A New Trend on the Development of Fault-Tolerant Applications:

Software Meta-Level Architectures

Maria Lúcia Blanck Lisbôa

Universidade Federal do Rio Grande do Sul
Instituto de Informática, Caixa Postal 15064
91501-970 Porto Alegre, RS, Brasil
llisboa@inf.ufgrs.br

Abstract

The purpose of this paper is to investigate a clearly defined way of developing fault-tolerant applications using software meta-level architectures. Meta-level architectures are software architectures based on computational reflection. It addresses complex pieces of software: fault-tolerant software. Fault-tolerant applications must cope with several non-functional requirements to maintain its functionality. So, it is particularly relevant to investigate how to alleviate developers from repeatedly deal with this complexity. Some solutions are presented such as software patterns and basic guidelines to help the development of such applications.

Keywords: *meta-level architecture, computational reflection, software patterns, and fault-tolerance*

Introduction

Meta-level architectures have recently acquired a great deal of researcher's attention as it offers new solutions to well known problems. Particularly among those researchers interested in fault-tolerance as can be seen on some recent works [1], [2], [3], [4], and [5].

As happens with any emerging methodology, the concepts related to meta-level architectures are not well understood. These concepts range from computational reflection to software architecture styles. Both are becoming explicit fields of study and there is currently no well-established methodology combining them into meta-level architectures.

Meta-level architectures are software architectures based on computational reflection. Computational reflection is the ability of observing and manipulating the computational behavior of a system. While software architecture addresses the structure and interaction among software components, computational reflection concerns are about introspection into software components. That is, components can get information about the internal properties of other components. In the object model, this means to use (a) metaclasses describing classes, methods, attributes, and (b) metaobjects representing objects. As a consequence, computational reflection defines an architecture composed by meta-levels, in which the meta-level components represent and retrieve information from application components.

Some reflection-based solutions offer obvious benefits. However, they can introduce additional complexity and liability on the fault-tolerant software development. The challenge of this paper is to investigate a clearly defined way of developing fault-tolerant applications using software meta-level architectures. Some ideas were inspired by the work of Buschmann [6] who documented a collection of eight architectural patterns including a general reflection pattern. Also recently, Islam [7] documented a pattern for distributed systems, which emphasizes the two fundamental forces: fault-tolerance and performance.

Fault-tolerant software can be very complex rendering its development subjects to human fallibility. Based on intentional inclusion of redundancy supported by special mechanisms as exception handling, checkpointing and persistence, fault tolerant applications must cope with several non-functional requirements to maintain its functionality. Also, as stated by [6], to specify non-functional requirements for software architecture, it is necessary to consider the interdependencies and trade-offs that exist between them. So, it is particularly relevant to investigate how to help developers to deal with this complexity. To achieve this objective, this paper addresses the structure of fault-tolerant software. The approaches are examined:

- how to structure a fault-tolerant software using meta-level architecture;
- how software patterns can contribute to make the software development process more effective;
- how to integrate successful solutions into a given application domain.

2 Software architecture and design patterns

As observed by [8], designers have begun to develop a shared repertoire of methods, techniques, patterns and idioms for structure complex software systems. Informally speaking, in a bottom-up point of view, programs are composed by interacting components, which are styled and organized according to a structure defined by software architecture. Some of these components might be implemented following a design pattern while others could be specializations of classes from a framework. Hence, software architectures molds some relationships among frameworks, design patterns and components used to build an application.

Architectural styles define the fundamental structure and interactions among the components of an application. Components are atomic units and their interaction provides the functional and non-functional requirements. One or a group of components fulfills each requirement. Conversely, one component can fulfill a number of requirements. Moreover, an application may include several components not directly related to any requirement (figure 1). The structural and logic relationships among these components constitute the major task in system design.

