A Meta-Level Software Architecture based on Patterns for Developing Dependable Collaboration-based Designs[1]

Delano M. Beder Cecília M.F. Rubira

Instituto de Computação - Universidade Estadual de Campinas
Caixa Postal 6176. 13083-970. Campinas - SP
e-mail:{delano,cmrubira}@dcc.unicamp.br

Abstract

The focus of this paper is the design and implementation of dependable quality requirements, and their incorporation in the description of software architecture in an explicit and structured manner. More specifically, we propose three architectural styles for developing dependable collaboration-based software designs based on three notions: (i) the idealized fault-tolerant component model, (ii) the collaboration/role interaction model, and (iii) computational reflection together with a set of design patterns that focus on providing design solutions for implementing fault tolerance techniques, namely, error handling, coordinated recovery and software redundancy. Computational reflection defines a meta-level architecture that is composed of a base level where the application's logic is implemented and a meta level where meta components are responsible for implementing the application's quality requirements in a way that it is transparent to application designers. Application designers can apply the notion of separation of concerns and concentrate their attention on the functional requirements, abstracting from the quality requirements.

1 Introduction

Software architectures define the overall structure and organization for designing software systems. Usually the software architecture is decided during the first design stage in which the basic approach to solving a specific problem is selected. The software architecture provides the context in which more detailed design decisions are made in later design stages and a software system's quality requirements (or attributes) are largely permitted or restrained by its architecture. The conception of dependable software architecture can become extremely difficult as different software quality requirements amalgamate with the functional requirements of the application (related with what needs to be done, independent of how it is done). In order to ease the task of constructing dependable software systems, is crucial to apply the engineering principle of *separation of concerns*. However, a scheme to support separation of concerns should provide: (i) separation according the multiple kinds of concerns simultaneously and (ii) overlapping/interacting concerns (not simply independent or orthogonal ones) and understanding of their mutual interference. A key goal of this work is to develop a software architecture within which multiple quality requirements related to dependability can be expressed coherently and necessary tradeoffs be made.

In this paper, we focus on the design and implementation of dependable quality requirements, and their incorporation in the description of a software architecture in an explicit and structured manner. In our proposed software architecture, we are primarily concerned with the provision of features that would facilitate the design of collaborations that are expected to cope with faults. For instance, in complex concurrent dependable applications it is interesting to incorporate explicitly in the description of their software architecture the notion of coordination to support error handling and coordinated recovery between multiple interacting components.

[1]This work is supported by CNPq and CAPES.

We propose three architectural styles for developing dependable collaboration-based software design based on three notions: (i) the idealized fault-tolerant component model [4], (ii) the collaboration/role interaction model [8], and (iii) computational reflection together with a set of design patterns that focus on providing design solutions for implementing fault tolerance techniques, namely, error handling, coordinated error recovery and software redundancy.

Computational reflection defines a meta-level architecture that is composed of a base level where the application's logic is implemented and a meta level where meta components are responsible for implementing the application's quality requirements in a way that it is transparent to application designers. Application designers can apply the notion of separation of concerns and concentrate their attention on the functional requirements, abstracting from the quality requirements.

2 Software development phases

Among the different phases of the software development process, the focus of this paper is on the design and implementation phases. Furthermore, the design phase can be subdivided into two different stages: architecture definition and detailed design. These phases are presented in this section.

2.1 Architecture definition

According to Shaw and Garlan [7], the architecture of a software system can be described by means of architectural styles, that is, as the description of *components* from which systems are built, *connectors* among those components, *patterns* that guide their composition, and *constraints* on these patterns.

The Idealized Fault-Tolerant Component architectural style

A system is defined as a set of components interacting under the control of a design. The system model is recursive in the sense that each component can itself be considered as a system on its right. If a component cannot satisfy a request for service, then it will return an exception. At each level of the system, an *idealized fault-tolerant component* [4] will either deal with exceptional responses raised by components at a lower level or else propagate the exception to higher level of the system. In this style, there are only two forms (i.e. connectors) that components can communicate to each other: service requests and services responses. Besides, an idealized fault-tolerant component can be designed as a containing component that encloses several diversely (redundant) designed sub-components called variants and an adjudicator. Variants deliver the same service through independent designs, and the adjudicator selects a single, presumably correct, result from the results produced by variants.

