An Aspect-Based Approach to Modeling Fault Tolerance Concerns

Robert France
Department of Computer Science, Colorado State University
Fort Collins, USA
france@cs.colostate.edu

Geri Georg
Agilent Technologies, Fort Collins, USA
geri_georg@agilent.com

Abstract

In this paper we show how design-level aspects can be used to separate fault tolerance concerns from essential functional concerns during software design. The aspect-oriented design technique described in this paper allows one to independently specify fault tolerance and essential functional concerns, and then weave the specifications to produce a design model that reflects both concerns. We illustrate our technique using a small example.

1 Introduction

Aspect-oriented programming (AOP) is concerned with localizing cross-cutting concerns (e.g., exception handling, data logging), or *aspects*, to better manage program evolution and to facilitate reuse of the cross-cutting concerns (e.g., see [6, 5]). AOP can be viewed as an outgrowth of efforts concerned with developing programming language support for the separation of concerns design principle. In AOP, the units of modularization are not restricted to functional abstractions. From a design perspective one can view an aspect as a concern that cross-cuts the primary design modularization. In particular, an aspect approach to design allows one to isolate cross-cutting reliability concerns from a system's essential functional design.

In our approach, an *aspect-oriented design* (AOD) consists of a model of essential functionality and a number of aspect models each modularized around a specific concern. In this paper, we focus on aspects that reflect reliability concerns (specifically, fault tolerance). An integrated model of the system can be obtained by "weaving" (composing) the model of essential functionality with the aspect models.

Treating reliability concerns as aspects during design has the following advantages:

- Aspects allow one to understand and communicate reliability concerns in their essential forms, rather than in terms of a specific application's behavior.
- The reliability aspects are potentially reusable across different systems in an organization. One can express reliability policies and procedures that are intended to be applied across a family of systems as aspects.
- Changes to reliability concerns are made in one place (aspects), and effected by weaving the aspects into the models of essential functionality. This eases the management of change: changes in reliability approaches can be handled by changing the respective aspects and weaving them into the models of essential functionality.

In order to realize the above benefits we felt it was important that aspects be defined in an application-independent manner (to the extent that this is possible). This allows relatively independent development of logical design models of functionality and aspects. This is unlike some approaches to AOP (e.g., see AspectJ [5]) in which development of aspects requires detailed knowledge of the programs.

Such dependency limits reusability of aspects and constrains how and when aspects are developed.

In our AOD approach an aspect is defined in terms of structures of roles called *Role Models* [3, 4]. A role is a property-oriented specification that conforming model elements must satisfy. Weaving an aspect into a model involves modifying targeted elements of the model such that they conform to the properties expressed in the aspect's Role Models. This can be done by (1) extending targeted model elements so that they conform to roles or (2) generating conforming model elements from the roles. The models used in this paper are expressed in the UML (Unified Modeling Language [8]).

A number of authors have tackled the problem of defining and weaving aspects at the design level (e.g., see [1, 2, 7]), but these works fall short in defining a precise notation for expressing aspects. The approach in this paper supports a more rigorous approach to aspect definition and weaving in AOD.

In section 2 we give an overview of Role Models and describe how they can be used to precisely express aspects. We illustrate our approach to modeling aspects by developing Role Models for two fault tolerance mechanisms. In section 3 we illustrate our approach to AOD by weaving the fault tolerance aspects into UML static and interaction models of a simple Order Entry System (OES). We conclude in section 4 with an overview of our plans to further develop this work.

2 Modeling Aspects using Role Models

We use Role Models (see [4]) to define aspects in an application-independent manner. Roles are used to define properties of aspects. A model element conforms to a role if it possesses the properties defined in the role. Weaving an aspect defined by Role Models is essentially a model transformation process in which a non-conforming model is transformed to a conforming model (i.e., a model that incorporates the aspect). One can view a Role Model as a characterization of model element structures that incorporate the aspect.