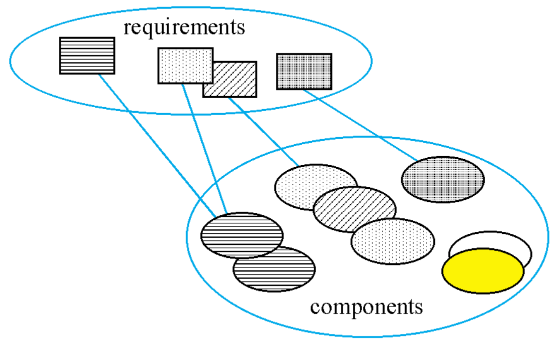

Figure 1: Relationships among components and requirements

However, it is possible to look at some of these relationships and identify them as recurrent on different systems. Hence, such problems can have an application independent solution and thus described using abstractions of structure and interactions. The description of a solution using components, collaborations among components and their design context is a design pattern.

In an attempt to provide help to a number of software designers, experts captures the essentials of a design problem that arises in many systems and then they describe the basic structure of well proven solution to it. As stated by [7], early experiences indicates that pattern-based solutions design reuse reduces the time needed to solve a problem and makes it easy to customize the core solution for a specific implementation.

Both frameworks and design patterns describe solutions to software problems. Frameworks intend to solve several problems related to a specific domain presenting a collection of classes. A framework class is an incomplete or complete implementation of an abstraction. As a consequence, a class is programming language dependent. A design pattern describes a solution using a language independent *idiom* [6], but it is not supposed to present a specific solution to problems found in an application. It focuses on a particular, well-known and widely applicable software design problem.

3 Meta-level architectures

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. Additionally, those layers have different but related responsibilities.

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 [6]. Computational reflection provides this basis for meta-level architectures.

Thus, a reflective architecture defines a system that incorporates data structures representing itself. The self-representation makes it possible for the system to get information concerning to the state, structure and behavior of some systems components. Moreover, based on that information it can dynamically interfere on its current computations. More precisely, the meta-level components can change and adapt some structural and behavioral aspects of the base-level components.

A MOP - Metaobject Protocol, establishes the relationships among the base level and the meta-level components. 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.

When a compiler hides internal information, it follows the principle that an abstraction should expose its functionality but hide its implementation. Since the MOP provides an interface to the object representation, it relaxes the information hiding principle. Thus, these components have open implementation [16]. As a consequence, a programmer can develop language extensions, adapt component behavior and even make non-permanent changes into the system.

During the compilation process, the compiler has information about the classes, classes hierarchies, identifiers, types and other attributes of classes, data structures and methods regarding to the program. Such information, that is not likely to change during the execution, constitutes the structural meta-information part of a metaobject protocol.

Conversely, the non-permanent information, e.g. the values of arguments passed to a method, depends on the execution. This kind of data deals with the behavioral aspects of the components and it is part of the dynamic information that metaobjects hold. Depending on the metaobject protocol, metaobjects provide different meta-information. In summary, metaobject hold static and dynamic information about the base-level objects. Figure 2 depicts some static (e.g. class and type) and dynamic information (e.g. values) reified as data within the metaobject.

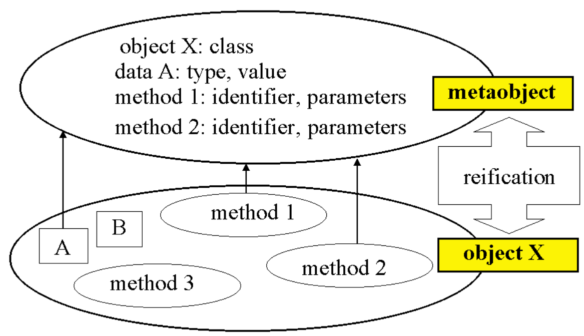

Figure 2: Meta-information

Before we begin to discuss how to use meta-level information, we examine how to get it through a metaobject protocol. In the object model, the meta-information interface is usually implemented at one single class, which by further specialization or direct instantiation offers to the program the meta-level interface.

The meta-level interface defines the methods that implement the reflective services. Generalizing from different MOPs, it is possible to identify four groups representing the basic aspects of metaobjects protocols.

• *Attachment of base and meta-level*: The purpose of this group is to bind base-level components to meta-level entities. The binding can be static or dynamic. An entity is one or a group of objects or classes. It is possible to bind them in a one-to-one basis and also multiple metaobjects can be bound to one base-level entity. Thus, one can devise many semantics: one metaobject can control just one base object or a number of base objects of the same class, or multiple metaobjects can be used to control one or a number of base objects.