The Role-based Collaborative architectural style

Objects participate in many interactions, playing different *roles* in each one. Each interface (role) that a component provides only makes sense in the context of related services and interactions with other components; specifically, in the context of related interfaces of those other components. Hence it is logical to group these related interfaces together into a unit that defines one architectural design of a certain service. We call this unit *collaboration*. In this style, collaborations are the connectors among the roles.

We have identified different kinds of collaborations. Figure 1 shows the different connectors defined by this style:

- The first connector defines a simple collaboration [8]. That is, a group of objects that cooperate to perform a task. A role is the part of a object that fulfills its responsibilities in the collaboration. This connector does not concern the provision of dependability requirements.
- If was determined that there no interactions between that group of objects and the rest of system for the duration of collaborative activity, then this activity should be specified as an atomic action [4]. Atomic actions provide error damage confinement since they guarantee that no information is smuggled to or from the collaboration.
- If the collaborations are expected to cope with faults, then it is interesting to incorporate explicitly the notion of coordination to support error recovery between multiple interacting objects. Besides, different error recovery techniques can be used: only backward (for example, conversations [6], only forward or one combination of these [1].
- If the collaborations are expected to deal with both cooperative and competitive concurrency (for example, coordinated atomic actions [9]), then the collaboration should implement coordinated error recovery and maintain the consistency of external resources in the presence of failures and concurrency among different collaborative activities competing for these resources.

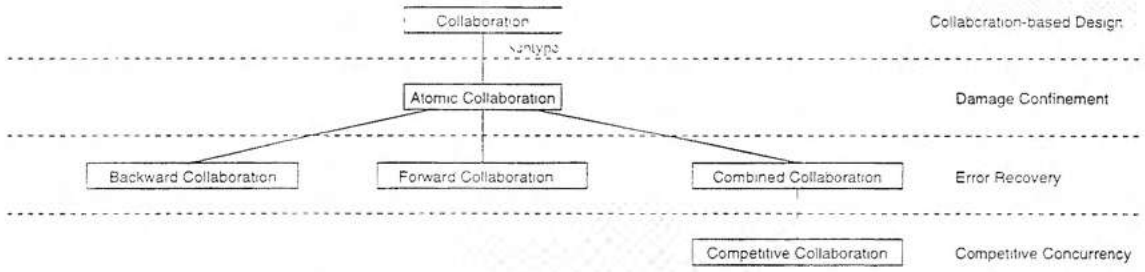

Figure 1: Different kinds of collaborations

The Meta Level architectural style

This style separates the system components in at least two levels (or layers): the meta level and the base level. The meta level encompasses the components (objects) that deal with the processing of self-representation and management of an application, and the base level encompasses the components (objects) responsible for implementing the functionality of the application. The connector in this style is the meta-object protocol (MOP). The interactions between base-level and meta-level objects are realized through a meta-object protocol which establishes the allowable design rules that guide the construction of a system organized with this style. Meta-level architectures address separation of concerns, providing means to implement quality requirements of an application transparently separated from its functional ones.

Furthermore, in order to achieve structured composition of quality requirements, we propose a special kind of connector (meta object) called *delegator*. It delegates operations and results to other meta objects. In this way, base-level objects can be directly associated with a meta object that encapsulates (composes) several quality requirements. This connector defines the semantics of such compositions. That is, it defines the possible composition rules among quality requirements.

2.2 Detailed design

A design pattern names, abstracts and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design [2]. Design patterns refine the

general components of an architectural style, providing the detailed design solutions. Usually, the selection of a design pattern at this phase is influenced by the architectural styles that were previously chosen at the architecture definition phase.

