety and performance.

As we will show afterwards, our system offers dynamic selection of the storage, indexing mechanism and update policy programmatically.

## 4 THE NITRO REFLECTIVE SYSTEM

The main technique we have used to achieve system goals is reflection. Reflection is the capability of a computational system to reason about and act upon itself, adjusting itself

to changing conditions [Maes 1997]. Its computational domain is enhanced by its own representation, offering its semantics and structure as computable data.

Although there exists many different classifications [Ortin 2003], we will just focus on runtime computational reflection: customization of system structure and semantics. An example is the dynamic modification of the message-passing semantics, in order to update objects in a database every time their state is modified.

Meta-Object Protocols (MOPs) is the most famous mechanism employed to obtain runtime computational reflection [Kizcales 1991]. However, they basically have two drawbacks: all of them use a fixed programming language, and they offer a too limited set of primitives to develop highly adaptable systems [Ortin 2002]. That was the reason why we developed nitrO, a non-restrictive computational-reflective system [Ortin 2002]. It offers much more adaptability than existing MOPs and is language neutral –i.e. it can be programmed in any programming language.

The theoretical definition of reflection [Smith 1982], considers that a reflective computation is a computation about the computation, i.e. a computation that accesses the interpreter (what is call *reification*). We have designed nitrO following this concept: if an application would be able to access its interpreter at runtime, it could modify its structure and customize its language semantics. In this way, we have developed a generic interpreter (Figure 1) capable of interpreting any programming language by previously reading its specification. This generic interpreter is language-independent: its inputs are both the user application and the language specification.

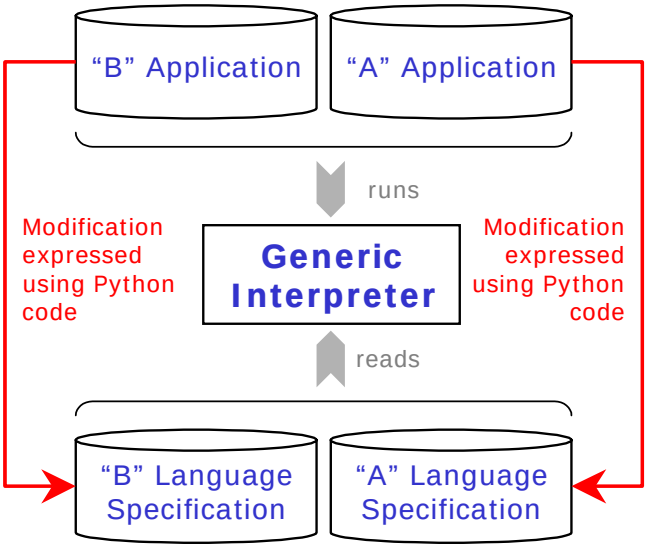
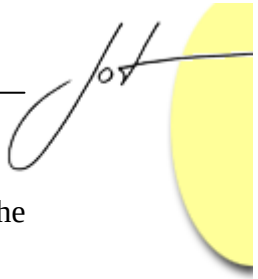



Fig. 1. Architecture of the nitrO system.

At runtime, any application may access language specifications by using the whole expressiveness of a meta-language: the Python programming language. 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. This feature is offered by the `reify` statement that the generic interpreter automatically recognizes.

This mechanism is language neutral. Any application, whatever its language would be, may access and adapt another program in a language independent way. The meta-language employed is always Python.

## Language Specification

Programming languages are detailed in nitrO with language specification files. Their lexical (`Scanner` section) and syntactic (`Parser` section) features are expressed by means of context-free grammar rules; their semantics, by means of Python code, placed at the end of each rule (between `<#` and `#>` characters).

We have specified Python and Java and some domain-specific languages. Currently we are specifying ECMAScript. Correctness verification (e.g., type checking) is expressed inside the semantic actions using Python code. The next specification is a first example of a *VerySimple* language definition without any semantic correctness verification:

```
Language = VerySimple

Scanner = {
  "Digit Token"
    digit -> "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"  ;
  "Number Token"
    NUMBER -> digit moreDigits  ;
  "Zero or more digits token"
    moreDigits -> digit moreDigits
          |  ;
  "Character Token"
    char -> "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|
            "l"||"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"w"|"x"|
            "y"|"z"  ;
  "Character or Digit Token"
    charOrDigit -> char | digit  ;
  "ID Token"
    ID -> char moreCharsOrDigits  ;
  "Zero or more chars or digits token"
    moreCharsOrDigits -> charOrDigit moreCharsOrDigits
          |  ;
  "SEMICOLON Token"    SEMICOLON -> ";"  ;
  "ASSIGN token"       ASSIGN -> "="  ;
}

Parser = {
  "Initial Context-Free Rule"
    S -> statement moreStatements SEMICOLON <#
              global vars
              vars={}
              nodes[1].execute()
```

```
                nodes[2].execute()     #>  ;
    "Zero or more Statements"
      moreStatements -> SEMICOLON statement moreStatements <#
                nodes[2].execute()
                nodes[3].execute()     #>
            |  ;
    "Statement"
      statement -> _REIFY_ <#  nodes[1].execute()  #>
            | assignment <#  nodes[1].execute()  #>
            | expression <#
                nodes[1].execute()
                write("Expression value: "+
                          str(nodes[1].value)+".\n") #>  ;
    "Assignment Statement"
      assignment -> ID ASSIGN expression <#
                nodes[3].execute()
                vars[nodes[1].text]= nodes[3].value    #>  ;
    "Binary Expr. Factor"
      expression -> ID <# nodes[0].value=vars[nodes[1].text] #>
            | NUMBER <# nodes[0].value=int(nodes[1].text) #>  ;
}

Skip = {"\t";  "\n"; " ";}
NotSkip = {  }
```

Every application must identify its programming language previously to its source code. 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.

The `_REIFY_` reserved word indicates where a `reify` statement might be syntactically placed. `Skip` and `NotSkip` sections tell the interpreter which tokens have to be automatically ignored and which ones should be appended to the scanner buffer.

## Application Execution.

Any application code starts with its unique ID followed by its language name. The next code is an example of a *very simple* application:

```
Application = "Very Simple App"
Language = "VerySimple"
   a=10;
   b=a;
   a;
   b;
```

The previous code is executed as it was specified in the *VerySimple* language: two assignments are performed and the respective values of `a` and `b` variables are written. The generic interpreter runs this code by executing its language semantics. However, using the `reify` statement, python code could be run at the interpreter level, accessing and modifying the application representation.