• *Reification*: Reflection means materialization of information otherwise hidden to the program. This group deals with reification of incoming and outgoing messages, reification of arguments, data and other structural information.

• *Execution*: This group addresses the meta-level computation and the ability to invoke base-level methods. Metaobjects are ordinary objects and thus they provide services through public interfaces and invoke others objects in a normal way. But, to perform base-level computation transparently to the server they depend on reification. When a base object receives a message, the corresponding metaobject intercepts and materializes the method identifier and the actual parameters. Re-sending the message to the original destiny, before or after meta-level computations, the metaobject acts as an indirect client of the base object.

• *Modification*: Once the meta-level holds information about the behavior and structure of base objects, computations can change some aspects of the base components. The ability to handle structural information is critical since it can affect multiple objects. One alternative is to make changes on shadow classes, as proposed by [9] without affecting semantics of other objects of the same class.

There is no universally accepted MOP that serves as a recipe to implement reflective support in programming languages. First, it is necessary to establish what kind of meta-information is useful to reveal to the program. Second, there are several implementations of programming languages intended to support the object model. Some of them use a common root class to derive the classes used to instantiate application objects while others use objects to describe the application objects. Some programming languages have native reflective facilities while in others the support for reflection is included by language extensions.

Some examples are: Smalltalk meta-classes define a native interface for structural reflection; the original design of the Java Virtual Machine supports some introspection about classes and objects, but recently it has been extended to offer more reflective mechanisms [10], [9]; C^{++} does not support reflection - only little information about types provided by library functions. However, different MOPs [11], [12], [13] extended the C++ language.

4 Meta-level approach for fault-tolerance

Software fault-tolerance encompasses all techniques and programming language 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. Thus, we can figure a domain-specific architecture for software fault-tolerance.

4.1 Domain-specific meta-level architecture

A domain-specific architecture, as stated by [14], comprises: a) a reference architecture, which describes a general computational framework for a significant domain of applications, b) a component library, which contains reusable *chunks* of domain expertise, and, c) an application configuration method for selecting and configuring components within the architecture to meet particular application requirements.

Meta-level architectures have been achieving considerable attention as reference architecture for fault-tolerant applications. Evidence of this trend can be seen on some recent works.

• Fabre [2]: "... our experiments in implementing fault-tolerance and secure communication metaobjects were really successful and confirmed that a really good level of transparency and abstraction could be achieved within application objects, fault-tolerance and secure communication metaobjects."

• Stroud [3]: "With a metaobject protocol approach, non-functional aspects of atomic data types such as synchronization and recovery are implemented in metaobject classes at the meta-level, whilst functional aspects such as object operations are implemented at the base-level."

• Fraga [5]: "To improve the flexibility of the implementation, we use a reflexive approach which allows to separate aspects related to the replication model from those related uniquely to the service being replicated."

Those authors stressed the major issue related to meta-level architecture: the separations of concerns within the fault-tolerant application. It is advisable to separate out the fault-tolerant components from the rest of the system to reduce problems associated with the interaction between redundant components [16]. It follows that this separation, although focusing on independent responsibilities, rely on a tightly interaction among components.

In addition, computational reflection helps to organize fault-tolerant activities at meta-level and it provides transparent interfaces among components dealing with fault-tolerance and those implementing the functional requirements.

Specializing the definition of a domain-specific architecture given above, the fault-tolerance meta-level architecture comprises:

• A meta-level architecture as reference architecture for structuring fault-tolerant applications split on two layers: the fault-tolerance layer and the application logic layer.

• A collection of design patterns describing successful solution of fault-tolerance problems, and an open component library, which contains reusable meta-classes implementing fault-tolerant mechanisms to support the development of fault-tolerant applications.

• A metaobject protocol to provide a nice interface among the application components and the fault-tolerant level.

There are tangible benefits from this separation but there are also some liabilities. First, computational reflection adds cost to the application due to extra-level computation and indirection mechanisms. Second, it depends on reflective facilities to be supported by the target implementation language. Third, reflection is not a universal solution to programming problems. Instead, it is primarily concerned with implementation of open library components.