The UML is used as the design modeling notation in our work, thus the Role Models we developed are property-oriented characterizations of conforming UML models [4]. A UML model element (e.g., a class or an association) that has the properties specified in a role can *play the role*, that is, it *conforms to* (or realizes) the role. A UML model is said to conform to (or realize) a Role Model (i.e., is a realization) if it consists of model elements that conform to the roles in the Role Model.

A design aspect can be modeled from a variety of perspectives. In this paper we focus on two aspect views: static and interaction views. An aspect's static view focuses on the structure of the aspect. The interaction view focuses on the interactions that take place within the aspect.

To facilitate weaving, the Role Models in this paper are constructed in a manner that allows one to generate conforming structures from them. Weaving a Role Model into a UML model, M, can involve (1) merging roles with model elements M, that is, modifying model elements in M so that they conform with the roles and (2) generating new model elements from roles and inserting them into M.

In previous work we developed two types of Role Models: *Static Role Models* (SRMs) and *Interaction Role Models* (IRMs) [4]. SRMs characterize UML static structural models (e.g., Class Diagrams), while IRMs characterize families of UML interaction diagrams (e.g., Collaboration Diagrams). In the following subsections we show how specialized forms of SRMs can be used to define aspects from a static perspective and how IRMs can be used to define aspects from an interaction perspective. An aspect definition typically consists of a single SRM and one or more IRMs.

2.1 An Overview of SRMs

An example of a SRM characterizing Observer-Subject static models, and a conforming model are shown in Fig. 1. *Observer* and *Subject* are class roles, while *Observes* is an association role. This SRM characterizes class diagrams in which an observer class (i.e., a class that conforms to the *Observer* role) is linked, via an association that conforms to the *Observes* role, to a subject class (i.e., a class that conforms to the *Subject* role). The *base* of a role determines the type of UML model elements that can play the role. In an SRM the base must be a classifier type (e.g., Class, Interface) or a relationship type (e.g., Association, Generalization) in the UML metamodel. The role *Subject* shown in Fig. 1 has a base *Class* indicating that only UML class constructs can play this role. The properties specified in the *Subject* role determine the form of classes that conform to the role. A role can consist of two types of properties: *Metamodel-level costraints* and *Feature roles.* Feature roles are associated only with classifier roles in our work.