## Dynamic Adaptation

Independently of the language used, the generic interpreter automatically recognizes a `reify` statement. Inside a `reify` instruction Python code can be written. It will not be processed as the rest of the application code: it will be taken and evaluated at the same level as the interpreter process. This code, using Python structural reflection, may access and modify any application's symbol-table and language specification, achieving the theoretical definition of reflection: a computation that accesses its interpreter [Smith 1982].

The way Python code access any application running in the nitrO system (independently of its language) is by the `nitrO` global object. This is the system's Facade [Gamma 1994]. Its attribute `apps` is a hash table of existing applications in the system. Each application object has two main attributes (Figure 2):

1. Attribute `language`: Its language specification. Accessing this attribute, language semantics might be dynamically modified.
2. Attribute `applicationGlobalContext`: Its dynamic symbol table, which permits the programmer knowing and modifying any application's structure at runtime.

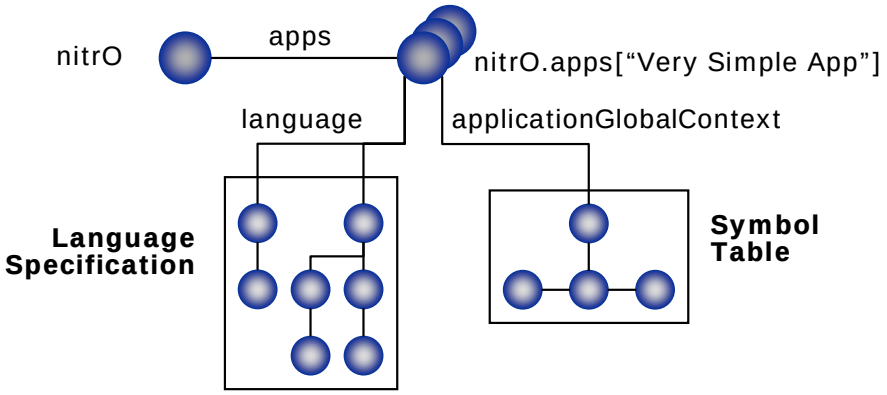



Fig. 2. Accessing nitrO applications by means of the nitrO object.

A `reify` sentence may dynamically access and modify the running application, no matter which program or language might be used to execute them. It can be executed by the previous *very simple application* or by another one to access the former.

```
reify <#
vars=nitrO.apps["Very Simple App"].
          applicationGlobalContext["vars"]
write( str(vars)+"\n" ) # Shows {b:10,a:10}
vars["a"]=vars["a"]*2 # Modifies "a"
vars["c"]=0 # Creates a new variable
del vars["b"] # Erases a variable
#>;
```

The code above takes the variables from the symbol table (accessing the `applicationGoblalContext` attribute), shows their values, modifies the value of one,

creates a new variable, and erases an existing one. Note that this code is executed at the system's meta-level.

As a second example, we may enhance the assignment-statement semantics by showing a trace message every time an assignment takes place. This is the reification code that accesses the application's language attribute:

```
reify <#
from langSpec import SemanticAction
assignment=nitrO.apps["Very Simple App"].
           language.syntacticSpec["assignment"]
code="write(\"Assignment of \"+nodes[1].text+\" with value \"+
      str(nodes[3].value)+\".\\n\")"
# Behavior adaptation
assignment.options[0].actions.append(SemanticAction(code) )
#>;
```

First, the assignment-statement syntactic rule is taken. Then, the code representing the new trace semantics is created, setting it to the `code` variable. Finally, the assignment semantics is enhanced in order to display a trace message. Once this code is evaluated, the *very simple application* will show a trace message whenever an assignment is made (i.e., reflection has taken place).

As a result, nitrO is a computation platform that uses a non-restrictive reflective technique; it can be programmed using any language; is completely adaptable at runtime, and has a great level of application interoperability.

## 5 THE NITRO PERSISTENCE SYSTEM

Once we have introduced a resume of the nitrO reflective platform, we are going to present the persistence system developed. Three main subsystems (shown in Figure 3) were employed on its design:

1. Interpreter: This is the unique module dependent of the programming language. It is responsible for performing the contextual analysis and application execution. In our implementation, we have developed an interpreter of a subset of the Java Programming Language –the main simplification was the elimination of primitive types, in order to simplify the implementation.
2. 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. It can be replaced with a new implementation by only developing a reduced interface.
3. Persistence: The main package. Offers the language neutral persistence system. Its design has been performed taking into account that different storages, indexing mechanisms and update policies could be used and dynamically replaced. Due to the use of reflection, the persistence mechanism is completely reusable regardless of application's structure.

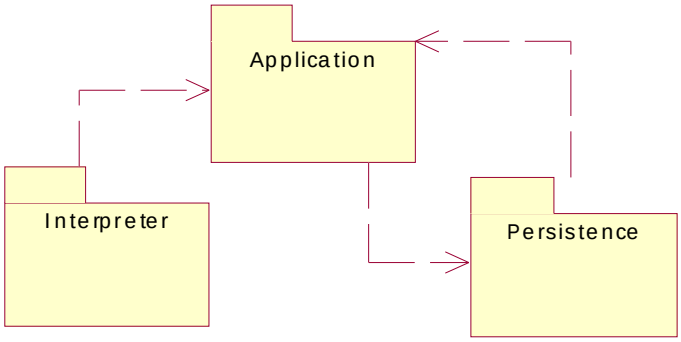



Fig. 3: Subsystems diagram of the persistence system.

## 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 [Gamma 1994]) 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 simple interpreter takes the program's AST and performs its interpretation. A reduced class diagram of the interpreter's design is shown in Figure 4. The interpretation mechanism is based on performing different decorations of the AST, following the Visitor pattern [Gamma 1994]. 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 (code generation into memory) and execution is performed.

At execution time, the interpreter context should be managed. It is composed of references to current class, instance, method and a stack of local references.