2.2.1 Idealized fault-tolerant components

The **Reflective Contract**, **Error Handling** and **Software Redundancy** patterns provide design solutions for implementing idealized fault-tolerant components. Considering that a system is not free from faults, exceptions may be produced as responses of requests that cannot be satisfied due to component faults. The **Reflective Contract** pattern provides one design solution for detect errors caused by the occurrence of faults. Application designers structure a set of **contract** classes, which check the preconditions and postconditions of each component service, returning exceptions if these conditions are not satisfied. The **Error Handling** pattern provides the explicit separation between the normal and error-handling activities of an idealized fault-tolerant component. Application designers structure their components by creating normal and exceptional classes. The normal classes consist of a collection of methods which implement component's normal services, while the methods of exceptional classes are the handlers for the exceptions raised during the execution of normal services. An idealized fault-tolerant component can be designed as a containing component that encloses several diversely designed sub-components called variants. The **Software Redundancy** pattern supports the disciplined and non-intrusive design of variants, that is, components that have the same interface but they are implemented by several different designs.

Dynamics: Meta objects are responsible for implementing the application's quality requirements. In order to achieve structured composition of quality requirements, a special kind of meta object called **Delegator** was defined. Figure 2 shows how such quality requirements can be composed in the design of idealized fault-tolerant components. In this example, **Supplier** is associated to a meta object (**Delegator**) that composes three meta objects.

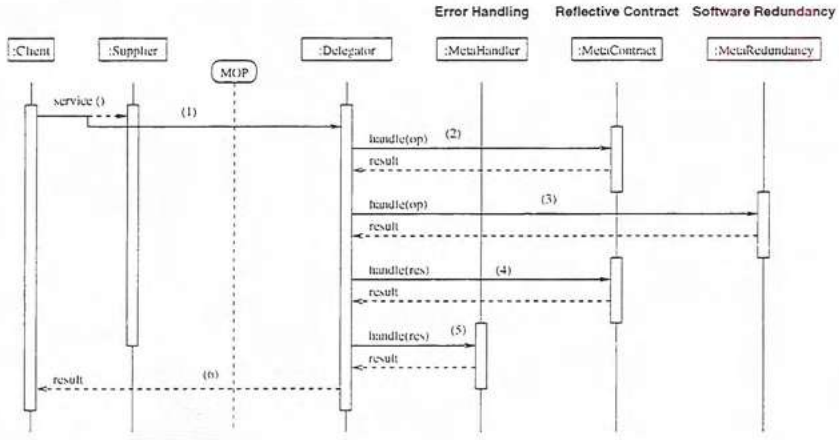

Figure 2: Dynamics: Idealized fault-tolerant components

1. While working on this task, **Client** asks **Supplier** to carry out a service;
2. Delegator intercepts this method invocation and it delegates the service invocation to **MetaContract** that checks the preconditions of this service returning an exception if these are not satisfied.
3. If the result is an exception then **Delegator** delegates this exception to **MetaHandler** (step 5). Otherwise, it delegates the service invocation to **MetaRedundancy** that execute the

variants and return the single result of these variants. If each variant was executed unsuccessfully, then an exception is returned.
4. **Delegator** receives this result and then delegates it to **MetaContract** that checks the postconditions returning an exception if these are not satisfied.
5. **Delegator** receives this result and then delegates it to **MetaHandler**. If the result is not an exception, it returns the same result. Otherwise, it invokes the handler of this exception and returns the result of this handling.
6. **Delegator** receives this result and then returns.

2.2.2 Collaboration-based design

The **Collaboration** and **Reflective Role** patterns provide design solutions for implementing simple collaborations. The **Reflective Role** pattern captures the role concept. This pattern adapt an object to different designer's needs through transparently attached role objects, each one representing a role the object has to play in different contexts. The **Collaboration** pattern provides the design solution for implementing the first connector among roles shown in Figure 1. We have defined other 5 patterns that present design solutions for implementing the other connectors but due to space limitation, only present one of these.

Competitive collaboration

The **Lock Server** [3] pattern provides a controlled concurrent access to shared resources. Before a client accesses a shared resource, it has to acquire a certain lock for this resource from the lock server. After the client has completed its work with the resource it should release the lock in order to allow other clients access the resource fairly. The **Competitive Collaboration** pattern ensures that there are no interactions between that group of roles and the rest of the system. It asks the locks for accessing the related roles before starting the collaborative activity and releases them after the collaborative activity has been finished.