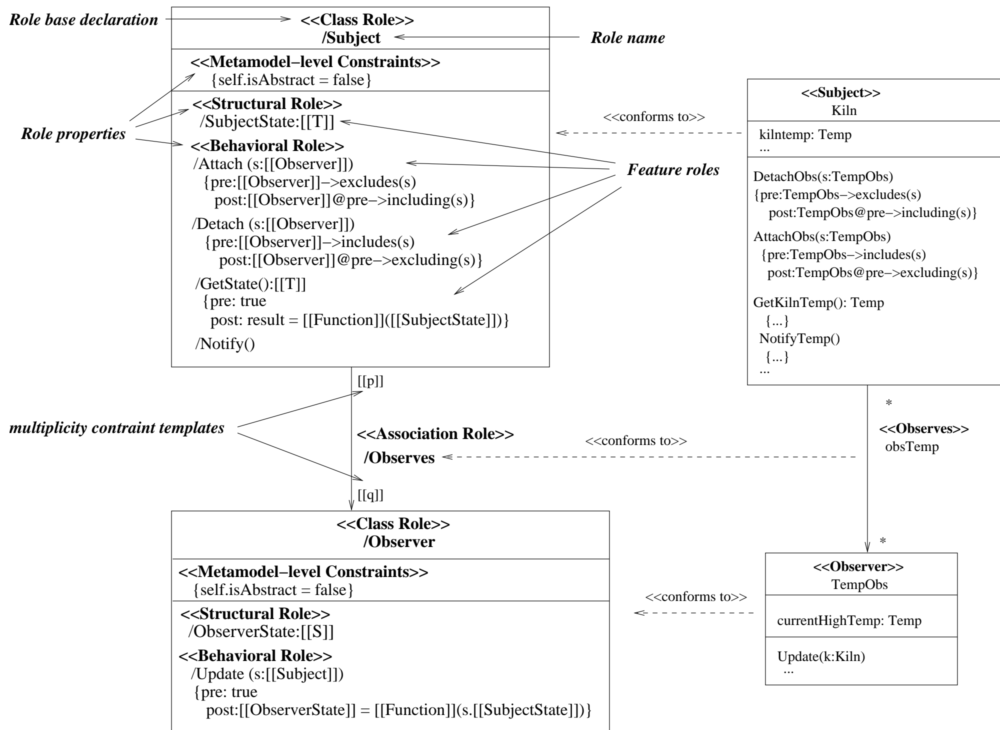


Figure 1. A SRM Characterizing Observer-Subject Static Models

Metamodel-level constraints are well-formedness rules that constrain the form of UML constructs that can realize the role. They are expressed as predicates (expressed in the Object Constraint Language (OCL) [8]) over the UML metamodel elements. The metamodel-level constraint of *Subject* states that classes that conform to the role must not be abstract.

Feature roles characterize properties that must be present in the model elements that conform to the enclosing SRM role. A feature role consists of a name, and an optional property specification expressed as a *constraint template*. Realizations of a feature role have properties that imply the property generated by appropriately instantiating the feature role's constraint template.

There are two types of feature roles: *Structural roles* (e.g., $SubjectState$ in *Subject*) specify state-related properties that are realized by attributes or value-returning operations in a SRM role realization, while *behavioral roles* specify behaviors that can be realized by a single operation or method, or by a composition of operations or methods in a SRM role realization. The attribute *kilntemp* in $Kiln$ conforms to the structural role $SubjectState$ in *Subject* after substituting $Temp$ for T. *AttachObs* and *DetachObs* conform to *Attach* and *Detach* respectively because their specifications imply the specifications obtained by substituting *TempObs* for *Observer* in the constraint templates of *Attach* and *Detach*.

From the above, a realization of the *Observer* role must (1) be a concrete class ($self.isAbstract = false$), (2) have a realization of $ObserverState$, and (3) have behaviors that realize $Update$. A realization of the *Subject* role must (1) be a concrete class ($self.isAbstract = false$), (2) have a realization of $SubjectState$, and (3) have behaviors that realize *Attach*, *Detach*, *GetState*, and $Notify$.

The *Observes* association role is expressed as a template, thus a conforming association can be generated from it by substituting a set of ranges for p and q (in the UML multiplicities are sets of ranges).

In summary, a model element conforms to a role if (1) it satisfies the metamodel-level constraint and if (2) the constraints associated with its features imply the constraints obtained by appropriately instantiating the role's constraint templates. In this paper, a stereotype with a role name (e.g., $\ll Subject \gg$) in a model construct is used to in-

dicate that the model construct is a realization of the role. These stereotypes are printed in bold to distinguish them from other UML- and user-defined stereotypes (our use of stereotypes to indicate the relationship between UML model constructs and roles is not a typical usage of stereotypes; stereotypes used in this manner sometimes result in UML constructs with more than one stereotype).

Conforming model elements can also be generated from a SRM classifier role as follows:

- Create an instance of the role base that satisfies the metamodel-level constraints.
- For each structural role, generate an attribute and associated constraints by substituting a name for the role name, and appropriately instantiating type and constraint templates.
- For each behavioral role, generate an operation and associated constraints by substituting a name for the role name, and appropriately instantiating the pre and postcondition constraint templates.