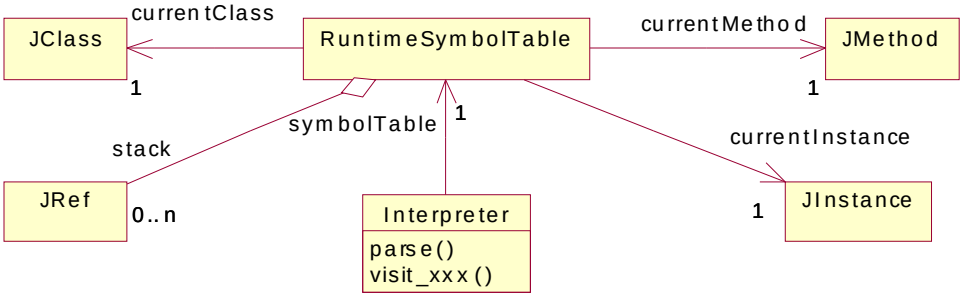



Fig. 4: Interpreter subsystem class diagram.

## Application Subsystem

Figure 5 shows the straightforward class diagram of the elements 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.

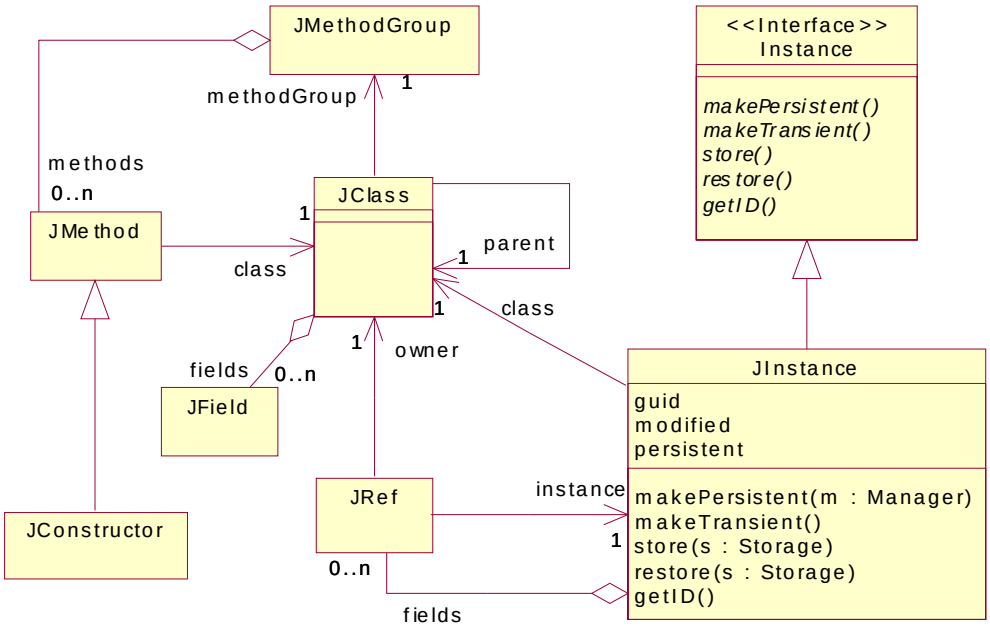



Fig. 5: Application subsystem class diagram.

One important thing of this module is that it has been designed indicating the interface that should be implemented to make an element persist, whatever its language would be. Implementing the `Instance` interface[1], any object could be persistent. In our design, only objects are persistent because classes (code) are stored in the file system. However, if we prefer a complete persistence system, we should implement these five operations in every class in Figure 5.

**Persistence ID**

A common persistence issue is the persistence ID of every element to be stored. As application's objects are going to survive to program execution, a reference to them (its memory address) will not be valid. Therefore, any object must have a unique global ID.

The persistence ID of an element should be returned 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 selected a complex reference implementation trying to avoid any possible collision, taking into account that different storages and applications can be used and the system could be extended to support distribution in the future.

[1] Python do not have interfaces, so any object that implements the five methods shown in the diagram would be capable of being persistent.

## Persistence Subsystem

Figure 6 shows the persistence subsystem. The `Manager` class is the Facade of the module and it has been implemented with a Singleton instance [Gamma 1994]. The behavior of the persistence system will be established by the selection of specific `Storage` and `StoragePolicy` instances.

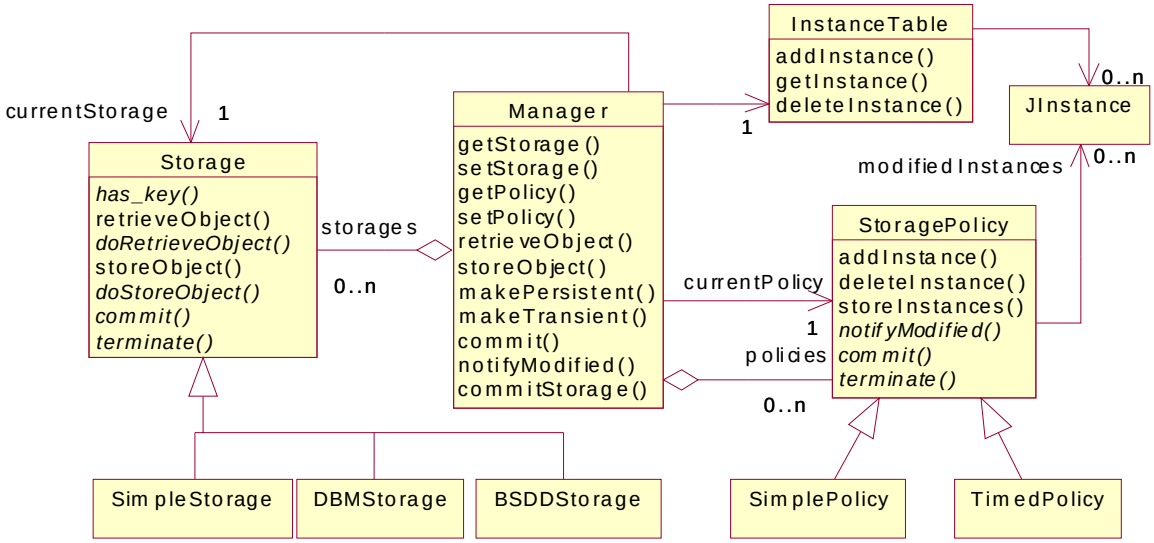



Fig. 6: Persistence subsystem class diagram

Different update policies and storage systems can be employed in the system. The `Storage` and `StoragePolicy` abstract classes are partial implementations offered by the framework to facilitate the addition of new elements. Storages are different ways to keep information persistently and its indexing mechanisms; policies are 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:

- `SimpleStorage`: It is a simple dictionary that is saved and load from a file. It is the default storage selected by the manager.
- `BSDDBStorage`: Provides access to Berkeley DB Library. The user can create extended linear hash, B+tree or variable-length record storages, depending on parameters passed to the `BSDDBStorage` constructor. This storage might be used to dynamically change indexing mechanism depending of runtime emerging requirements or even program's structure.
- `DBMStorage`: This storage offers a Unix `(n)dbm` library. `Dbm` objects behave like mappings, except that keys and values must be always strings.