4.2 Drawing design patterns

The next step toward a reference meta-level architecture is to devise design patterns capturing the essentials of fault-tolerance schemes. Considering the fault-tolerance domain, the body of knowledge and the solutions given to many non-functional requirements are well established and thus it is possible to use design patterns to describe successful solutions.

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. When applied to 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.

Suppose a fault-tolerant application built from several components. Some of these components should provide critical services while others are not so critical but there are many interactions occurring among them. In our architecture, components are implemented according to the object model. For several reasons, the object model offers good support for the development of fault-tolerant applications:

- one can easily replicate and distribute objects,
- object diversity can be achieved by changing the implementation of objects without changing its interface or external behavior,
- objects can be considered the smallest grain of failure and distribution, and,
- the message passing mechanism facilitates object monitoring.

Since the objects represent all the application activity, both critical and non-critical objects must provide reliable services but only the critical ones are subject to fault-tolerance techniques. The interaction among a fault-tolerant object and its client shall be the same as the interaction among any other objects. That is, the client should not be aware of the fault-tolerant behavior. The following pattern presents a basic guideline to solve the interaction problem between a fault-tolerant object and its client.

Pattern: Fault-tolerant Object interaction

Fault-tolerant objects (FT objects) are highly reliable objects whose services are assured by redundant components.

Context: **Software fault-tolerance**

Redundant components can be managed according to different fault-tolerance techniques as n-version programming, recovery blocks, active or passive replication protocols and so on. Despite the technique adopted, a fault-tolerant object should interact with other objects to provide reliable services.

Problem**: Transparency**

The client requests services to a fault-tolerant object but it is not aware of the existence of redundant components used to provide the requested service. As an alternate, the fault-tolerant component can manage all its counterparts thus providing the expected service. The inclusion of redundancy within an application component is quite intrusive and it violates the separation of concerns. Hence, the provision of redundant services must be done transparently to the application objects. Both the client and the server must not be involved with this issue.

Solution**: Meta-level architecture**

The client, the fault-tolerant object and the redundant objects are implemented as base-level objects but the redundant services are managed at meta-level. Using Class-Responsibilities-Collaborators (CRC-) cards [6], the figures 3a and 3b depict the responsibilities and collaborators of base-level components

. **Class** Base Level Client	**Collaborator**
Responsibility - Requests services to a FT Object - Depends on reliable services provided by the server	FT Object

Figure 3a: Base level client

Class FT Object	**Collaborator**
Responsibility - Provides reliable services to its clients	- Metaobject - Redundant Components

Figure 3b: Base level FT Object

The fault-tolerant object is connected to a metaobject who intercepts the incoming messages. For each incoming message, the target method and the actual parameters are materialized at meta-level, and then the same message is dispatched to all redundant components. The responses to the requested service are collected at meta-level and one response is selected according to the adopted fault-tolerance technique. It can follow the first-response semantic, votation or acceptance. Finally, the selected response is delivered to the original client. Note that the clients of FT-objects do not know any of the details how those strategies are implemented, but they certainly must provide the base level components when they have diverse implementation. Figure 4a and figure 4b show the responsibilities and collaborators of the meta-level components.

. **Class** Metaobject	**Collaborator**
Responsibility - Intercepts messages - Reifies base level information - Acts as an indirect server	Meta level FT manager

Figure 4a: Meta level object

Class Meta-level FT Manager	**Collaborator**
Responsibility - Invokes the corresponding method in each version - Collects the answers - Select one result	- Redundant Components - Voter or adjudicator

Figure 4b: Meta-level FT manager

Another concern is about the fault-tolerance technique adopted by the application. The functional core of the meta-level fault-tolerance scheme should be separated into a component. The component class shall encapsulate the minimum memory size and services to manage the redundant components because it can be inherited or instantiated by the specialized metaobject associated with the base-level FT-object.

To avoid performance problems and to guarantee independent reusability, the services dealing with the multiple results and the selection of one response - the adjudicators - should be encapsulated in separate components. Note that the meta-level program is composed by several metaobjects.