2.2 Modeling Aspects as SRMs

The static view of a fault tolerance aspect is shown in Fig. 2 (role properties are suppressed in this SRM). This aspect is concerned with achieving fault tolerance by replicating resources that are repositories of entities. We refer to this aspect as the *Replicated Repository* aspect. In this aspect, *RepUser* is intended to be played by classes representing repository users, *Repository* is intended to be played by classes, with multiplicities 2..∗, represent repositories, *Entity* is intended to be played by classes that represent the contents of repositories, and *RepContainer* is intended to be played by classes representing containers of replicated repositories. The association roles shown in Fig. 2 are associated with constraints that restrict the values that can be substituted for n, q, s and t (the constraints are not shown in the diagram). Some of these constraints are informally expressed below (OCL can be used to precisely express these constraints)

- q, s, t and n are restricted to singleton sets of multiplicities (e.g., $q \rightarrow size = 1$).
- The start (lower end) of the ranges for s and t must be greater than 0 (i.e., the container of replicated repositories must have at least one user, while a user must be related to at least one container of replicated repositories).

Some of the other constraints defined in the Role Model are informally stated below:

- Each repository container (i.e., instances of classes that conform to *RepContainer*) is linked to each repository that has been created but not yet destroyed.
- An entity stored in a repository is also stored in each other replicated repository.
- *RepUser* has behavioral roles such as *AddEntity* (adds an entity to each replicated repository), *DeleteEntity* (deletes an entity from each replicated repository), and *RetrieveEntity* (returns an entity from the repository; the entity is not deleted from the repository it is retrieved from).
- *RepContainer* has behavioral roles such as *repAddEntity*, *repDeleteEntity* and *repRetrieveEntity* that manipulate specific repositories in a container of repositories. Similarly *Repository* characterizes behaviors for adding, deleting and retrieving repository contents.

Another fault tolerance aspect is shown in Fig. 3. The *Redundant Controller* aspect is concerned with introducing redundant controllers into a system to provide fault tolerance: only one controller (the *activeController*) is active at any given time; if it fails another controller from the set of redundant controllers takes control. The multiplicity parameter is restricted as follows: p must be the symbol '*' or an integer greater than 0.

2.3 IRMs: Modeling the Dynamic View of Aspects

UML interaction models (e.g., Collaboration and Sequence Diagrams) are used to specify or describe interactions between system parts. Interactions characterized by an aspect are defined using a template form of interaction models called *Interaction Role Models* (IRMs) [4]. An IRM consists of UML collaboration role, link, and message templates. Instantiating these templates results in a partial Interaction Diagram that can be woven into a design Interaction Diagram. In this paper we discuss only the Collaboration Diagram template form of IRMs.