This pattern incorporates explicitly the notion of coordination to support coordinated error recovery between multiple interacting components. The error recovery technique used is one combination of backward and forward error recovery. When one error is detected, it first try to recover by using exception handling, but if it is not possible then it try to bring the system back to a state prior to error occurrence.

Due to the nature of concurrent systems, various exceptions may be raised concurrently while components are cooperating. Thus, a mechanism of exception resolution is necessary in order to agree on which exception to be notified to all collaboration participants. The work of Campbell and Randell [1] describes a resolution model called exception tree that allows us to find the exception that represents all exception raised concurrently. The **Cooperating Exception** pattern provides support to application designers define such exception trees.

Backward error recovery involves the establishment of recovery points, which are points in time during the execution of a component for which the then current state may be subsequently need to be restored. The **Memento** pattern [2] captures and externalizes an object's internal state that can be restored later. The **Competitive Collaboration** pattern establishes the recovery points by using the services provided by the **Memento** pattern before starting the collaborative activity. If some error is detected, then the role's states are restored to the states saved previously. Roles can be designed as idealized fault-tolerant components (section 2.2.1). Contracts can be used to implement error detection (**Reflective Contract** pattern) and each role can be structured by enclosing several diversely designed variants (**Software Redundancy** pattern). The collaborative activity starts by execute the first variant of each role. If some error is detected, then each role is restored to state prior to error occurrence and the other variants of

these roles are tried. Exceptional classes defined by **Error Handling** pattern are the handlers for the exceptions raised during the collaborative activity.

This pattern is also expected to deal with competitive concurrency. The combination of **Memento** [2] and **Lock Server** [3] patterns provides one design solution for implementing transactional semantics on external objects. These semantics are responsible by maintaining the consistency of these external objects in presence of failures and competitive concurrency. Base-level objects are also associated to a delegator that composes several quality requirements. Due to space limitation, the dynamics of this composition is not presented.

2.3 Implementation

Designers reuse our software architecture design for the needs of their application by subclassing the base-level classes of our software architecture (white-box components) or associating non-specialized base-level objects of the application with instances of meta-level classes of our software architecture (black-box components). The association between base-level and meta-level objects is automatic since the designers provide some configuration files that are read in order to initialize the meta-configuration associated with the application. We utilized the meta-object protocol called Guaraná [5] that supports the definition of a special kind of meta object called **composer** that delegates operations and results to other meta objects. Composers can delegate to meta objects sequentially or following whatever policy fits the needs of a developer. In our software architecture, the delegation is based on the semantics of quality requirements composition. If another meta-object protocol was utilized, then the meta object (delegator) that delegates operations and results should had been implemented.

3 Concluding remarks

In this paper, we have focused on the design and implementation of dependable quality attributes, and their incorporation in the description of a software architecture in an explicit and structured way. As future work, we identify the following: there are some kinds of applications, for example, secure electronic financial transactions, where the collaboration participants have to be authenticated in order to avoid attacks. Thus, we plan investigate how much is practicable add quality requirements related to security into our software architecture.

References

[1] R. Campbell and B. Randell. Error Recovery in Asynchronous Systems. *IEEE Transactions on Software Engineering*, SE-12(8):811–826, Aug. 1986.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison Wesley Publishing Company, 1995.

[3] R. Hirschfeld and J. Eastman. Lock Server. In *4th Pattern Languages of Programming Conference (PloP'97)*, 1997.

[4] P. Lee and T. Anderson. *Fault Tolerance: Principles and Practice.* Springer-Verlag, 2 edition, 1990.

[5] A. Oliva and L. Buzato. Composition of Meta-Objects in Guaraná. In *Workshop on Reflective Programming in C++ and Java, OOPSLA'98*, pages 86–90, Vancouver, BC, Canada, Oct. 1998.

[6] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, 1975.

[7] M. Shaw and D. Garlan. *Software Architecture: Perpectives on an Emerging Discipline.* Prentice Hall, 1996.

[8] Y. Smaragdakis and D. Batory. Implementing Reusable Object-Oriented. In *5th International Conference on Software Reuse (ICSR'98)*, Victoria, Canada, June 1998.

[9] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In *FTCS-25*, pages 499–509, 1995.