We have also implemented two reference update policies:

- `SimplePolicy`: The update of the storage selected will be performed (called its `commit` method) whenever a persistent object has been modified a specified

number of times. This is the default policy, with only one modification needed to update each instance.

- `TimedPolicy`: A timer is employed to do the updating. The policy is parameterized with a number of seconds. When the timer reaches the number of selected seconds, a `commit` is performed: every persistent object that has been modified is updated on the storage.

Obviously, each of the parameters of the previous policies can be modified at runtime depending on runtime requirements –as well as exchanging the whole policy.

**Instance Storing**

In the storages implemented, we have used the `pickle` Python module to serialize objects, i.e. converting any object to a stream of bytes and back. 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 unique persistent-object IDs (section 5). The process of converting an object's reference to its corresponding persistent ID is called *pointer swizzling*; the converse operation is termed *unswizzling*.

The persistence `Manager` does the process of swizzling on the fly, meaning that the reference translation is performed when the object is about to be stored. Its fields, that are also persistent, are translated following the same scheme.

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

This process is achieved by means of an `InstanceTable` instance (Figure 6). 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. If a persistent object is reclaimed and it has not an entry on the `InstanceTable`, the `Manager` will recover it from the storage registered.

**Modification of a Persistent Instance**

In order to make possible the implementation of the update policy, the interpreter will have to notify the persistence manager of instances modification. This is performed by the `notifyModified` method, as shown by the sequence diagram in Figure 7. This way, the update policy will collect every modified instance in order to do the future storage as appropriate.

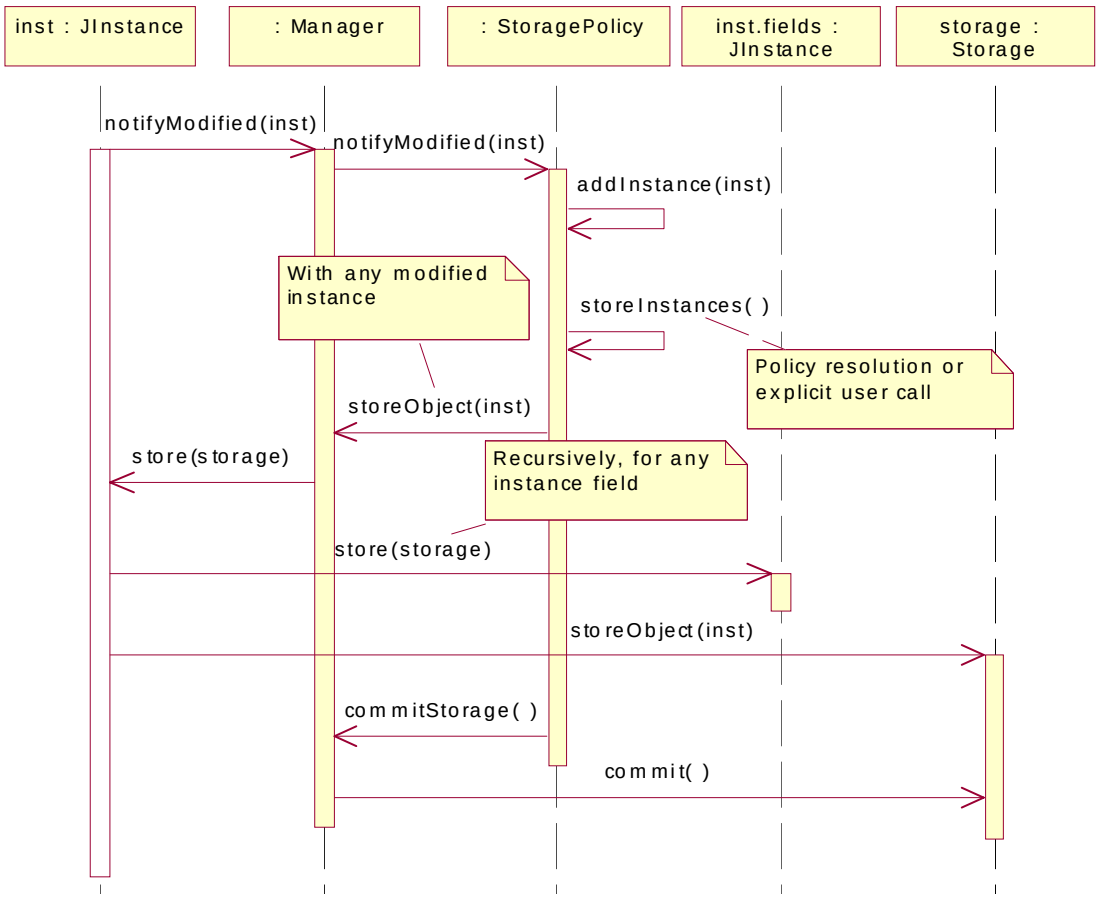



Fig. 7: Sequence diagram of the modification of a persistent instance.

Once the registered policy resolves to do the update (the user may also do it explicitly) its `storeInstances` method is executed. This call causes the invocation of the `storeObject` method of the manager as many times as modified instances have been collected. Then, each instance must prepare to serialization doing the swizzling as mentioned in the previous point –this produces recursive calls to each instance's field. Finally, once the modified instances have been swizzled in the `InstanceTable`, the commit operation will synchronize the table with the storage, making the modified instances persist by committing the corresponding update transaction.

## 6 A SAMPLE BIBLIOGRAPHY APPLICATION

Within this section we will present a sample bibliography application derived from information stored on the DBLP server [Ley 2003]. The data model, represented as a UML class diagram, is shown in Figure 8.