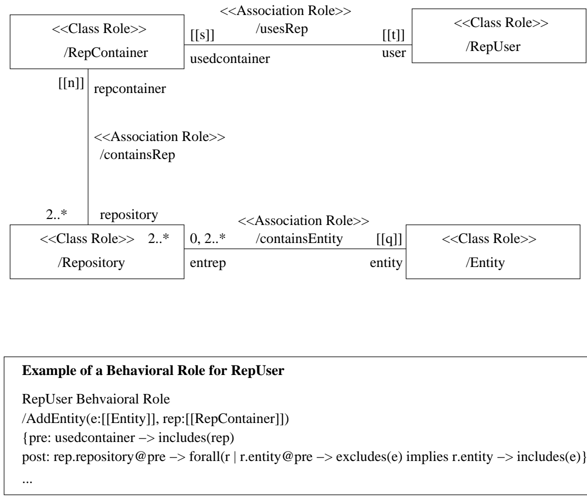


Figure 2. The Replicated Repository Fault Tolerance Aspect - Static View

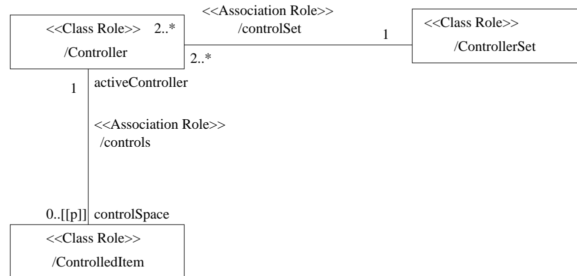


Figure 3. The Redundant Controller Fault Tolerant Aspect

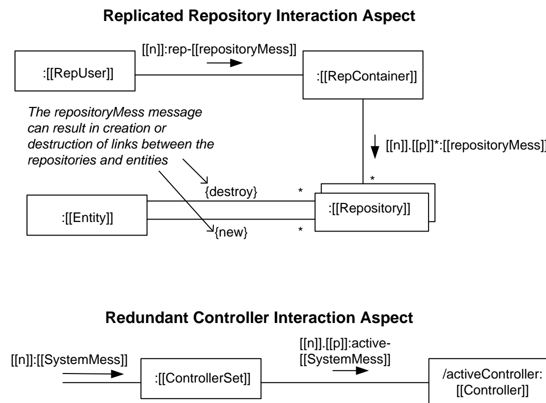


Figure 4. The Interaction Views of the Replicated Repository and Redundant Controller Aspects

We use IRMs to describe the interaction pattern defined by an aspect. For example, one can describe the interactions that take place when a client (user) needs to interact with a repository in the presence of replicated repositories or in the presence of redundant controllers as shown in Fig. 4. In the *Repli-*

cated Repository Interaction Aspect, the client (an instance of a class that conforms to *RepUser*) sends a message to the repository container (a container of replicated repositories). The container then sends the message to each repository in the container. In the *Redundant Controller Interaction Aspect*, messages are sent to the controller set, which then send the messages to the active controller. In the next section we illustrate how SRMs and IRMs can be used to support aspect weaving.

3 Weaving Aspects into a Design Model: An Example

In our AOD approach, a weaving is defined in terms of two elements: (1) a characterization of the models into which an aspect is to be incorporated (referred to as *source models*), and (2) a characterization of model transformation sequences that, when applied to source models, result in models containing the aspect. We give only informal characterizations of these elements in this paper. In this section we illustrate the weaving of the Replicated Repository and Redundant Controller fault tolerance aspects into a simple Order Entry System (OES).

3.1 Weaving Aspects into Static Models using SRMs

The static structure of the OES is modeled by the Class Diagram shown in Fig. 5. The OES design consists of two controllers: an order entry controller *OrderEntryController* and an inventory controller *InventoryController*. Orders consist of multiple line items (instances of *OrderLineItem*) and are stored in the order repository (instance of the singleton class *OrderRepository*). Order line items are filled by products (instances of *StockProduct*) stored in the inventory (instance of the singleton class *Inventory*). Products carried by the organization are described in catalogs (instances of *ProductCatalog*).

The original OES is defined as having an order entry controller, an inventory controller, an order repository, an inventory, and any number of product catalogs. The controllers manage access to their associated repositories (the order and inventory repositories). A specific order line item refers to a single type of product (product types are described by instances of *ProductDescription*). During invoicing products are allocated to the order line item. Since the class *StockProduct* represents physical products,

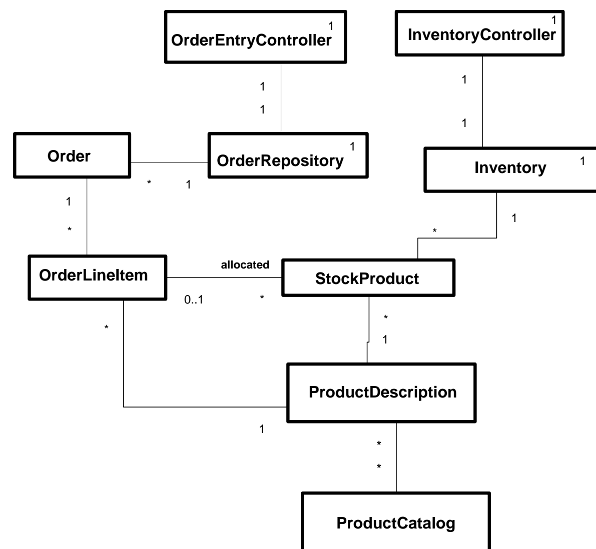


Figure 5. Design Class Diagram for the OES

each stock product must be associated with at most one line item. A stock product can only be associated with one product description.

