Reflective Implementation of an Object Recovery Design Pattern

Acauan Pereira Fernandes
Universidade da Região da Campanha
Centro de Ciências da Economia e Informática
Av. General Osório 2289 – Bagé – RS – Brasil – 96400-101
+55 (53) 242-8244
acauan@urcamp.tche.br

Maria Lúcia Blanck Lisbôa
Universidade Federal do Rio Grande do Sul - Instituto de Informática
Av. Bento Gonçalves, 9500 - Campus do Vale - Bloco IV
Porto Alegre - RS - Brasil
CEP 91501-970 Caixa Postal: 15064
+55 (51) 3316-6159 Fax: +55 (51) 3316-7308
llisboa@inf.ufrgs.br

Abstract

Patterns are powerful tools to document software problems and their solutions, as well as when and how to use them. They can help improve software reuse. The implementation of non functional requirements, such as atomicity, can benefit from this approach. This paper discusses and shows how computational reflection features can be employed within such context, increasing reuse of the software produced this way. It also shows how a reflective implementation of a software pattern created to introduce customizable recovery to objects can use all these concepts in a way to get the best from each one of them. Benefits from such reflective implementation are discussed, also considering other aspects such as flexibility, simplicity, dependability and development speed. It gathers concepts from different paradigms as software patterns, computational reflection and the object oriented model in order to achieve such characteristics.

Key words: software patterns, fault-tolerance, atomicity, computational reflection, data recovery.

1 Introduction

The capacity of defining reusable solutions for recurrent problems makes software development faster and more dependable. Within such context, software patterns have been used in a way to supply answers to that search [RIE 96].

Patterns help to reduce software complexity by clearly describing a software solution in terms of structure, dynamic behavior and context. Similar to what happens for any intrinsically complex systems, the definition of clean interfaces among components contributes for their independent development and reuse. In the case of software fault-tolerance domain, patterns can be used to present a collection of relatively independent solutions to common non-functional properties problems such as atomic actions, recovery points, replication policies and exceptions [LIS 98].

Since both computational reflection and patterns have important features to offer when it comes to software reuse, they might be used together. Patterns show a reusable solution, whereas reflection can turn it into a reusable implementation. This separation of aspects means to discharge the developer from having to deal with non functional requirements.

This paper shows how the implementation of a recovery pattern suggested by Silva [SIL 96] can be done using computational reflection concepts in order to make it easier and more adaptable.

1.1 The meta-level approach for fault-tolerance

The key concept in the design of software using meta-level architecture style is the separation of the system in two tiered layers: a meta-level and a base level. Those layers have different but related responsibilities thus representing separate aspects of the same software system. The base level encompasses all the components that implement the functionalities of an application as defined on its functional requirements. The meta-level provides a self-representation of the software to give it knowledge of its own structure and behavior [BUS 96]. Computational reflection [MAE 87] provides this basis for meta-level architectures. By using computational reflection one can easily prospect information about base-level classes during execution, such as their fields and current values, as well as alter the latter. All these features can be widely used to implement very adaptable and reusable applications.

Software fault-tolerance encompasses all techniques and programming languages mechanisms intended to support the development of high reliability software. We can consider the fault-tolerance area a specific domain of knowledge composed by well-defined techniques used to guarantee the reliability of applications built over other domains. This view of fault-tolerance makes it ideal to promote separation of aspects [KIC 97] by means of reflection-based implementation techniques.

The relationships among the base-level and the meta-level components are established by a MOP - Metaobject Protocol. The MOP provides a high level interface to the programming language implementation in order to reveal to the program information normally hidden by the compiler and/or run-time environment [LIS 98]. As a consequence, a programmer can develop language extensions, adapt component behavior and even make non-permanent changes into the system. A metaobject holds static and dynamic information about the base-level objects. Figure 1 depicts some static (e.g. class and type) and dynamic information (e.g. values) reified as data within the metaobject.

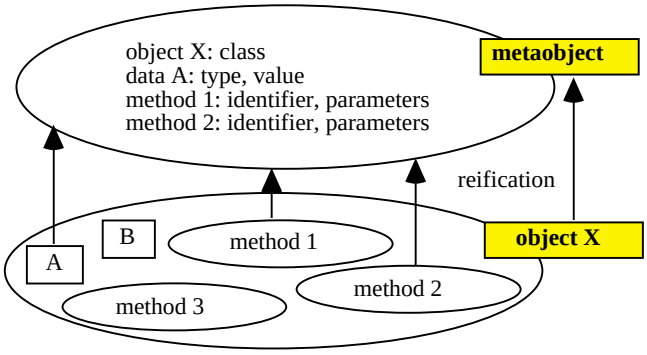

Figure 1- Meta-information

The implementation shown in this paper uses a meta-object protocol, which allows Java programming language to support computational reflection using message interception. This protocol was developed by [OLI 98] for Java and is called Guaraná. It has been used along with Java native reflection API because it enhances the latter features.

