Using Subject-Oriented Modeling to Develop Jini Applications

Gagan Tandon and Sudipto Ghosh
Computer Science Department
Colorado State University
Fort Collins CO 80523, USA
Email: {gagan, ghosh}@cs.colostate.edu

Abstract

A major contributing factor to the complexity of creating and evolving distributed systems is the tangling of middleware-specific functionality with core business functionality in system designs. Changing middleware functionality that is entangled with business functionality can lead to costly and risky rearchitecting of the system or extensive redesign of parts of the system. The subject-oriented software development approach addresses this problem by separating the design of crosscutting features into design subjects. In this paper we describe an approach for separating Jini middleware features as design subjects which can be composed with primary design subjects that realize the core functionality of the application. In this context, we identify limitations in the existing specification notation and propose extensions.

Keywords: *Model Driven Architecture, distributed computing, modeling and meta-modeling, service-oriented architecture and design, Jini, UML, OMG, middleware platforms, subject-oriented modeling, composition patterns.*

1. Introduction

Advances in network and middleware technologies have spawned a new generation of distributed software-intensive systems. As organizations seek to gain competitive advantage through innovative service offerings and to satisfy growing demand for sophisticated services, developers are under increasing pressure to quickly evolve distributed systems to take advantage of new middleware technologies.

Middleware technologies provide abstractions that help reduce coupling between clients and servers and provide some level of programmatic transparency. However, current techniques used to develop distributed systems do not fully support the separation of concerns needed to decouple evolution of the middleware platform from evolution of the business functionality. Lack of support for separation of business and middleware concerns forces developers to consider and incorporate technology-specific middleware elements into design artifacts that address business-specific functionality. For example, client-server systems include middleware functionality that enables clients to access remote services in a transparent manner. In peer-to-peer applications, each peer has middleware functionality that enables it to access other peers on the network. Even though certain middleware services may be provided as components, there are other middleware features that crosscut components in the application's software architecture. Evolution of distributed systems is particularly difficult when middleware functionality is tangled with core business functionality. A decision to evolve a system to take advantage of a new distributed system platform can result in costly and risky rearchitecting of the system when platform-specific and platform-independent functionality are intertwined in a system. Similarly, changes in business functionality can result in extensive changes to the system.

Existing object-oriented and component-based techniques do not provide the separation of concerns mechanisms needed to support evolution of distributed systems with changes in middleware technologies. There is a move towards the use of techniques, such as modeling and programming using the aspect-oriented and subject-oriented paradigms, that support the encapsulation of crosscutting features. In this paper we describe the use of subject-oriented modeling to design a distributed application that uses Jini middleware [6]. We define Jini design subjects that model Jini service lookup, service binding and leasing. We describe the construction of a complete application using these subjects.

The remainder of this paper is structured as follows. We summarize related work in Section 2. We give an overview of the subject-oriented modeling approach in Section 3. We describe the use of this approach to develop Jini applications in Section 4. We also point out some limitations with the existing notation and propose extensions. We conclude and outline directions for further research in Section 5.

2. Related Work

Middleware features at the code level can be described using an aspect oriented programming (AOP) language [1, 10, 11, 12, 13, 16, 17, 19, 20]. An AOP aspect is an implementation of a design concern that cross-cuts the primary functional units of a program. Bussard [2] encapsulated several CORBA services as aspects using AspectJ to make CORBA programming transparent to programmers. Hunleth, Cytron, and Gill [9] suggest the creation of an AspectIDL for CORBA to complement the IDLs that are now available for languages such as Java and C++. Their proposed AspectIDL supports several new types of AspectJ *introductions*: interface method and field, interface super class, structure field, oneway specifier, and IDL typedef and enumerations.

The Model Driven Architecture (MDA) [14] initiative recognizes the need for raising the level of abstraction at which developers describe complex systems and advocates the use of model-centric approaches to developing complex systems. Often, companies use the time-to-deliver pressure as an excuse to adopt code-centric approaches to software development. However, developers are finding it increasingly difficult to cope with the complexity of developing secure, fault-tolerant, highly available distributed software systems using only code level descriptions. MDA advocates separation of technology independent concerns from technology specific concerns. A platform-independent model (PIM) describes the behavior of the application "undistorted by the idiosyncrasies of the technology or technologies in which it will be implemented" [14]. A platform specific model (PSM) describes an application in technology specific terms. An MDA design consists of PIMs, PSMs, and PIM to PSM mappings. The mappings, ideally implemented in tools, transform the PIMs to PSMs. Use of an MDA approach makes it unnecessary to repeat the process of modeling an application's functionality and behavior each time a new implementation platform comes along.

Researchers have started to address the problem of describing and using aspects at an abstraction level higher than the code level (e.g., Clarke, Harrison, Ossher and Tarr [4], Clarke and Murphy [5], Gray et al. [8], and Rashid et al. [18, 21]). France et al. use aspect-oriented modeling and composition at the design level [7]. Clarke et al. [4, 5] proposed a subject-oriented design approach in which a design called a *subject* is created for each system requirement. A comprehensive design is a composition of subjects. Subjects are expressed as UML model views, and composition merges the views provided by the subjects. Subject-oriented modeling is an abstraction of the notion of subject-oriented programming developed by Ossher, Tarr, Harrison and others [15]. Specifying composition of subject-oriented designs or code is a key enabling feature of subject-oriented approaches. Ossher et al. describe several low-level and high-level composition rules for composing object-oriented programs (especially C++ programs). When used at the modeling level, the subject-oriented approach aligns well with MDA. The primary design subject is a PIM, and subjects describe middleware functionality. The PSM is obtained by composing middleware subjects with the primary design subjects.

3. An Overview of Subject-Oriented Modeling

Subject-oriented modeling has been described in [3, 4, 5]. Requirements criteria are used to decompose designs into subjects. Subjects are designed independently even though they may interact with or crosscut other subjects. A complete design is synthesized from several subjects using composition relationships. Each subject may consist of one or more UML diagrams. Because requirements are separated into different design models, both overlapping and crosscutting specifications can be supported. Multiple requirements often involve the same core concepts. A subject includes only those concepts that are involved in the design realization of the corresponding requirement. Crosscutting requirements are designed as separate subjects with composition capabilities that handle their integration with concepts in other subjects.