Fig. 6 shows the result of weaving the static view of the Replicated Repositories aspect into the original OES Class Diagram. Weaving of the Replicated Repository aspect into a (source) static model that consists of singleton repository classes and singleton repository user classes, but no replicated repositories, is described by the following informal characterization of a transformation sequence:

1. Identify and augment the classes that are intended to play the *RepUser*, *Repository* and *Entity* roles, and associations that are intended to play the *containsEntity* role. Augmentation is necessary if the classes that are intended to play these roles do not have all the features defined in the roles, in which case the features are generated from the feature role properties after suitably instantiating their constraint templates. For example, the multiplicity of the classes playing the *Repository* role must be changed from 1..1 to 2..∗, that is, the multiplicities of *OrderRepository* and *Inventory* must be changed to 2..∗. The associations that are intended to play the *containsEntity* role have their multiplicities at the repository ends changed to $0, 2..*$. For example, the associations between *Order* and *OrderRepository* and

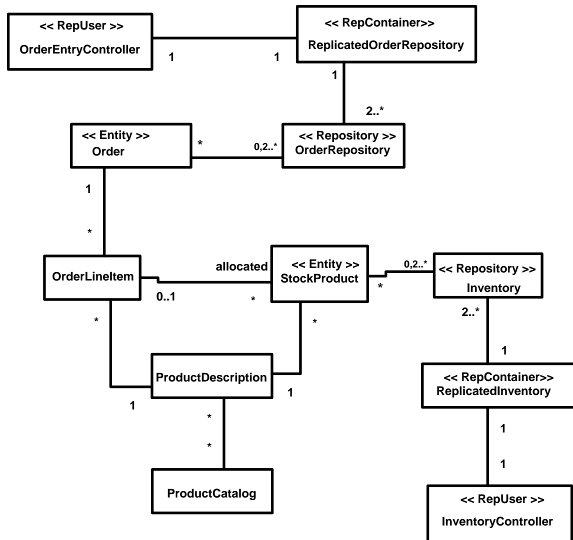


Figure 6. Result of Weaving The Replicated Repository Aspect into the OES Class Diagram

between *StockProduct* and *Inventory* have multiplicities $0, 2..*$ at the repository ends.

2. For each repository class identified in the previous transformation generate model elements that conform to the *usesRep* and *containsRep* association roles and the *RepContainer* class role. Generation involves appropriately instantiating the constraint templates associated with the roles. The resulting class structure is added to the original Class Diagram.

3. Remove the original associations between the repository user classes (e.g. *OrderEntryController*) and the repository classes (e.g., *OrderRepository*). These associations are replaced by the associations conforming to *usesRep* added in the previous transformation.

Fig. 7 shows the result of weaving the static view of the Redundant Controllers aspect into the result of the previous weaving. In this case we have chosen to have redundant controllers for only the *OrderEntryController*.

3.2 Weaving Aspects into Interaction Models using IRMs

Fig. 8 shows the Collaboration Diagram describing the main design flow for adding an order from