They distinguish from base-level objects since they are specialists on fault-tolerance schemes and thus hold information about the application objects. However, meta-level components provide its own interface and the meta-level program has its own dynamic behavior, very much like the base-level program.

For example, considering the Recovery Block (RB-) scheme to manage redundancy. The recovery block is a control structure that consists of several alternates and an acceptance test. This structure can be represented by a code fragment as:

ensure <acceptance test>
by <alternate 1>
else by <alternate 2>
....
else by <alternate n>
else error

The execution of the recovery block terminates successfully when one result from any one of the alternates satisfies the acceptance test. When a faulty alternate is executed, the state that existed prior to the execution of the faulty alternate is restored and the next alternate is tried [15]. Resuming the design pattern example, diagrams are used in addition to CRC cards to illustrate runtime scenarios. The diagram in figure 5 shows the dynamic behavior of recovery block execution.

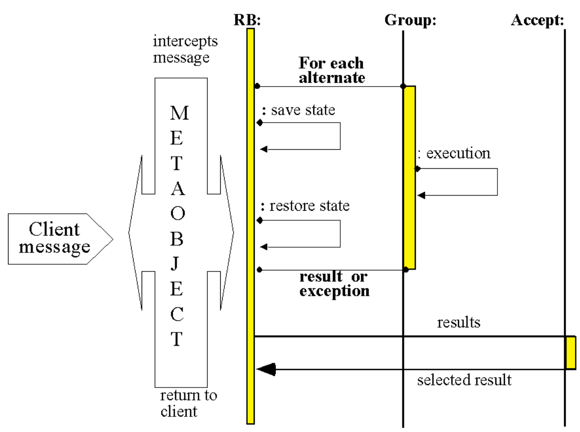

Figure 5: The Recovery Block dynamic behavior

As previously stated by [15], dealing with state restoration can be as simple as to make a copy of the original object or as complex as recovery cache memory implemented in hardware. 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. The discussion about patterns can go further but, for the moment, it is enough to think about the benefits they can bring to fault-tolerant software development.

4.3 Configuring fault-tolerant applications

When the fault-tolerance domain is merged into an application domain it becomes responsible for some non-functional requirements of the application. But, as previously discussed, there are mutual dependencies between both domains. However, the designer of the whole application need not be concerned with a "global" property of fault tolerance of the application. The designer can identify individual critical objects thus subject to fault tolerant behavior. The other objects are considered non-critical and thus not subject to fault tolerant behavior.

Here, a major point of discussion is how much fault-tolerance is needed and what kind of faults the application should tolerate – only software design faults and/or hardware faults. In trying to keep the fault-tolerant behavior as independent as possible, each critical object is represented by a FT-object and the FT-object implements the fault-tolerant strategy. A FT object interfaces a set of objects that realize the same specification. Members are redundant components subject to the same fault-tolerance strategy. They can have diverse implementation to avoid software faults or they can be replicas executing on distributed hardware, in order to increase system availability in case of hardware faults.

Since the application consists of fault-tolerant (FT-) and non-fault tolerant objects, and only fault-tolerant objects are connected to metaobjects, the meta-level program must provide information only about those aspects on which the metaobjects depend. For example, metaobjects must be aware of incoming messages addressed to critical methods within the FT objects. Also, metaobjects often need information about the current state of computation.

The following guidelines, adapted from those suggested by [6], help with designing a meta-level architecture:

1 *Define a model* of the fault-tolerant application using an appropriate analysis method. Identify the services according to the functional requirements, the components that can fulfill these services and the structure and the relationships among the components.

2. *Identify structural aspects* of the system, which, when changed, should not affect the implementation of the components. Examples include the type structure or the distribution of components.

3 *Identify fault-tolerant behavior* of the application such as atomic transactions, real-time constraints, reliable distribution, exceptional behavior and special algorithms for critical services.

4 *Identify fault-tolerant services* to be provided to guarantee the reliability, availability and safety of the whole application. Several fault-tolerance strategies can be combined to achieve these goals. Identify the systems components, which provides the services. For example, the run-time system may provide some safety/security mechanisms.

5 *Define the metaobjects* that provide the fault-tolerance services needed by the application. The metaobjects can be specially implemented to fulfill the requested services, or they can be reused from frameworks or library classes.