The subjects need to be understood together as one complete design. This is achieved by using composition relationships that allow the designer to (1) identify and specify overlaps between subjects, (2) specify how models should be integrated, and (3) specify how ensuing conflicts are reconciled. Since separate subjects may have overlapping concepts, these concepts need to be integrated into the same unit. Subjects may need different methods for integration depending on how they were modularized. For example, design elements may be merged if they are all needed in the integrated design. However, if a design element needs to be replaced by another, there needs to be an override strategy to replace the existing element with a new one. Conflicts that arise during integration must be resolved in different ways. For example, precedence relationships may be used to favor one design subject over another. Since crosscutting requirements tend to have behavior that affects multiple classes in different design subjects in a uniform way, the mechanism for such composition may be defined as a composition pattern. Pattern classes are those that are replaced during the composition. Non-pattern classes are those that are added to the composed subject.

The following *Logging* example is adapted from the *Tracing* example provided in [5]. Consider a system where all operations need to be logged. Logging is a crosscutting feature. We can design a separate *Logger* subject that will be composed with other design subjects to enable logging

of operation calls. Figures 1 and 2 describe the structure and behavior of the *Logger* subject. The UML template in Figure 1 defines the template parameters as classes and operations within the design subject that need to be replaced by classes and operations from other subjects at the time of composition. A template parameter `_loggedOp(..)` represents any operation that needs to be logged. The ".." indicates that operations with any signature can replace `_loggedOp(..)`.

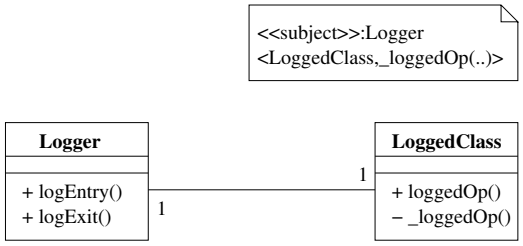

Figure 1. Logger subject class diagram.

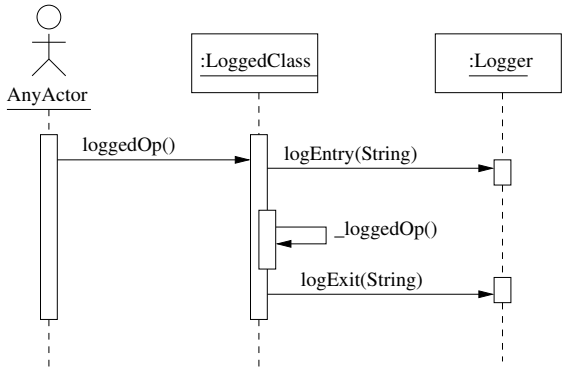

Figure 2. Logger subject sequence diagram.

Figure 3 shows the class diagram of the `CurrencyConversion` design subject that realizes some business functionality (`convert()`). Figure 4 shows the binding relationship between `CurrencyConversion` and the *Logger*. The `bind` attachment pattern specifies that all classes in `CurrencyConversion` and all operations in those classes will require the logging behavior.

As a result of composition, non-pattern classes from the *Logger* subject (e.g., `Logger`) are added to the composed output. The properties of the pattern class `LoggedClass` are added to each class that replaces it (e.g., `CurrencyConverter`. Sometimes a pair of operations is defined and referenced within the same pattern class (e.g., `loggedOp(..)` and `_loggedOp(..)`). One is a template operation (`_loggedOp(..)`). For each operation (e.g., `convert()`), we add a new operation

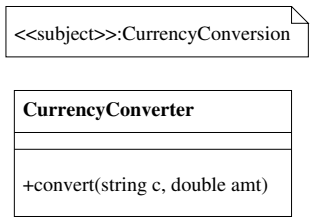

Figure 3. A CurrencyConversion subject.

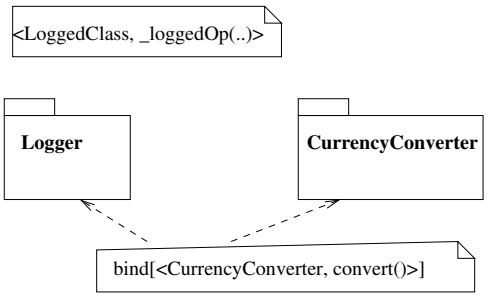

Figure 4. Bind specification for logging.

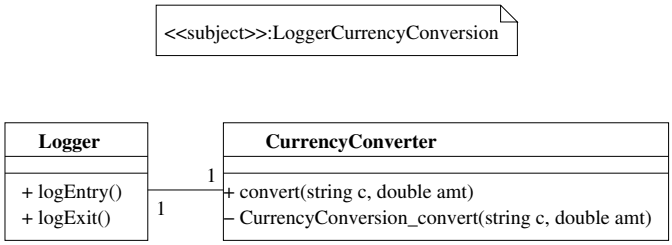

Figure 5. Composed class diagram.

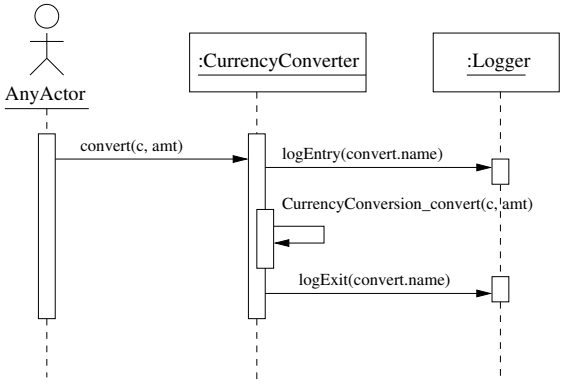

Figure 6. Composed sequence diagram.

(`CurrencyConversion_convert()`) that substitutes the template operation, and create a new interaction dia-

gram. Figures 5 and 6 show the composed class diagram and sequence diagram respectively.

4. Applying Subject-Oriented Modeling to Jini

Jini provides a distributed infrastructure for service-oriented applications. Services announce their availability on the network to enable clients to avail of the services. The Jini specification provides the basic features of lookup service discovery, service registration, service lookup, and leasing.

Jini-aware applications first need to discover at least one lookup service so that they can either register with the lookup, or query the lookup for other services. There are three discovery protocols: (1) unicast, (2) multicast request, and (3) multicast announcement. The unicast discovery protocol is used when an application is already aware of a specific lookup service and wants to talk directly to it. The multicast request protocol is used when an application needs to find out available lookup services. The multicast announcement protocol is used by lookup services to announce their presence. The application illustrated in this paper assumes a multicast request discovery protocol. Subjects that model each discovery mechanism can be developed separately.