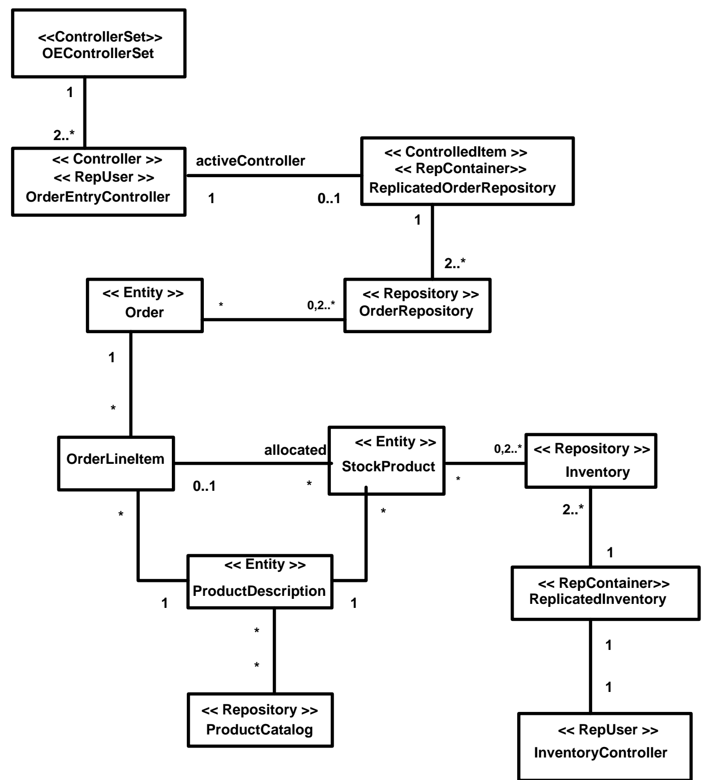
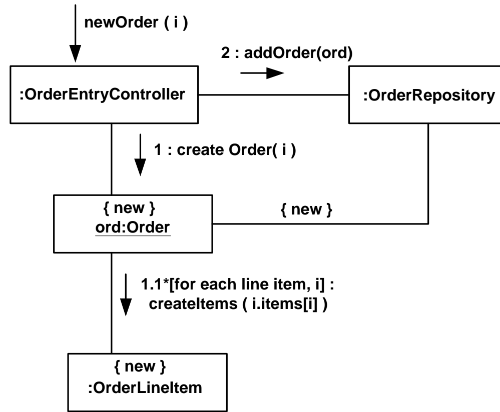


Figure 7. Result of Weaving The Redundant Controller Aspect into the Replicated Repositories OES Class Diagram

the order repository, and the result of weaving the Replicated Repository aspect interaction view into this model. The weaving is accomplished as follows:

- Delete the link and interaction between the *OrderEntryController* and the *OrderRepository* collaboration roles. This interaction will be replaced by the interaction between the repository user and the repository container in the next step.

- Add interactions between *OrderEntryController* and the repository container, and between the repository container and the repositories as characterized by the Replicated Repositories IRM. This is done by substituting the classes that play the SRM roles indicated in the collaboration role templates and by carrying out the following additional substitutions in the IRM: (1) let n = 2 and p = 1, and (2) let $repositoryMess = addOrder(ord)$.



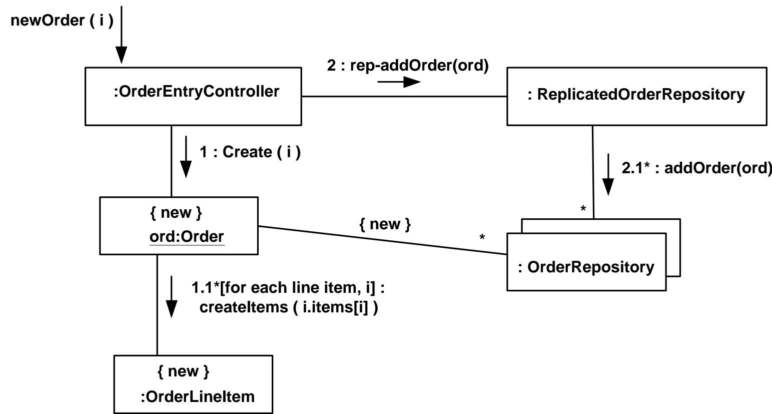


Figure 8. Adding an order in the OES

Fig. 9 shows the result of weaving the interaction view of the Redundant Controller aspect into the Collaboration Diagram produced by the previous weaving. In this case, the weaving resulted in a renumbering of the original interactions. The model in Fig. 9 is obtained by substituting, in the IRM, 0 for n (indicating that the first interaction is the outermost interaction and is thus unlabeled)), 1 for p ($0.n$ corresponds to the label n), and renumbering the other interactions by giving each an outer interaction, 1.