1.2 Object Recovery and Atomicity

Fault tolerance is an important example of non-functional requirement that can benefit from this approach. Among the concepts related to fault tolerance, there is one we have particular interest, which is atomicity. This property must guarantee data consistency even in the presence of failures. Two basic factors must be a concern of the sofware system: concurrency accesses control and state recovery [JAL 95]. The possibility of an object being accessed concurrently by any number of external calls brings the need of managing them to avoid that such situation leads that object to inconsistent states. The state recovery deals with policies to keep data consistency even after any kind of failure. Any data which was incompletely changed by an application at the moment of a failure ocurrence should be returned to its previous state, or else the operations that led it to such state should be finished until their end [BER 87].

Let's consider an **S** object with a **t1** interface. Messages sent to one of its methods may trigger the execution of a number of operations that lead the object from its initial state (**is**) to a final state (**fs**). The object state is the current set of its attributes' values. If during the execution of such operations there is a failure, the message sender (the object's client) can be notified by an exception generated by **S**, resulting in the end of the method execution in **S**. However, there is no guarantee that there have been no changes in the object's state during the method execution. Therefore, granting an object atomicity consists of assuring that its previous state (**is**) is preserved as the current state if the transition **is** -> **fs** is not successfully completed, or else the unsuccessful operations are redone in order to reach the object's final state (**fs**). Recovery deals with policies to achieve such goals.

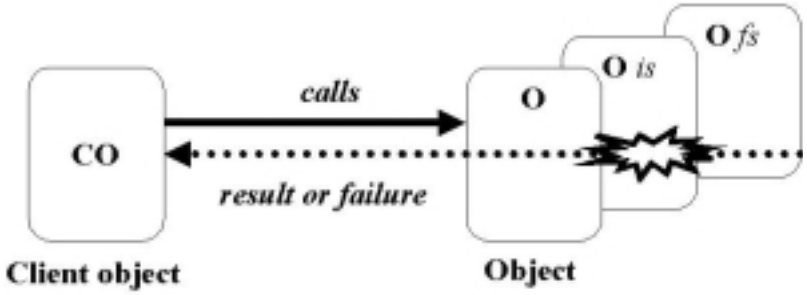

Figure 2 – Object state transition

Dealing with state restoration can be as simple as making a copy of the original object or as complex as recovery cache memory implemented in hardware [RUB 94]. Thus, the problem of state restoration needs special solutions and this concern suggests the existence of another opportunity to create design patterns for different contexts of state restoration [LIS 98].

1.3 Software Patterns and Computational Reflection

A software pattern is a three-part rule which expresses a relationship between a context, a certain system of forces that occur recurrently in such context, and a software configuration that allows these forces to resolve their conflicts. Patterns can be classified according to their level of abstraction [GAB 96]. There may be conceptual patterns, design patterns and implementation patterns. They help create

applications which face problems that have already been solved somewhere else. We can reuse the concept, the abstract solution or the implementation.

Reflection can be seen as a pattern [BED 98]. This pattern splits the application in two parts (meta-level, base level) linked in a transparent way by using interception mechanisms. Messages sent to the base level objects linked to meta-objects are intercepted. According to this pattern, it should be possible to design a system focusing only in its functional requirements, integrating the non functional requirements later on, without needing to alter its original structure.

Anyway, the point is reuse; more specifically, we claim to reuse the implemenation in a transparent manner.

2 Abstract and Meta-class Implementation

2.1 The base pattern

The Customizable Object Recovery Pattern [SIL 96] is a design pattern that allows an application object to supply its own recovery. It also allows the implementer to chose among different recovery policies, according to the current context of the application. The authors suggest an implementation based on abstract classes structured as depicted on Figure 3.