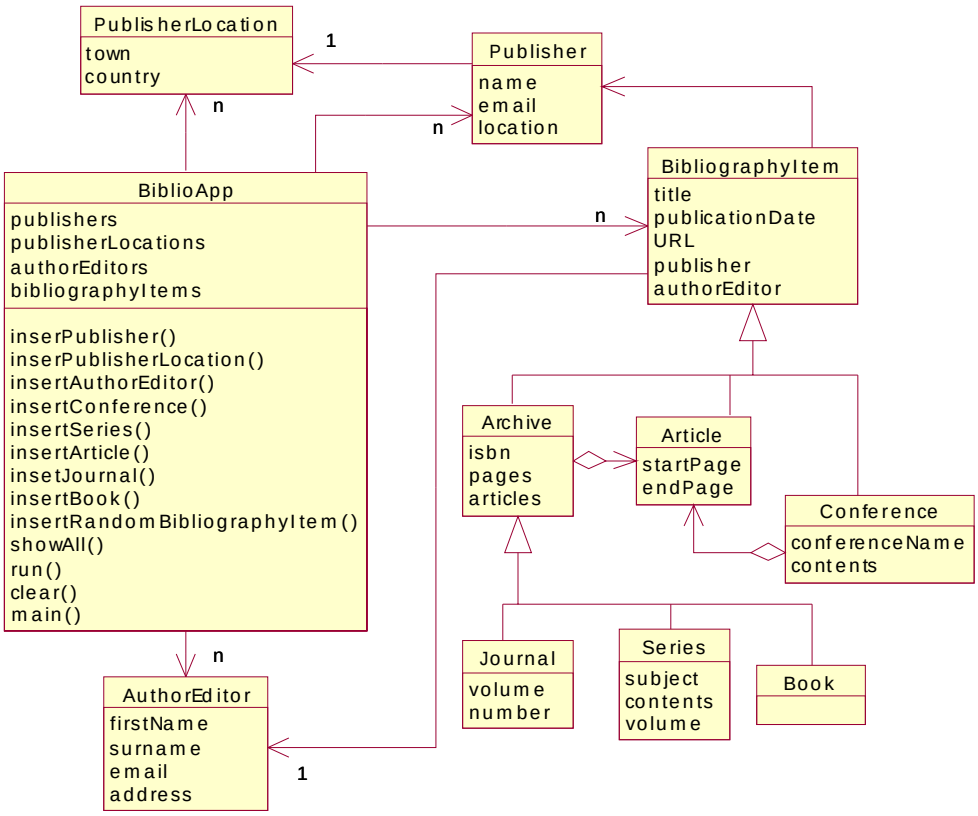



Fig. 8: Data model for the sample application.

The whole application has been developed in Java and it is no persistent at all. Its main class is `BiblioApp`, which randomly creates different bibliography elements, modifies all of them (by means of the `run` method) and, finally, restores its initial state (with the `clear` method). The following code is the `run` method of the `BiblioApp` class. It measures the time employed to insert and modify an established number of bibliography items. The `cons` reference represents the console of the application's graphic window.

```
Application = "Biblio"
Language = "Java"
...
class class BiblioApp {
...
  void run() {
    Integer insertions = 5000;
    Timer timer = new Timer();
    cons.println('Measuring '+insertions.toString()+
                 ' insertions...');
    insertPublisherLocation();
    insertPublisher();
    timer.start();
    for(Integer i = 0; i < insertions; ++i) {
      insertRandomBibliographyItem();
    }
```

```
    timer.stop();
    cons.println(insertions.toString() + ' insertions: ' +
                 timer.getTime() + ' seconds');
    cons.println('Measuring '+data.numItems.toString()+
                 ' updates...');
    timer.start();
    for(Integer i = 1; i <= data.numItems; ++i) {
      BibliographyItem item = (BibliographyItem)data.
           bibliographyItems.getItem('Title' + i.toString());
      item.setUrl('http://www.anotherURL.com');
    }
    timer.stop();
    cons.println(insertions.toString() + ' updates: ' +
                 timer.getTime() + ' seconds');
  }
```

## Persistence Assignment

This application is executed in the nitrO system with the *Biblio* ID. Its execution shows the time employed to insert and modify 5,000 instances stored in RAM. Once this application has been started, we may develop another program that, using the reflective persistence system, will set and dynamically modify persistent features of the *Biblio* application. Moreover, this new application will be capable of calling methods of the *Biblio* program, although they have been written in different programming languages.

The new program (called *Benchmark*) is going to execute nested loops in which different storages, update policies and indexing mechanisms will be assigned to the *Biblio* application. During each loop iteration, the benchmark calls the `run` and `clear` methods of the bibliography application causing insertion, modification and deletion of persistent objects with different persistence settings. The following code is a fragment of the benchmark application:

```
[1] reify <#
[2] import persistence

[3] try:
[4]   biblioApp = nitrO.apps['Biblio']
[5] except KeyError, e:
[6]   raise SystemExit('"Biblio" is not running.')

[7] biblioInterpreter=biblioApp.applicationGlobalContext
                      ['theInterpreter']
[8] biblioManager = biblioInterpreter.getPersistenceManager()
[9] biblioPrint=nitrO.apps["Biblio"].window.writeText
[10] symtable = biblioInterpreter.getSymbolTable()

[11] def setStorage(storage):
[12]   biblioManager.setStorage(storage)
[13]   print("Setting "+storage+" persistence storage")
[14]   biblioPrint("Setting "+storage+
                   " persistence storage\n")
```

```
[15] def setPolicy(policy):
[16]   biblioManager.setPolicy(policy)
[17]   print("Setting "+policy+" update policy")
[18]   biblioPrint("Setting "+policy+" update policy\n")

[19] appInstance = symtable.getVar('app').getInstance()
[20] appClass = appInstance.getClass()

[21] print("\nDYNAMIC PERSISTENCE BENCHMARK")
[22] biblioPrint("\nDYNAMIC PERSISTENCE BENCHMARK\n")
[23] for policy in application.policies:
[24]   print("\n------- "+policy+" Policy -------")
[25]   biblioPrint("\n------- "+policy+" Policy -------\n")
[26]   for storage in application.storages:
[27]     print("\n"+storage+" Storage:")
[28]     biblioPrint("\n"+storage+" Storage:\n")
[29]     makePersistent()
[30]     setPolicy(policy)
[31]     setStorage(storage)
[32]   appInstance.getClass().getMethod('run',()).invoke(
                    biblioInterpreter,appInstance,())
[33]   appInstance.getClass().getMethod('clear',()).invoke(
                    biblioInterpreter,appInstance,())
[34] #>
```

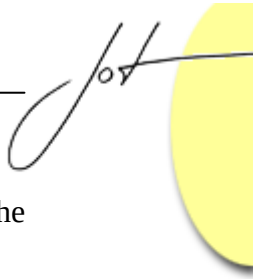
The previous code is a sole reify statement. Thus, it is executed at the interpreter level to directly access to the *Biblio* application. This is performed using the `nitrO` facade object (line 4), trying to get the object that represents the bibliography application. If the application is running, its interpreter (line 7), persistence manager (line 8), console (line 9) and symbol table (line 10) are also obtained.