Once a Jini service has discovered a lookup service, it registers its proxy with the lookup service. The service creates a service item and fills in a set of standard attributes (e.g., service name and location). The service item is passed to the lookup service's register operation. A client application queries lookup services for other available services that are registered. Searches are performed using the service name, interface type, and other attributes. In this paper, we use an application that relies on searches based on the interface type. Once a proxy is downloaded to the client side, the client can send requests to it. Proxies can either implement the entire business functionality inside themselves, or perform some part and transmit the rest to a remote server, or communicate the entire request to a remote server. In this paper we describe the third scenario.

Clients and services use up memory, CPU, and storage resources in the lookup services to store registration information. There may be loss of performance if the information is kept for a long time just because the lookup service was unaware that the client and service were no longer available. Jini uses leasing which allows resources to be granted to consumers for a fixed time period. Once a lease expires, the resource is reclaimed. Leases can also be renewed.

In this section we describe our design of subjects for each feature. These subjects were composed with a subject that specifies the core business functionality of a SmartHome (SH) application. The SH application consists of a smart refrigerator that contains food from several food groups. Figures 7 and 8 describe the primary subject of the SH application. When the `Refrigerator` wants to restore its food supply, it calls the `getFood()` operation inside `servObj`, am implementation of the `SuperMarketInterface`) and an instance of `SuperMarket`. The arguments specify the name of fooditem and quantity requested. Figure 8 also describes the operation when `getFood()` operation is called on `SuperMarket`.

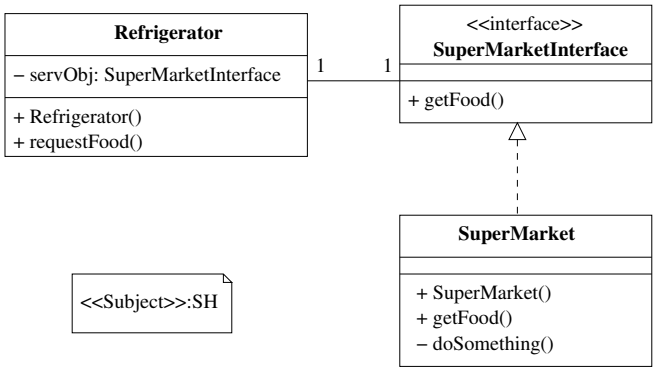

Figure 7. SH subject class diagram.

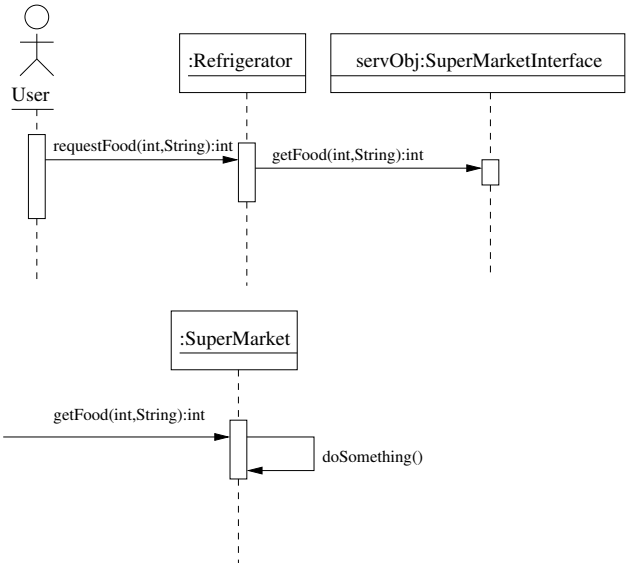

Figure 8. SH subject sequence diagrams.

4.1. Jini Design Subjects

We designed a Jini subject that performs lookup service discovery, service registration and service lookup. All three are related features, and we decided to include them in one subject even though not all may be used at the same time. For example, a service uses the discovery and registration

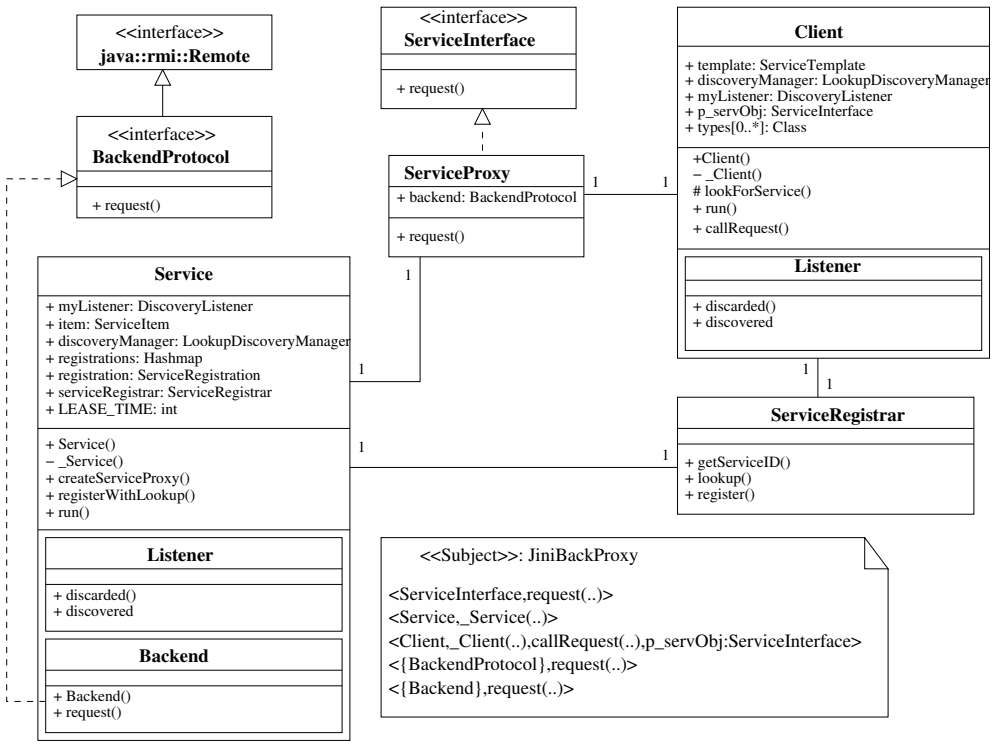

Figure 9. Jini Backend Proxy subject class diagram.