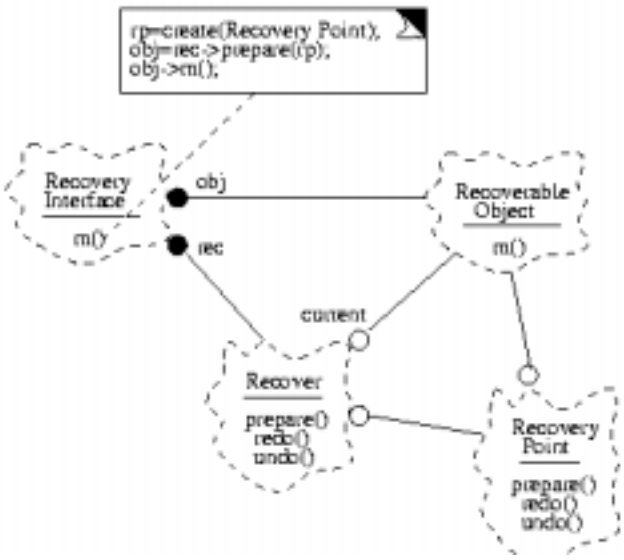

Figure 3 - The Customizable Object Recovery Pattern [SIL 96]

If we chose abstract classes to implement the pattern we would leave the developer the task of specializing those abstract classes in order to provide a concrete implementattion by means of concrete classes tailored to the context of the application. We extend this pattern to deal with interception mechanisms in order to achieve a more transparent link between the application and the state recovery library. Reflective implementations of these policies are much more flexible since they demand no previous structural knowledge.

2.2 A Reflective Implementation

In order to demonstrate how objects can benefit from atomicity addition in a simple and quick way, just by passing few parameters, a meta-class - called MC class - was developed. Such meta-class implements generic services to provide state recovery to any application object and it has a few methods which serve as interface between them and the base level objects. It is an example of a generic meta-class that can be used by a developer in many different ways and contexts. Its goal is to show the flexibility provided by the use of computational reflection in patterns implementations.

The meta-object MC intercepts calls from clients to the Recoverable Object (its base object) as shown at Figure 4. Then MC object makes introspection into the object and save the object current state for an eventual posterior recovery. To increase flexibility and reduce the overhead caused by reflection, just method invocations are intercepted. This means that all of the operations called from inside method are not intercepted.

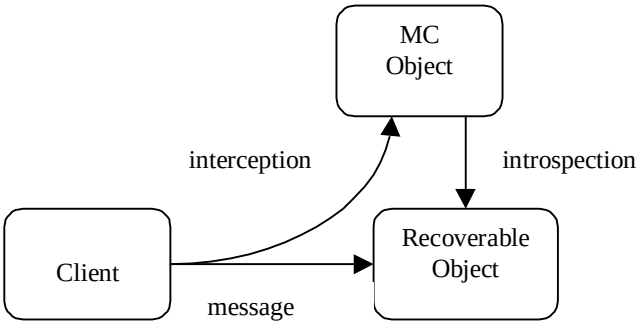

Figure 4 - Meta-class implementation

The reflective implementation corresponding to Customizable Object Recovery Pattern (see Figure 3) uses the same suggested class but focus on transparency, as follows:

- *Recoverable Interface*: Intercepts messages to the Recoverable Object, encapsulating it. Its effects are recoverable. This job is done by the MC meta-class. This meta-class may also be a link between the base level object (Recoverable Object) and the different recovery policies, implemented in the pattern as Recover subclasses, in multiple meta-configurations.
- *Recoverable Object*: the object that contains the data which will be dealt with and may need to be restored. It corresponds to the base level object which implements recovery.
- *Recover*: object independent part of the recovery policy. It holds the Prepare, Redo and Undo operations. Particular policies may be implementent by the use of subclasses. Corresponding methods are implemented in the MC meta-class and are used in its Handle method. Particular policies are easily implemented just by adding or excluding new meta-classes on multiple meta-configurations. Reflective implementations substitute inheritance mechanisms by dynamically adding and excluding new meta-classes. It turns the reflective implementation by far more flexible and reusable, since new meta-classes are added to or excluded from the meta-configuration without requiring it to be previously aware of their structure or even their existence. The new classes do not need to be declared as Recover subclasses.

- *Recovery Point:* object-specific recovery policy. It may need to know the Recovery Object's internal data. The use of computational reflection does not require any knowledge of any data structure, because they can all be gotten during run time. Thus, any class can be used as a Recoverable Object, since there is no specific code attached to any specific class. It makes this approach much more reusable.

The Redo and Undo operations implemented by the Recovery Point object may use different policies. Such policies depend on the operations' targets. If operations are done on copies of the objects, then the Redo is accomplished by updating them, otherwise they are already updated. Undo operations follow the same pattern: in the first case, the object did not suffer their effects and nothing is necessary to be done. However, if operations were done directly on the objects, then they must be restored.