To dynamically change persistence storages and update policies, the `setStorage` (line 11) and `setPolicy` (line 15) functions are defined. They tell the persistence manager which storage and policy must be set, showing a message in both the bibliography (`biblioPrint` function) and benchmark (`print` function) applications. Lines 19 and 20 take from the symbol table the `app` reference of the *Biblio* main method and its class: `BiblioApp`.

From line 23 to 33, we can see the nested loops previously mentioned. During each loop, one storage and policy are selected and the two `run` and `clear` methods of the `app` instance are called (lines 32 and 33). We have employed dictionary, DBM, hashing and B+Tree storages. The update policies used were each object's state modification, at 10 object modifications, each second and every 30 seconds.

## Benchmark Execution

Windows of each running application are shown in Figure 9. We can see the nitrO shell at the bottom of the picture: the main window where we can tell nitrO which applications must be executed. The application on the left is the bibliography program showing



insertion and modification times measured. Finally, the window on the right is the benchmark program that modifies the bibliography persistence settings.

Note that the first information shown by the bibliography application is insertion and modification in memory. The reason is that the benchmark had not been launched and *Biblio* was not persistent in the beginning. The following measurements are all persistent, caused by running the benchmark.

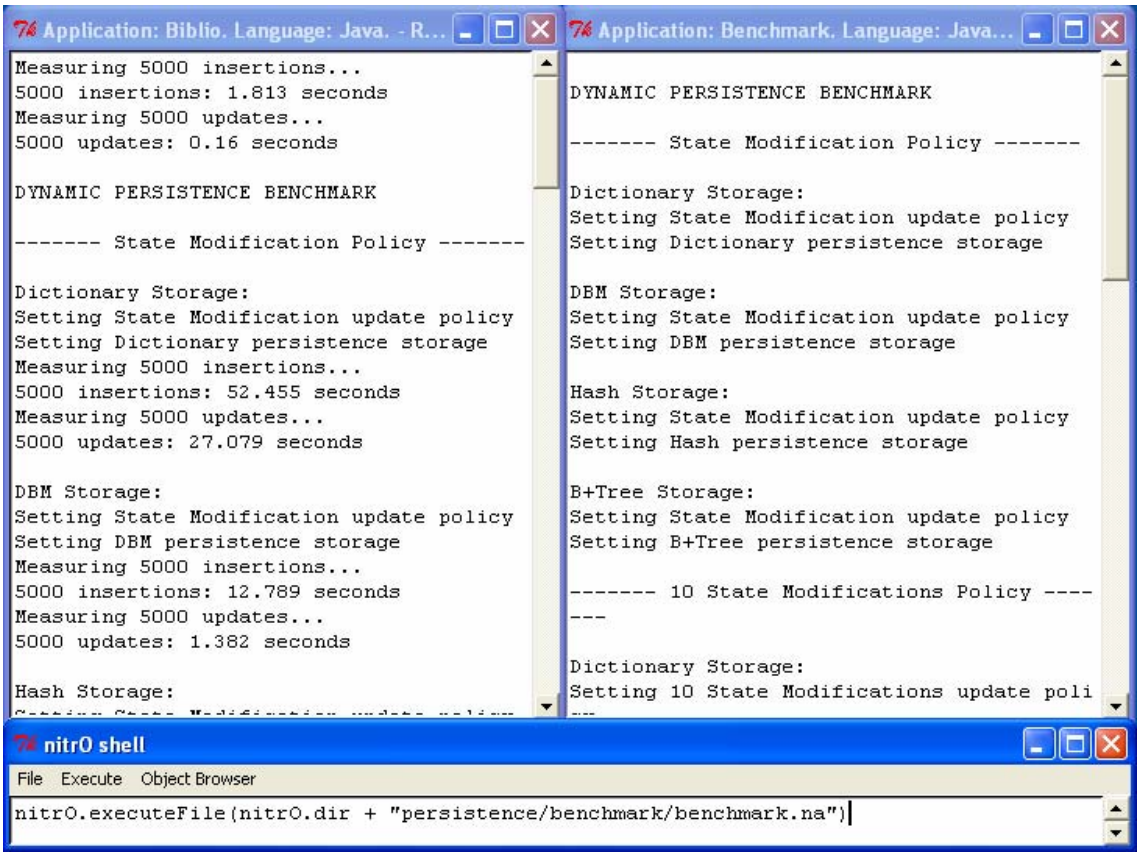



Fig. 9: Running applications in the nitrO system.

We have measured five different storages (including memory) with four policies. The number of insertions and modifications on each configuration was 5,000. All tests were carried out on a lightly loaded 1.0 GHz iPIII system with 256 Mbytes of RAM running WindowsXP. The metric employed was execution time. Figure 10 shows different measurements graphically.