6 *Define the metaobject* protocol suitable to the application. The metaobject protocol can be defined on early stages of software development, when an implementation language is chosen. Otherwise, it can be customized by the user [16].

5 Conclusion

Along the previous section important remarks about the current trend in adopting software reference architecture to provide a skeleton for building application were presented. Concerning to development of fault-tolerant application, meta-level software architecture is suggested as reference architecture. Then, basic guidelines to help the development of such applications were addressed.

One could have chosen any other software architecture or other ways to develop fault-tolerant software. However, recent works dealing with software development on fault-tolerance domain serve as evidence of this trend. Those works have confidence on the object model for composing fault-tolerant applications from pre-defined components and they use reflective architecture to achieve separation between functional requirements and those related to fault-tolerance. Meta-level architecture attacks several important issues in implementing non-functional requirements in a non-intrusive, transparent way to the application. Some other advantages are:

- separation of concerns through separate high level abstractions - base level and meta-level programs;
- it provides a design that is more adaptable, flexible and easier to extend than traditional designs.

This new dimension of modularization – the separation of base level and meta-level objects – and the way they interact – transparent message interception- promotes design and or code reuse. This paper focused on the design reuse by examining design patterns, instead of single components, as architectural elements. Design patterns capture the essentials of fault-tolerance schemes without compromising with any programming language.

Acknowledgements

This work was partially supported by grants from FAPERGS and Capes.

References

[1] M. Ancona et al. Reflective architecture for reusable fault-tolerant software. In: *Latin American Conference in Informatics*, Canela-RS, pages 87-98, 1995.

[2] J. C. Fabre et al. Implementing fault-tolerant applications using reflective o-o programming. In: *IEEE International Symposium on Fault-Tolerant Computing*, pages 489-498, 1995.

[3] R. Stroud and Z. Wu. Using metaobject protocols to implement atomic data types. *Lecture Notes in Computer Science* 952:168-189 (ECOOP'95 proceedings), 1995.

[4] L. E. Buzato, C. M. F. Rubira and M. L. Lisboa. A Reflective Object-Oriented Architecture for Developing Fault-Tolerant Software. Journal of The Brazilian Computer Society. To appear, December, 1997.

[5] J. Fraga et al. Implementação de Serviços Replicados em um Ambiente Aberto Usando uma Abordagem Reflexiva. In: *XVI Congresso da Sociedade Brasileira de Computação*, Recife-PE, pages 261-272, 1996.

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

[7] N. Islam and M. Devarakonda. An essential design pattern for fault-tolerant distributed state sharing. *Comm. of the ACM*, 39(10): 65-74, 1996.

[8] D. Garlan and D. E. Perry. Introduction to the special issue on software architecture. *IEEE Trans on Software Engineering*, 21(4):269-274, 1995.

[9] M. Golm. Design and implementation of a meta architecture for Java. Diplomarbeit im Fach Informatik, Friedrich-Alexander Universität, Erlangen-Nürnberg, january 1997.

[10] JAVA Core Reflection: API and specification. Java Soft, Mountain View, CA, USA, october 1996.

[11] S. Chiba and T. Masuda. Designing an extensible distributed language with meta-level architecture. *Lecture Notes in Computer Science* 707: 482-501 (ECOOP'93 proceedings), 1993.

[12] S. Chiba. A metaobject protocol for C^{++}. *ACM SIGPLAN Notices*, 30(10): 482-50, 1995.

[13] F. Buschmann, K. Kiefer, M. Stal and F. Paulisch. The meta-information protocol: run-time type information for C++. In: *International Workshop on New Models for Software Architecture* (IMSA'92), Tokyo, Japan, pages 82-87, 1992.

[14] B. Hayes-Roth. A domain specific software architecture for adaptative intelligent systems. *IEEE Trans. on Soft. Eng.*, 21(4): 288-301, 1995.

[15] C. M. F Rubira and R. Stroud. Forward and backward error recovery in C++. *Object Oriented Systems*, 1(1): 1-85, 1994.

[16] G. Kiczales et al. Open Implementation Design Guidelines. 1997. http://www.parc.xerox.com/oi