Some of the MC meta-class attributes are used to help implement the different recovery policies mentioned above. It supports all of them without requiring any change. For instance, if operations are executed directly on the objects, then the Object "copy" attribute can be used. If the adopted policy changes object copies instead of the objects themselves, then the vector "values" is used jointly with the Field array "fields" to contain copies of the object current or previous values. New meta-classes with different policies are added to the meta-configuration and use these attributes accordingly to their needs. Yet, reuse is increased by the total absence of previous knowledge request.

2.2.1 The MC meta-class architecture

The MC meta-class uses many reflective features to get important information about its base level object and to decide about data reification. Among its atributes there are a reference to the base level object, an array containing this object fields, a vector with their values and a variable that is used as a flag to activate reification. It also has a hashtable and an *OperationFactory* instance. This class is responsable for generating meta-level operations in the Guaraná meta-object protocol. :. A MC object shall be instantiated using a parameterized constructor as follows:

```
public MC (Object ob_) {
copy = ob_;
Class c = copy.getClass( );
fields = c.getDeclaredFields( );
for (int i = 0; i<fields.length; i++)
                try{
                        values.addElement(fields[i].get(copy));
                }catch (IllegalAccessException e) {}
Guarana.reconfigure(copy,null,this);
}
```

The base level object should be sent as a parameter to the meta-class constructor:

```
Class c = new Class( );
MC mc = new MC(c);
```

The MC constructor saves a reference to the base level object, gets information about its class, amount and types of fields and saves its initial state in the meta-level. Then, it associates this object to itself, starting message interception. A first advantage of this mechanism is that it is totally transparent to the base level object, and it can be reused with instances of any other class. It neither depends on the class structure nor needs to know it in advance.

2.2.2 The MC interface

The definition of the meta-class in charge of intercepting messages to a base level object in the application, saving and restoring its state has some atributes needed to achieve such goal.

```
public class MC extends MetaObject {
private Object copy;                                  // a reference to the base level object
private Vector values = new Vector( );                // stores reified values of the base level object
private Field[ ] fields;                              // stores the base level object fields
private Hashtable pending = new Hashtable( );         // used to avoid infinite recursion
private OperationFactory opf;                         // creates operations in the meta-level
private int on_off = 0;                               // flag          .
}
```

2.2.3 State saving

Reification is the key to object state inspection and saving. The methods developed for this purpose are:

1. *Reify:* it reifies the base level object, creating read operations in the meta-level. Such operations are created by an instance of an *OperationFactory* class.

```
Operation op = opf.read(fields[i]);
pending.put(op,op);
Object value = op.perform( ).getObjectValue( );
pending.remove(op);
```

These operations created in the meta-level are stored in a hashtable to avoid infinite recursion, one of the problems that are faced when dealing with message interception.

2. *Save:* it restores the values stored in the meta-level back into the base level object. It also uses an instance of *OperationFactory* to create writing operations.

```
Operation op =opf.write(fields[i],values.elementAt(i));
pending.put(op,op);
Object value = op.perform( );
pending.remove(op);
```

Just like in the previous method, operations created in the meta-level are stored in a hashtable to avoid infinite recursion. This is achieved by including the operation in the hashtable immediately

before executing it, and removing it right after it is done. The test shown in the Handle method below does the rest.

The object state is saved just once before the method execution. The *isMethodInvocation* function from the Guaraná MOP used in the *handle* method in the meta-level alllows one to realize such situation. The meta-class also supplies an interface to its base level object which is formed by four methods

1. *Permanent:* unables interceptions for recording, keeping the last saved version of the base level object in the meta level.
2. *Temporary:* enables interceptions for recording, so that the object current state can be saved in the meta-object at every call to any of its methods.
3. *Restore:* restores the latest object state saved in the meta-level back into the base level object.
4. *Checkpoint*: reifies the base level object atributes and then unables posterior interceptions for recording.

The meta-object initially explores the base level object in order to discover all of its field names and types, as well as their current values, and records all this information in the meta-level. It also has a reference to the base level object itself, for posterior accesses. From this moment on, its behaviour can be controlled accordingly to the developer's needs, employing the methods supplied by the meta-object interface.