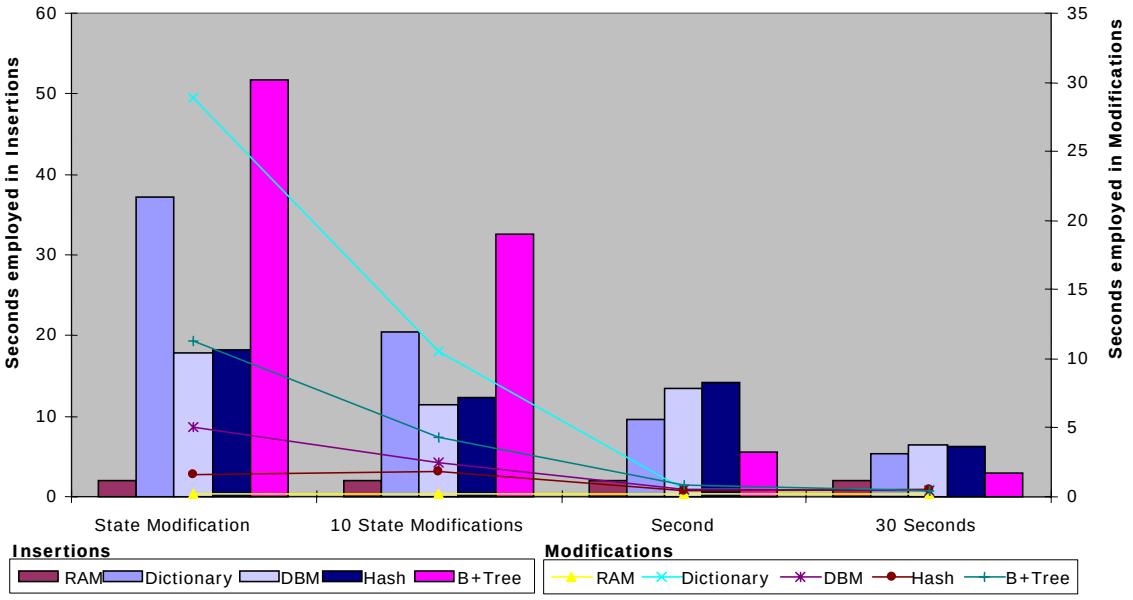



Fig. 10: Running the benchmark with 5,000 bibliography items.

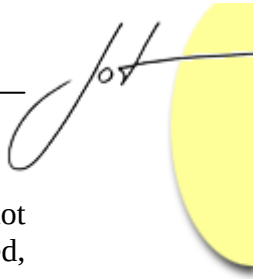
As an example of how the persistence system can be used to obtain persistence parameter tuning, a first analysis of the results obtained executing the previous benchmark shows that:

- The trade-off between safety and performance is confirmed: the higher update frequency, the worse performance (state modification is the most safe policy, but the slowest one).
- B+Tree storage is the fastest when we use timed policies, but employing the state modification ones, hash and DBM are better.
- Using state modification policies, insertions with B+Tree are the slowest. However, under the same conditions, its modifications are much faster than the dictionary ones.
- With a small number of bibliography items (up to 100), the best storage, independently of the policy employed, is the dictionary. However, when there exists a huge number of objects, its performance is disastrous.

Depending on variables such as system load, persistence level, number of connected users, or even application's structure, any (part of an) application could dynamically select the most suitable persistence configuration, in a programmatic way. Therefore, our persistence system is capable of adapting to contexts only known at runtime, offering the optimum parameter tuning.

## 7 SYSTEM BENEFITS

The most important feature of the persistence system is the higher degree of flexibility provided by reflection. Some advantages derived from this property are mentioned below.



- SoC benefits. Applications programming is simpler now, as programmers do not have to explicitly deal with persistence. Once the application has been developed, or even at runtime, the user will just need (if desired) to identify which objects should persist and its persistence features. The abstraction level is consequently raised and software development complexity decreases. This also implies better legibility and maintainability of applications.
  Another advantage on separating the functional code from its persistence aspect is that programs do not suffer from changes on persistence issues; its functional code remains the same.
- Dynamic selection of persistence. The persistence system offers the ability to set and unset persistent any runtime object. This feature can be assigned and erased at runtime. Moreover, the storage employed, the persistence level and the indexing mechanism, are features that may be selected and modified at runtime. All of them can be customized in a programmatically way.
- Reutilization. Using reflection, the system has been codified analyzing objects' structure at runtime. This way, the persistence framework makes objects persist and retrieves them later from the database, whatever its structure might be. This feature makes the persistent system highly reusable, independently of the functionality –and even the language– of the application stored.
- Transparency. The workings of the nitrO persistence system are transparent to the user. She only has to specify the type of persistence desired (or leave the default settings) following the SoC principle.
- Language neutrality. The system offers adaptable persistence independently of the language used. We separate the interpreter from the language specification and the persistence framework has been designed without any dependency of a specific language.
- Parameter tuning. Finding the optimum performance level of a computer system with limited resources is a complex task. This is applicable to the field of databases, trying to manage higher data volume in the shortest time possible. For example, nested index has the best retrieval performance, however multiindex has the best update performance [Bertino 1989].
  Reflection makes simple for an application to programmatically modify these variables and generate statistics in order to make a decision. The system is a very suitable platform for the benchmarking of different persistence features (for instance selecting the best indexing mechanisms depending on objects' structure).

## Runtime Performance

The main disadvantage of dynamic application adaptation is runtime performance [Böllert 1999]. The process of adapting an application at runtime, as well as the use of reflection, induces a certain overhead at the execution of an application [Popovici 2001]. Adaptability and performance are usually two opposite concepts in computer science. In our first implementation we have tried to obtain maximum adaptability at runtime, following a complete SoC point of view.

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 [Gosling 1996], Python [Rossum 2001] or C# [Archer 2001]– due to optimization techniques such as just-in-time (JIT) compilation or adaptable native-code generation [Hölzle 1994]. 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 –such as the exploratory implementation of Python for .NET [Hammond 2001] that uses the .NET common-language-runtime (CLR) JIT compiler.

## 8 CONCLUSIONS

The Separation of Concerns principle aims at providing systematic means for effective modularization of crosscutting concerns, providing many benefits to software developers. The specific Aspect-Oriented Programming technique offers explicit language support for modularizing application concerns that crosscut the application functional code.

Although, persistence is often described as a classical candidate for aspectization, the existing aspect tools do not seem to be really suitable for developing persistence aspects. In addition, current aspect tools do not offer really dynamic weaving mechanisms, and all of them are dependent of a fixed programming language.

We have identified computational reflection as an appropriate technique to overcome the limitations mentioned. Our nitrO system is a reflective platform that offers non-restrictive computational reflection at runtime. Any application, written in any language, may use Python code to access its (or another one's) meta-level, achieving the modification of its structure or semantics.

A persistence framework has been codified in Python at the system's meta-level, offering dynamic persistence features transparently to the user. Taking Java as a sample language, programmers can develop transient applications without any reference to persistence, following the SoC principle. Then, employing any language, the user, the own application, or even another program, could set, unset, and modify, different persistence settings at runtime.

Our current implementation offers dynamic selection of storages, indexing mechanisms and update policies. This choice can be performed programmatically, depending on runtime emerging requirements. Therefore, the persistence system might be programmed to configure itself according to dynamic contexts. It can be also used as a parameter tuning or database research platform.

The Python platform, its persistence system, and the sample code presented in this paper can be downloaded from:
http://www.di.uniovi.es/reflection/lab/prototypes.html#persistence

## REFERENCES


[Archer01] T. Archer: *Inside C#*, Microsoft Press, 2001.