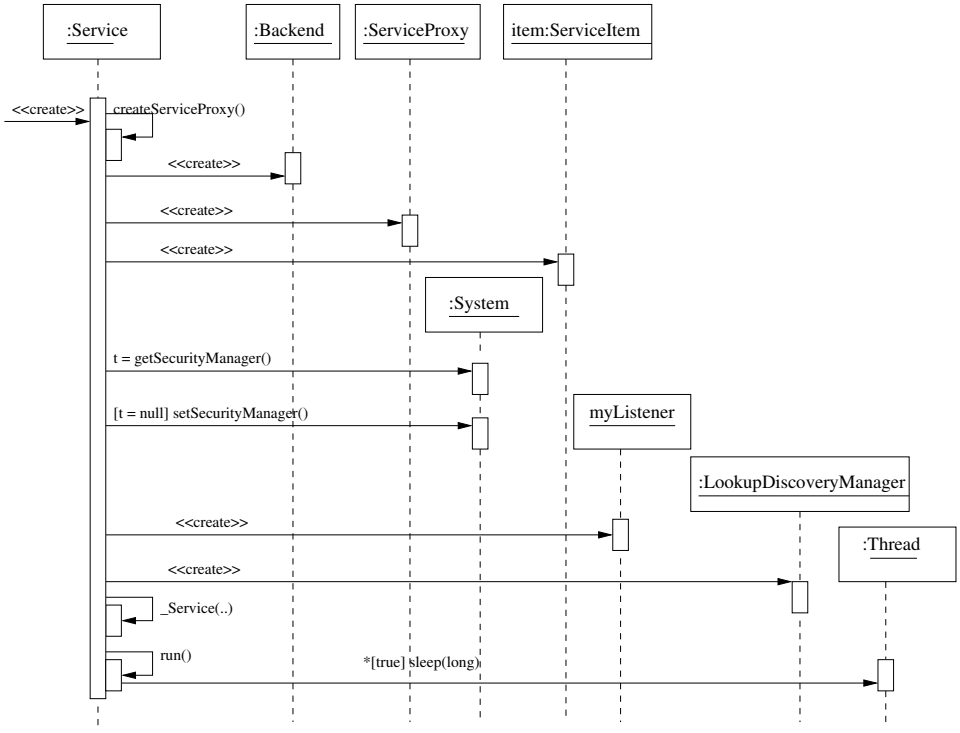

Figure 10. Jini Backend Proxy subject sequence diagram for `Service` and `ServiceProxy` creation.

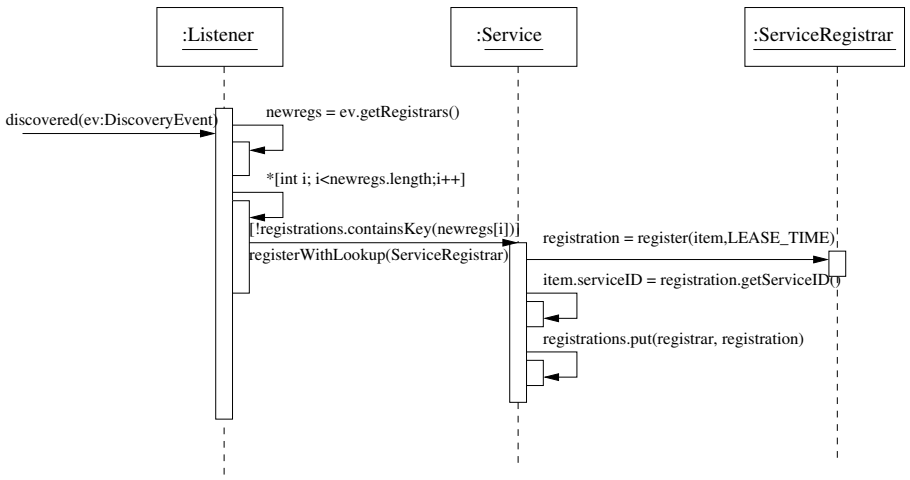

Figure 11. Jini Backend Proxy subject sequence diagram for discovering a lookup and registering the proxy with lookup service.

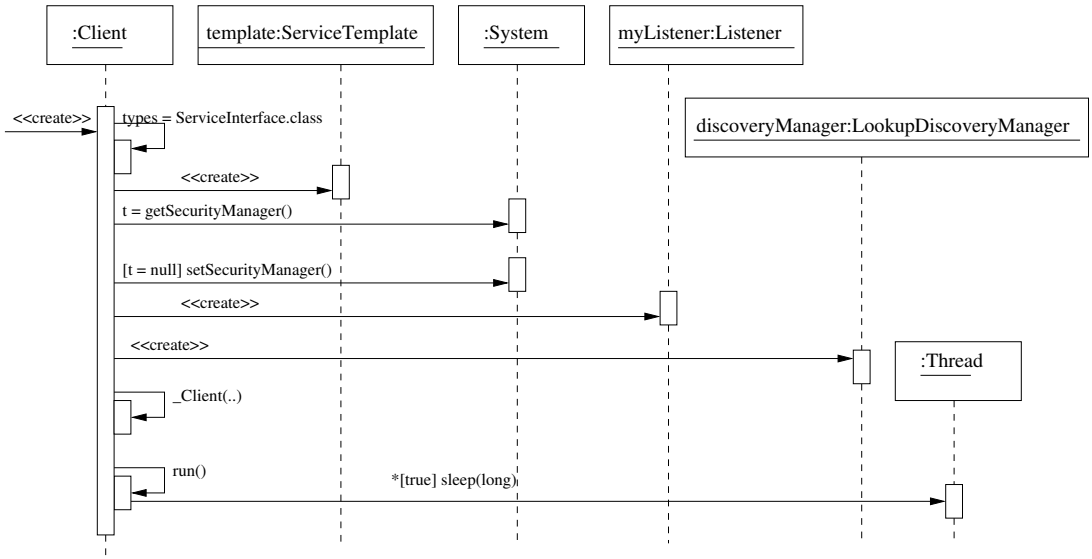

Figure 12. Jini Backend Proxy subject sequence diagram for creation of `Client`, `Listener`, and `LookupDiscoveryManager`.

features, and a client uses the discovery and lookup features. Figure 9 shows the class diagram of this subject.

A `DiscoveryListener` is needed on both the server and client sides to perform the initial discovery of a lookup service. We create this listener inside the constructors of the client and service classes. Java inner classes are used in our design. In the figure, we do not show details of the `Listener` class, such as the fact that it implements Jini's `DiscoveryListener` interface. Lookup services are instances of the `ServiceRegistrar`. The proxy obtained from the lookup service interacts with a `Backend` class that implements all the business functionality. The `Backend` class is a static inner class within `Service` and implements the `BackendProtocol` interface which has the same method `request(..)` as does `ServiceInterface`.