The following code shows how to use this interface. The "Permanent" method unables interceptions for recordings, making all changes in the object permanent. When the "Checkpoint" method is called, the current state of the object is saved in the meta-level and posterior recordings are unabled. Operations executed on the base level object from this point on can be undone by the "Restore" method ou committed by the "Checkpoint" method. The "Temporary" method enables interception again e restarts saving the base level object current states.

```
mc.permanent( );
... calls for base level object methods...
mc.checkpoint( );
... calls for base level object methods...
mc.restore( ); or mc.checkpoint( );
```

3. *Permanent, temporary, restore e checkpoint:* use the previous methods and the flag variable.

The capacity of creating operations in the meta-level gives computational reflection a brand new range of possibilities. Not only can operations be intercepted, but they can also be created according to the context. The MC meta-class creates reading and writing operations and execute them in the meta-level, without the base level object's knowledge.

4. *Handle:* this method is called by the meta-object protocol kernel whenever a call is sent to the base level object. Infinite recursion between both levels is caused by an operation in the meta-level that accesses a base level object associated to a meta-object. This causes a new interception and so forth. In order to avoid such problem, this meta-class employes a inclusion/exclusion mechanism and a hashtable, which is used to test whether the operation is a new one or just an old operation generating recursion, as shown in the following test sample code.

```
if (pending.containsKey(op))
        return null;
```

If the operation is already stored in the hashtable, recursion is avoided by the “return null” instruction, which sends the execution flow back to the base level.

As mentioned before, in order to decrease overhead, just invocations to methods are treated. That avoids that each instruction in the method generates a new data recording in the meta-level. This recording is done just once, in the beginning of this process.

```
if ( op.isMethodInvocation( ) && on_off == 0 )
        return null; //dos not reify

if ( op.isMethodInvocation( ) && on_ff == 1 )
        reify( );   //reifies

if ( op.isMethodInvocation( ) && on_off == 2 )
        save( );     //saves

if ( op.isMethodInvocation( ) && on_off == 3 )
 {
        reify( );      //reifies once and turns it off
        on_off = 0;
}
```

2.3 Meta-classes composition

The meta-class MC shows how to implement atomicity in a low level of abstraction using computational reflection. It achieves reuse and transparency decreasing the overhead that may be introduced by reflection. This abstraction level also brings more flexibility, because it can be employed in many different situations. From this kind of implementation, one can get to building frameworks with decision mechanisms to cover a wider range of sceneries to support atomicity.

We can get meta-classes together in a single configuration by using *Composer* and *SequentialComposer* meta-classes provided by Guaraná meta-object protocol (figure 5). They both allow method delegation and can connect a single base level object to more than one meta-object or vice-versa, providing the former with features from the latter ones. Such delegation allows dynamic reconfiguration, which is not achieved only by the use of inheritance mechanisms[SIL 96].

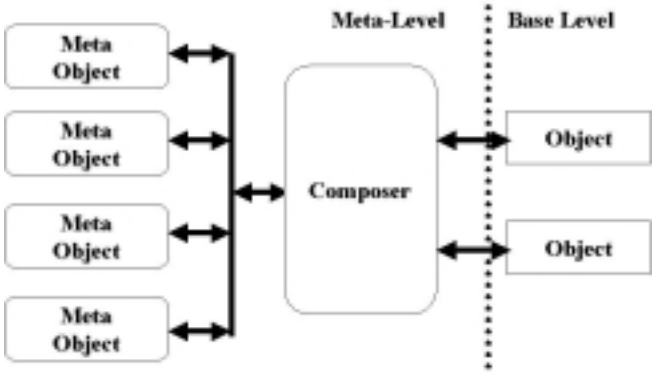

Figure 5 - Multiple meta-configurations

Each one of these meta-objects may be responsable for implementing a non functional requirement or an alternative policy, so that all a developer needs to do is connect his application base level objects to the meta-objects with the features he wants them to have. Some meta-object may be responsable for selecting the most appropriate policy for recovering or concurrency control, for instance, making the application more flexible and able to customize itself dynamically according to the context during its execution. All this can be done without the need of any extra coding and can be used with any class, even if it is not previously known, because the meta-objects use reflective features to determine each new class structure..

Non functional requirements, such as fault tolerance, distributed processing, data persistence and concurrency control, can be added to the application in a transparent and easy way, using the techniques listed in this paper, achieving a high degree of not only transparency, but also reuse and productivity, in a safer and simpler way.

3 Conclusions

Software patterns have been used as powerful tools to document experiences obtained when dealing with software problems. They not only spread a solution already tested and approved in a certain context, but also show when and how to use it.