[Atkinson96] M. Atkinson, L. Daynès, M. Jordan, T. Printezis and S. Spence: "An Orthogonally Persistent Java", *SIGMOD Record*, vol 25 n. 4, December1996.

[Bergmans94] L. Bergmans: *Composing Concurrent Objects: Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs*, Ph. D. Dissertation. University of Twente, The Netherlands, June 1994.

[Bertino89] E. Bertino and W. Kim: "Indexing Techniques for Queries on Nested Objects", *IEEE Transactions on Knowledge and Data Engineering*, Vol.1 nº2, June 1989.

[Bertino95] E. Bertino and P. Foscoli: "Index Organizations for Object-Oriented Database Systems", *IEEE Transactions on Knowledge and Data Engineering*, Vol.7, April 1995.

[Böllert99] K. Böllert: "On Weaving Aspects". *European Conference on Object-Oriented Programming* (ECOOP), Workshop on Aspect Oriented Programming, June 1999.

[Gamma94] E. Gamma, R. Helm, R. Johnson and J. Vlisside: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Gosling96] J. Gosling, B. Joy, and G. Steele: *The Java Language Specification*, Addison-Wesley, 1996.

[Hammond01] M. Hammond: *Python for .NET: Lessons learned*, Active State Corporation, 2001.

[Hölzle] U. Hölzle and D. Ungar: "A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance", *Proceedings of the Object-Oriented Programming Languages, Systems and Applications* (OOPSLA), October 1994.

[Hürsch95] W.L. Hürsch and C.V. Lopes: *Separation of Concerns*, Technical Report UN-CCS-95-03, Northeastern University, 1995.

[Kiczales91] G. Kiczales, J. des Rivieres, D.G. Bobrow: *The Art of Meta-Object Protocol*, MIT Press 1991.

[Kiczales97] 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, June 1997.

[Kiczales01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold: "Getting Started with AspectJ", *Communications of the ACM*, October 2001

[Kielze02] 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, June 2002.

[Lam02] John Lam: "CLAW, Cross-Language Load-Time Aspect Weaving on Microsoft's Common Language Runtime", *AOSD 2002*, Enschede, The Netherlands, April 2002.

[Ley03] Ley, M: *DBLP: Digital Bibliography and Library Project*. http://dblp.uni-trier.de/.

[Maes87] P. Maes: *Computational Reflection*, Technical Report 87_2, Artificial Intelligence Laboratory, Vrieje Universiteit Brussel, 1987.

[Mens97] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales: "Aspect Oriented Programming Workshop Report", *ECOOP Workshop Reader*, Springer-Verlag LNCS 1357, June 1997.

[Ortin02] F. Ortin and J.M. Cueva: "Implementing a Real Computational-Environment Jump in order to Develop a Runtime-Adaptable Reflective Platform". *ACM SIGPLAN Notices*, Volume 37, Issue 8, August 2002.

[Ortin03] F. Ortin and J.M. Cueva: "Non-Restrictive Computational Reflection", *Elsevier Computer Standards and Interfaces*, Volume 25, Issue 3, June 2003.

[Parnas72] D. Parnas: "On the Criteria to be Used in Decomposing Systems into Modules", *Comunications of the ACM*, Vol. 15, No. 12. 1972.

[Pawlack01] R. Pawlack, L. Seinturier, L. Duchien and G. Florin: "Jac: A flexible and efficient framework for aop in java", *Reflection'01*, September 2001.

[Pinto01] M. Pinto, M. Amor, L. Fuentes and J.M. Troya: "Run-Time Coordination of Components: Design Patterns vs. Component & Aspect based Platforms", *European Conference on Object-Oriented Programming* (ECOOP), Workshop on Advanced Separation of Concerns, June 2001.

[Popovici01] A. Popovici, Th. Gross and G. Alonso: *Dynamic Homogenous AOP with PROSE*, Technical Report, Department of Computer Science, ETH Zürich, Switzerland, 2001.

[Rashid00] A. Rashid: "On to Aspect Persistence", *GCSE Symposium*, Springer-Verlag LNCS 2177, October 2000.

[Rashid03] A. Rashid, R. Chitchyan: "Persistence as an Aspect", *International Conference on Aspect-Oriented Software Development*, March 2003.

[Rossum01] G. Rossum: *Python Reference Manual*. Fred L. Drake Jr. Editor, Relesase 2.1, 2001.

[Schult02] W. Schult and A. Polze: "Aspect-Oriented Programming with C# and .NET", IEEE *International Symposium on Object-oriented Real-time distributed Computing*, May 2002.

[Smith82] B.C. Smith: *Reflection and Semantics in a Procedural Language*, MIT-LCS-TR-272, MIT, Cambridge, 1982.

[Suzuki99] J. Suzuki and Y. Yamamoto: "Extending UML for Modelling Reflective Software Components", *The Unified Modeling Language - Beyond the Standard, Second International Conference*, Springer-Verlag LNCS 1723, October 1999.

[Tarr99] P. Tarr, H. Ossher, W. Harrison and S. Sutton: "N Degrees of separation: Multi-Dimensional Separation of Concerns", *International Conference on Software Engineering*, May 1999.

## About the authors

**Benjamin Lopez** is an Associate Professor of the Computer Science Department at the University of Oviedo (Spain). He is a Computer Scientist Engineer from the Malaga Computer Science Faculty. His research interests include Computer Graphics, Object-Oriented Persistence Systems, Computational Reflection, Multimedia, Human Computer Interaction and Object-Oriented Technologies. He can be reached at benja@lsi.uniovi.es.

**Francisco Ortin** is a Temporary Full University Lecturer at the Computer Science Department of the University of Oviedo, Spain. He received in 2002 his PhD in Computer Science with the Thesis entitled "A Flexible Programming Computational System developed over a Non-Restrictive Reflective Abstract Machine". His main research interests are Computational Reflection, Object-Oriented Virtual Machines, and Dynamic-Weaving Aspect Oriented Software Development. His website is http://www.di.uniovi.es/~ortin.

**Javier Noval** received in 2000 a Computer Science degree from the Technical School of Computer Science and, in 2003, a MS in Computer Engineering; both at Oviedo University. Currently, he is a PhD Student of the Computer Science Department. His research interests are the Design and Implementation of Programming Languages and Processors, Computational Reflection and Open Source Software Communities. He can be reached at javinoval@terra.es.