The template in Figure 9 specifies the template parameters that are replaced by design elements from SH sub-

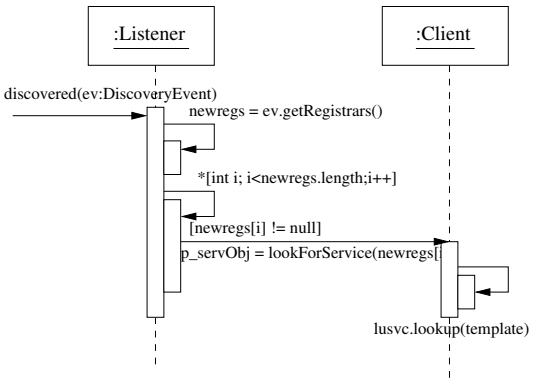

Figure 13. Jini Backend Proxy subject sequence diagram for binding of `p_servObj`, to the proxy implementing the `ServiceInterface`.

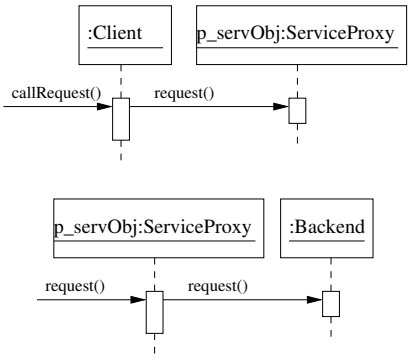

Figure 14. Jini Backend Proxy subject sequence diagram describing the interaction when the client requests the `p_servObj`.

jects during composition. Our definition of the template differs from Clarke's UML template [3, 5] which was used to specify a subject's pattern classes and the operations that needed to be replaced. We observed that the composition of subjects requires more than just composition of pattern classes and their operations. We modified the template to allow (1) the addition of pattern interfaces and their operations, (2) the replacement of attributes of pattern classes, and (3) the replacement of operations of non-pattern classes and interfaces. For example, during composition with the SH application, the `request(..)` method in `ServiceInterface` is replaced by a method from an interface of another subject. The attribute `p_servObj` of type `ServiceInterface` is replaced by an attribute of a class from another subject. A new non-pattern interface, `BackendProtocol`, is added to the SH design. New classes such as `Listener` and `Backend` need to be introduced inside the `SuperMarket` class. The `request()` operation inside `Backend` needs to be replaced by `getFood()`.

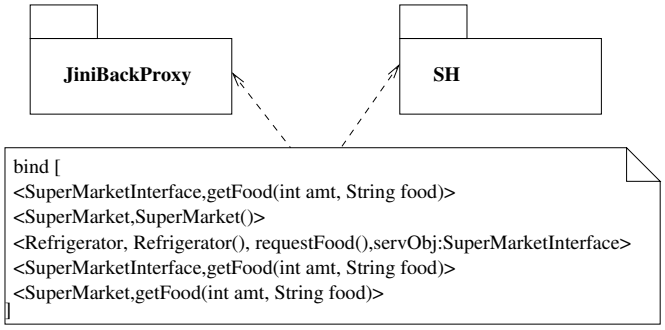

Figure 15. Binding specification between SH and Jini Backend Proxy subjects.

In the template, we use curly braces around the names of non-pattern interfaces and classes (`BackendProtocol` and `Backend`) to indicate that the names of their operations will get changed to the names of the operations specified in the binding specifications in Figure 15.

Figure 10 describes how the basic setup is performed to create a new backend process, proxy object, service item, discovery listener and other threads. When discovery events occur in the server side, the service registers its proxy service-item with the lookup service as shown in Figure 11. The setup process for the client side is shown in Figure 12. When discovery events occur at the client side, the client queries the lookup service for appropriate services (see Figure 13). Figure 14 shows the interaction between the client and backend server using the proxy, `p_servObj`, in between.

The binding specification shown in Figure 15 states the relationship between the template specifications of the Jini backend proxy subject and the elements in the SH subject that will replace the template parameters. Since we allow the specification of attributes, pattern interfaces and non-pattern interfaces or classes in the template, the binding specification needs to address the corresponding relationships as well. The bind specification states the following:

- The pattern interface, `ServiceInterface`, is replaced by `SuperMarketInterface`.
- The `request(..)` operation is replaced by the `getFood(int, String)` operation in `SuperMarketInterface`.
- The attribute `p_servObj` of type `ServiceInterface` is replaced by `servObj` of type `SuperMarketInterface`.

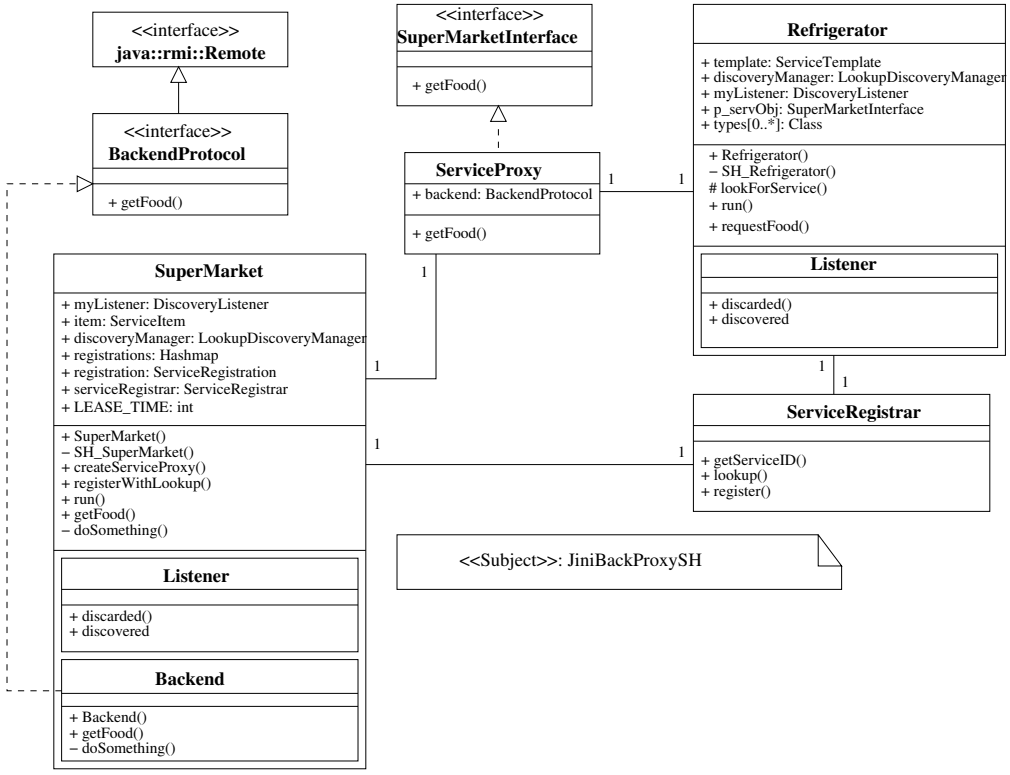

Figure 16. Composed class diagram.