Implementations based on established patterns and computational reflection allow not only faster application development without risking their dependability, but also increase the reuse of these solutions. Non functional requirements, such as atomicity, are present in practically any kind of application, so they are natural candidates to be implemented in a reusable way. Computational reflection has been used in the development of this kind of application requirements for its great number of features that work with this purpose. Implementing software patterns using reflection is, therefore, an easy noticing association when it comes to reusing software.

Reflective techniques can increase the usefulness of the solutions proposed by patterns, especially those which deal with non functional requirements or other areas where reflection has achieved success. The reflective implementation of this recovery pattern shows how, also making use of characteristics intrinsic to the object oriented model, computational reflection and software patterns can work together in the definition and solution of these kinds of problems.

The MC meta-class here described allows an object to manage its own state recovery, releasing the developer from worrying about this requirement in his application. Objects of any class can benefit from such feature, because computational reflectional mechanisms are used to implement a generic meta-class. Recovery can be activated an deactivated at any moment, according to the application demands. Besides that, functional and non functional requirements are set apart, which contributes to increase reuse and development speed. The developer's task is simplified and he can spend his time dealing only with problems concerning the application targets themselves.

The MC meta-class is to be included in a framework to help develop fault-tolerant applications trying to get the best features from many different paradigms, in order to provide reuse, transparency, dependability and flexibility to fault-tolerant software development, as well as speeding this processs up, without increasing its complexity or adaptability. Composer meta-classes with decision mechanisms can be responsable for connecting different aspects of fault-tolerance, such as concurrency control, data recovery, data persistence, etc, encapsulated in meta-classes, to any objects created by the application developer, providing them all these characteristics as simply as possible, delving into the use of metaobjects to supply fault-tolerance [LIS 95].

References

[APP 00] APPLETON, B. **– Patterns and Software: Essential Concepts and Terminology.** Available at http://www.enteract.com/ ~bradapp/.

[BED 98] BEDER, D.M. ; RUBIRA, C.F. – **Uma Abordagem Reflexiva baseada em Padrões de Projeto para o desenvolvimento de Aplicações Disribuídas Confiáveis.** Instituto de Computação, Unicamp, SP, 1998

[BER 87] BERNSTEIN, P.A.; HADZILACOS, V.; GOODMAN, N**. Concurrency control and recovery in database systems**. Addison Wesley, 1987.

[BUS 96] BUSCHMANN, F. et al**. A system of patterns: pattern-oriented software architecture**. John Wiley & Sons, England, 1996.

[GAB 96] GABRIEL, R.P. – **Patterns of Software: Tales from the Software Community**. Oxford, 1996.

[GAM 95] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES. J. **– Design Patterns: Elements of Reusable Design**. Reading, Massachssets: Addison-Wesley.

[JAL 95] JALOTE, P. – **Fault tolerance in distributed systems**. PTR Prentice Hall. New Jersey, 1995. Cap. 6. P. 217-253.

[KIC 97] KICZALES, G. et al. **Aspect-Oriented Programming**. Em Proceedings of ECOOP'97. LNCS 1241. Springer-Verlag, pp. 220-242, 1997.

[LIS 95] LISBOA, Maria Lúcia Blanck. **MOTF Meta-objetos para tolerância a falhas**. Porto Alegre: CPGCC UFRGS, 1995. 171p. PhD Thesis.

[LIS 97] LISBOA, Maria Lúcia Blanck. **A New Trend on the Development of Fault-Tolerant Application: Software Meta-Level Architectures.** Journal of the Brazilian Computer Society, 4(2): 31-38, Nov. 1997.

[MAE 87] MAES, P. **Concepts and experiments in computational reflection**. SIGPLAN notices, NY. OOPSLA 1987

[OLI 98] OLIVA, A. – **Guaraná – uma arquitetura reflexiva**. Available at http://www.sunsite.unicamp.br/~oliva/guarana/index.html.

[PAP 86] PAPADIMITRIU, C.H., **The theory of database concurrency control**, Computer Science Press, 1986.

[RIE 96] RIEHLE, D.; ZÜLLOGHOVEN, H. – **Understanding and Using Patterns in Software Development.** Available at http://www. citeseer.nj.nec.com/ riehle96undertanding.html

[RUB 94] RUBIRA, C. M. F, STROUD, R. **Forward and backward error recovery in C++**. Object Oriented Systems, 1(1): 1-85, 1994

[SIL 96] SILVA, A.R.; PEREIRA, J.; MARQUES, J.A. **- Customizable Object Recovery Pattern.** Available at http://www-rodin.inria.fr/~pereira/pub.html.