4 Conclusion

In this paper we (1) describe how aspects can be expressed as Role Models that define patterns of design structures, and (2) show how these aspects can be woven into designs expressed in terms of the UML. We are currently evaluating this AOD approach by applying it to some industrial strength systems. Thus far we have used this approach to define and (manually) weave fault tolerance, and security aspects into UML models of sample applications from industry.

Our experience indicates that flexible tool support for weaving will greatly enhance the practicality of our approach. Flexibility in the weaving process is necessary because the manner in which aspects are woven is highly dependent on the form of the source models and the type of aspect being woven. Thus far we have been able to algorithmically describe the weaving of the types of aspects we have

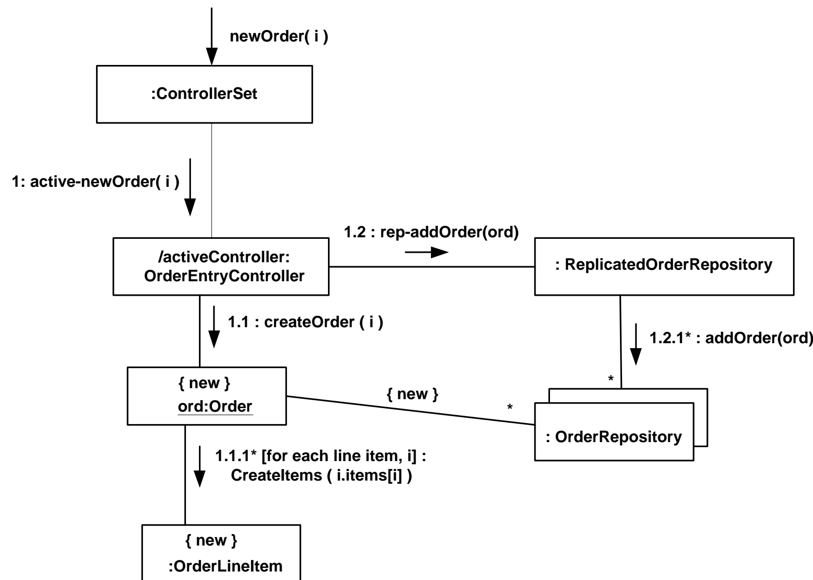


Figure 9. Adding an Order in the Presence of Redundant Controllers

developed. We are currently developing a prototype tool that will support flexible weaving, by providing users with a language for describing (reusable) weaving strategies and weaving procedures.

References

[1] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Separating concerns throughout the development lifecycle. In *Proceedings of the third ECOOP Aspect-Oriented Programming Workshop*, 1999.

[2] S. Clarke and J. Murphy. Developing a tool to support the application of aspect-oriented programming principles to the design phase. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 1998.

[3] R. B. France, D. K. Kim, and E. Song. Patterns as precise characterizations of designs. Technical Report 02-101, Computer Science Department, Colorado State University, 2002.

[4] R. B. France, D. K. Kim, E. Song, and S. Ghosh. Using roles to characterize model families. In *Proceedings of the Tenth OOPSLA Workshop on Behavioral Semantics: Back to the Basics*, 2001.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume LNCS 1241, June, 1997.

[7] J. Suzuki and Y. Yamamoto. Extending UML with aspects: Aspect support in the design phase. In *Proceedings of the third ECOOP Aspect-Oriented Programming Workshop*, 1999.

[8] The Object Management Group (OMG). Unified Modeling Language. Version 1.3, OMG, http://www.omg.org, June 1999.