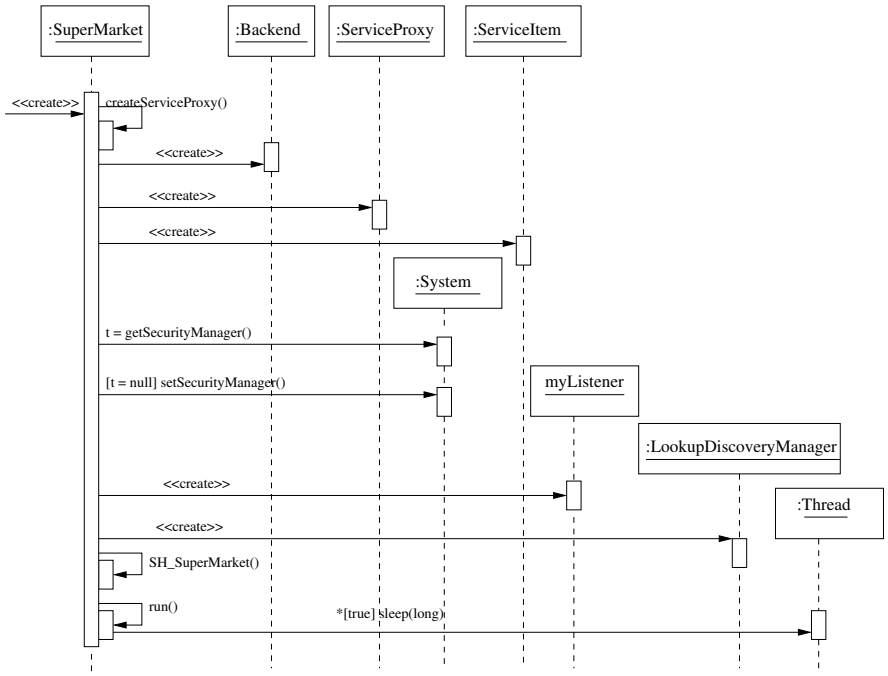

Figure 17. Composed sequence diagram from Figure 10.

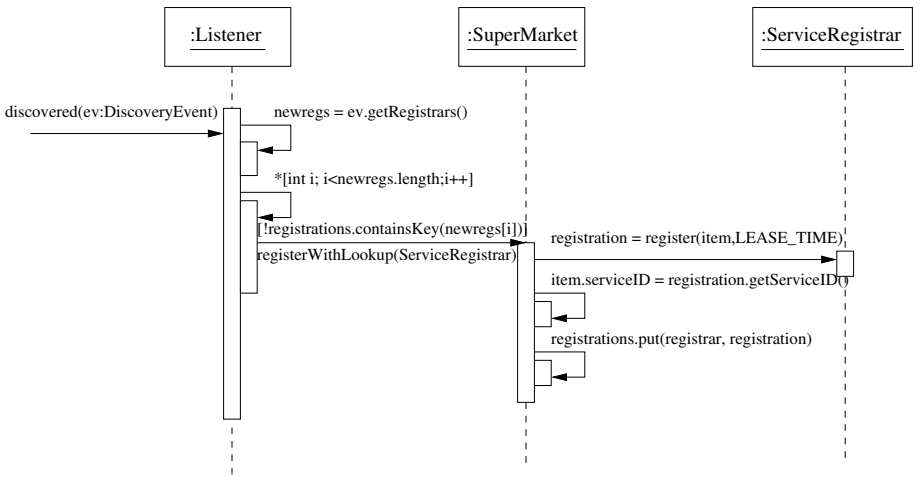

Figure 18. Composed sequence diagram from Figure 11.

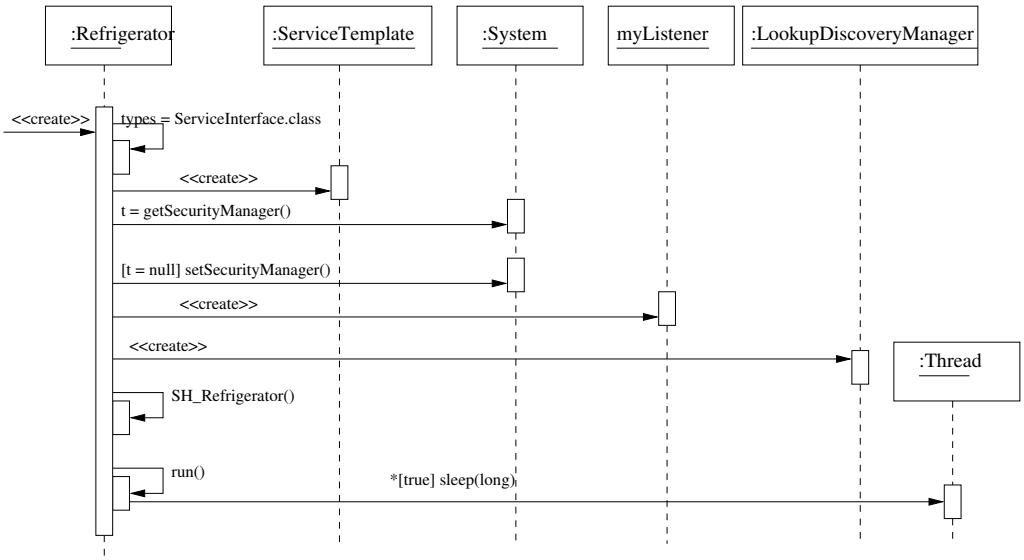

Figure 19. Composed sequence diagram from Figure 12.

- The `request(..)` operation in the non-pattern class `Backend` incorporates the behavior of `getFood(int, String)` in `SuperMarket`.

The `request()` operations in `Backend` and `BackendProtocol` get changed to `getFood()` (from `Supermarket` and `SuperMarketInterface`).

4.2. Subject Composition

The approach for composing class diagrams in design subjects is as follows:

1. Rename pattern interfaces using the application-specific interfaces specified in the binding specification. Retain all the non-template properties of the application-specific interfaces. Add the non-template properties of the pattern-specific interfaces to the application-specific interfaces.
2. Identify the operations in all the non-pattern interfaces (or classes) that extend (or implement) the pattern interfaces replaced in step 1. Rename them using names of the application-specific operations that replaced the pattern interface operations in step 1.
3. Replace the pattern classes with the application spe-

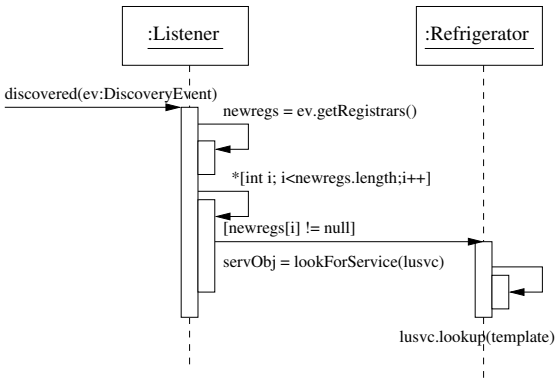

Figure 20. Composed sequence diagram from Figure 13.

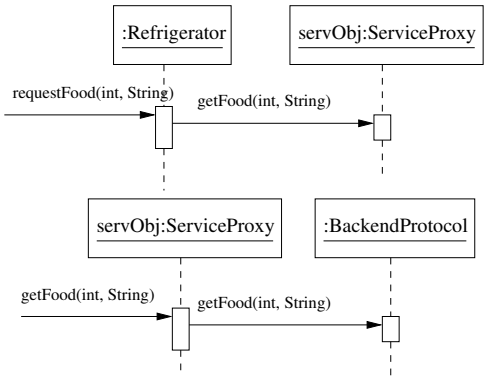

Figure 21. Composed sequence diagram from Figure 14.

cific classes named in the binding specification. Add all the non-template inner classes, operations, and attributes of the pattern classes to the application specific class.

4. Replace pattern attributes and their types with the corresponding application-specific attributes and their types. This information is available in the binding specification.
5. Replace the operations in the non-pattern interfaces with the operations specified in the binding specification. Rename operations accordingly in the non-pattern classes that implement non-pattern interfaces.
6. Operations in the non-pattern classes inherit the behavior of the corresponding template operations named in the binding specifications.

The approach for composing sequence diagrams is as follows:

1. Rename the template operations of the pattern classes with names of corresponding operations from the binding specification using Clarke's approach [3].
2. Introduce new sequence diagrams in the composed design subject as a result of step 6 in the class diagram composition approach.
3. Replace the template attributes in the introduced sequence diagrams with corresponding attributes in the binding specification.

We followed the steps listed below to compose the class diagrams. The resulting diagram is shown in Figure 16.

1. The `SuperMarketInterface` in the SH subject gets all the properties associated with `ServiceInterface`. The `request()` operation in `ServiceInterface` is bound to the `getFood()` operation in `SuperMarketInterface`. Therefore, all the `request()` operations in the non-pattern class `ServiceProxy` that implement the `ServiceInterface` are also replaced by `getFood()`.
2. We add all the non-template attributes, operations and inner classes in `Service` to the `SuperMarket` class. This class also gets associated with all the classes/interfaces that the `Service` class was associated with. All the template parameters are suitably replaced or renamed in `SuperMarket`. The composition of `Client` and `Refrigerator` follows the same approach. The composition of attribute `p_servObj` in `Client` with `servObj` in `Refrigerator` takes two steps. First, the type of `p_servObj` is changed to `SuperMarketInterface`. Next, following the binding specification, `p_servObj` is replaced by `servObj` in the `Refrigerator` class.
3. The `request()` operation in `BackendProtocol` is replaced by `getFood()` operation of `SuperMarketInterface`.

Figures 17, 18, 19, 20, and 21 show the result of composing sequence diagrams shown in Figures 10, 11, 12, 13, and 14 respectively. We used the following steps to compose the sequence diagrams:

1. The `p_servObj` attribute in `Client` is replaced by the `servObj` attribute in `Refrigerator`. The sequence diagram in Figure 13 for downloading a proxy is added to the composed subject with `p_servObj` replaced with `servObj`. The `Refrigerator's` interaction with `servObj` remains unchanged.
2. Since the `request()` operation in `Backend` is renamed `getFood(int, String)` and it incorporates the behavior of `getFood(int, String)` in `SuperMarket`, we need to create a new sequence diagram for this behavior (see Figure 22). All the operations that were part of the `getFood(..)` operation

in `SuperMarket` are copied to the `getFood(..)` operation in the `Backend`. The `getFood(..)` operation inside the `SuperMarket` class is retained in the composed subject.

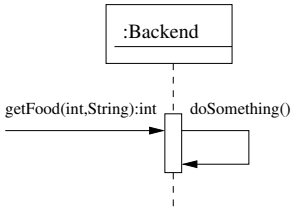

Figure 22. New sequence diagram for backend behavior.

3. The constructors `SuperMarket()` (and `Refrigerator()`) are composed with `Service()` (and `Client()`). The constructor of `SuperMarket` is replaced by a private method `SH_SuperMarket()`. We add another constructor `SuperMarket()` which executes the private `SH_SuperMarket()` method as shown in Figure 17 (composed from Figure 10). The `Refrigerator()` constructor is also composed in a similar manner (see Figure 19).
4. The sequence diagrams for discovery of lookup services are introduced in the composed design subject without requiring composition. The template parameters in these sequence diagrams just need to be replaced by the parameters specified in the bind attachment.

4.3. Design Subject for Leasing

Figure 23 shows the class diagram of the server side leasing subject. We assume that the service intending to use the leasing feature has already incorporated the Jini service registration features. The template parameter `p_registration:ServiceRegistration` will need to be replaced by an attribute from the actual service class. Figures 24 and 25 describe the creation and registration of the `LeaseRenewalManager`. The leasing subject was composed with the subject resulting from the composition of SH and Jini Backend proxy. Composed models are not shown owing to lack of space.

5. Conclusions and Future Work

We modeled Jini features separately as design subjects to reuse them in other Jini applications. We demonstrated the application of composition patterns to compose the Jini subjects with a smart home application. There were no conflicts during composition with Jini subjects. Therefore, we

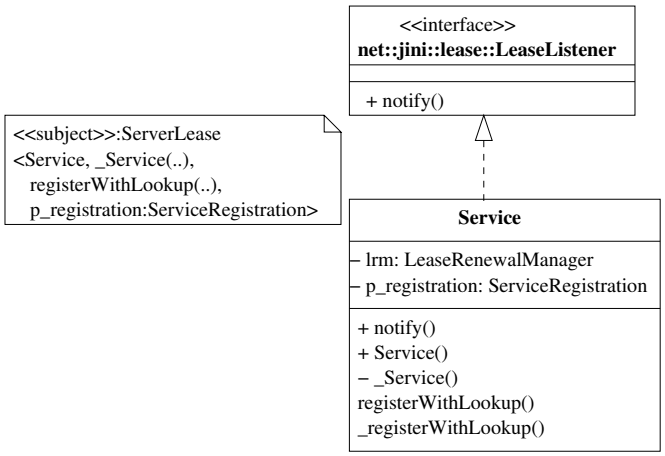

Figure 23. Leasing subject class diagram.

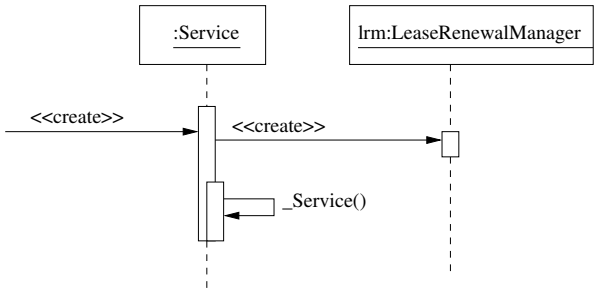

Figure 24. Leasing subject sequence diagram showing creation of `LeaseRenewalManager`.

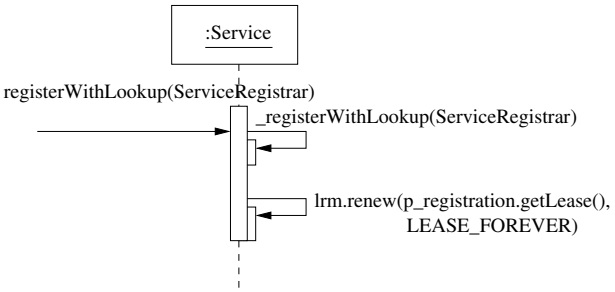

Figure 25. Leasing subject sequence diagram showing registration of `LeaseRenewalManager` with the `ServiceRegistrar`.

were able to achieve a clean separation of Jini features from the design of business functionality in the application.

The subject-oriented design approach can also be used to develop design subjects for different types of middleware

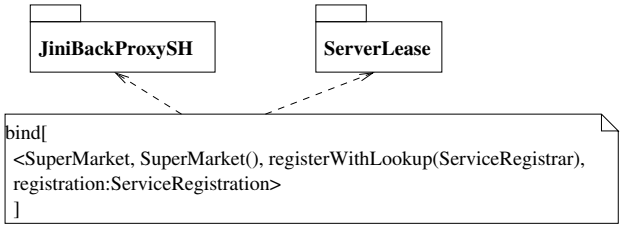

Figure 26. Binding specification for leasing subject.

technologies (e.g., Java RMI, CORBA). This will enable the reuse of primary design subjects when middleware platforms change.

We suggested some additions to the composition pattern notation. Our approach requires the composition of interfaces in addition to classes. We also require the composition of attributes. We suggested a notation for describing the composition of design elements in non-pattern classes and interfaces with design elements of classes/interfaces in primary design subject.

We are currently investigating techniques for mapping the Jini subjects to code aspects using AspectJ. This will enable the weaving of Jini code from libraries provided by third party implementations together with the code that implements core functionality. We will also evaluate the subject-oriented approach with other middleware platforms. This will help us understand the limitations of the subject-oriented approach and develop appropriate extensions. We are planning on carrying out empirical studies that assess the advantages resulting from the reuse of design subjects, and also the impact of this approach on the evolution of software designs in general.

References

[1] L. Bergmans and M. Aksit. Composing multiple concerns using composition filters. *Communications of the ACM*, 44(10), Oct 2001.

[2] L. Bussard. Towards a Pragmatic Composition Model of CORBA Services Based on AspectJ. In *Proceedings of ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*, Sophia Antipolis and Cannes, France, June 2000.

[3] S. Clarke. "Extending Standard UML with Model Composition Semantics". *Science of Computer Programming*, 44(1):71–100, July 2002.

[4] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Separating concerns throughout the development lifecycle. In *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, Lisbon, Portugal, June 1999.

[5] 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 '98)*, Kyoto, Japan, April 1998.

[6] W. Edwards. *Core Jini*. Prentice Hall, Inc. USA, 2001.

[7] R. B. France, I. Ray, G. Georg, and S. Ghosh. An aspect-oriented approach to design modeling. *To be published in IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2004.

[8] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, 44(10):87–93, Oct. 2002.

[9] F. Hunleth, R. Cytron, and C. Gill. Building Customizable Middleware Using Aspect Oriented Programming. In *OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, Florida, USA, October 2001.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct. 2001.

[11] 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 '01)*, pages 327–353, Budapest, Hungary, June 2001.

[12] 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 '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyvaskyla, Finland, June 1997.

[13] K. Kieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, Oct. 2001.

[14] Object Management Group. Model Driven Architecture. *URL* `http://www.omg.org/mda/`, 2004.

[15] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems, Wiley and Sons*, 2(3), 1996.

[16] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, Oct. 2001.

[17] J. A. D. Pace and M. R. Campo. Analyzing the role of aspects in software design. *Communications of the ACM*, 44(10):66–73, Oct. 2001.

[18] A. Rashid, A. Moreira, and J. Araujo. Modularization and Composition of Aspectual Requirements. In *2nd International Conference on Aspect-Oriented Software Development, ACM*, pages 11–20, Boston, March 2003.

[19] A. R. Silva. Separation and composition of overlapping and interacting concerns. In *OOPSLA '99 First Workshop on Multi-Dimensional separation of Concerns in Object-Oriented Systems*, Denver, Colorado, November 1999.

[20] G. T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, Oct. 2001.

[21] J. Suzuki and Y. Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. In *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, Lisbon, Portugal, June 1999.