Replace this file with prentcsmacro.sty for your meeting,
or with `entcsmacro.sty` for your meeting. Both can be
found at the ENTCS Macro Home Page.

Middleware Transparent Software Development and the MDA

Sudipto Ghosh, Robert B, France, Devon M. Simmonds [1]

Computer Science Department
Colorado State University
Fort Collins, CO USA

Abstract

An innovative *middleware transparent* approach for developing distributed applications is presented. The approach uses the aspect-oriented software development paradigm and separates application design from middleware specific concerns. Application elements specific to the middleware technologies are modeled separately from core business functionality using aspects and seamlessly integrated into the application later in the development process. Middleware transparency supports the MDA initiative and facilitates evolution of distributed applications with easy incorporation of new middleware technologies and enables reuse of high level application design and architectures that are independent of the middleware. This paper describes a technique to identify and localize middleware features as aspects. Middleware technologies such as Java RMI and Jini are used as examples.

Key words: aspect-oriented programming, distributed applications, middleware technologies.

---

1 Introduction

Due to the rapid growth of the Internet, distributed systems are becoming the norm for modern business applications. *Middleware* is a layer of software between these applications and the network, but it is transparent to end users. Middleware technologies, such as CORBA [24], COM [23], Jini [35], .Net [5] and SOAP [34], provide high-level programming abstractions to enable remote method invocations over underlying heterogeneous machines and networks. They also provide additional services such as naming, trading, and messaging, and quality of service features such as security and fault tolerance. However, the abundance of open and proprietary middleware technologies

[1] Email: `ghosh@cs.colostate.edu`

©2004 Published by Elsevier Science B. V.

and their rapid growth and evolution present significant challenges to software development organizations in the context of requirements engineering, design, implementation, testing and evolution. Developers must keep pace with changes in middleware technologies, and applications must also evolve. Application design and implementation in traditional software development are usually coupled with a specific middleware technology incorporated into the application. When the middleware is changed, entire applications must be redesigned and re-implemented to incorporate the changes.

This paper presents a *middleware transparent* approach using the aspect-oriented software development paradigm. This approach separates application development from the integration of middleware and shields application architects from the details of specific middleware. The high-level design architecture is independent of the middleware. Elements of the application that are specific to the middleware are modeled separately as aspects and seamlessly integrated (or woven) into the application later in the development process. Middleware transparency eases the evolution of distributed applications, supports easy incorporation of new middleware technologies, and enables reuse of high-level application designs and architectures that are independent of the middleware. The aspect-oriented paradigm supports separation of concerns in software development and makes it possible to modularize crosscutting concerns of a system. Since middleware features and services impact the design and architectural constraints as well as system implementation, it is natural to model them as aspects.

Incorporating the use of aspects early in the software development process has the potential to reduce development time for incorporating new middleware technologies. Aspect-oriented approaches raise the level of abstraction, insulating organizations from technology evolution and allowing them to evolve at the same pace as middleware technologies. The approach supports Model Driven Development (MDD). Current MDD initiatives that can benefit from this work are the Model-Integrated Computing (MIC) [2] and the Model Driven Architecture (MDA) of the Object Management Group [3] that is based on the Unified Modeling Language (UML). The MDA initiative targets the needs of the software industry faced with the challenges of business and technology change. The initiative attempts to decouple the design of the application from the target middleware.

In Section 2, we present background material on the MDA and aspect-oriented development, and summarize related work in the area of distributed application development. In Section 3 we give an overview of the proposed approach and its variations, and detail the activities related to the identification, specification and weaving of aspects. We illustrate the variation involving aspect-oriented programming in Section 4 with a case study in which an application was made Jini-compliant using our approach. We conclude and present

[2] See http://www.isis.vanderbilt.edu/research/research.html
[3] See http://www.omg.org/mda/

directions for future work in Section 5.

2 Background

Advances in internet and middleware technologies have spawned a new generation of software-intensive systems. As organizations seek to enhance services and gain competitive advantage, developers are under increasing pressure to develop quality systems that quickly utilize new technologies. The time-to-deliver pressure is often used as an excuse to adopt code-centric approaches to software development, but 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. Software evolution is problematic when developers must adapt software systems to rapidly evolving technologies in order to maintain or enhance an organization's effectiveness or competitive advantage.

The Model Driven Architecture (MDA) 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 development. The MDA approach uses a platform-independent model (PIM) as the basis for developing new applications. The focus is first on the functionality and behavior of the application, "undistorted by the idiosyncrasies of the technology or technologies in which it will be implemented" [4]. A complete MDA specification consists of a base UML PIM and mappings to specific middleware platforms. The mappings, ideally implemented in tools, automatically transform the PIMs to platform-specific models (PSMs). Thus, it is unnecessary to repeat the process of modeling an application's functionality and behavior each time a new middleware technology comes along.

Our approach requires developers to specify PIMs and then provide mappings to specific middleware platform models described by aspects. These aspects may be at the code level, the design model level, or both.

In this paper we show an example using code level Jini aspects. Aspect oriented programming (AOP) supports separation of concerns at the programming level [1,17,18,19,20,25,26,31,32]. An AOP aspect is an implementation or design concern that cross-cuts the primary functional units of a program. Researchers have started to address the problem of defining and weaving (or composing) aspects at an abstraction level higher than the programming language level [4,6,14,33]. In the aspect-oriented modeling approach proposed by Clarke et al. [3,4], 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. As part of the Early Aspects initiative, Moreira, Araujo, and Rashid targeted multi-dimensional separation throughout the software cycle

[4] See http://www.omg.org/mda/faq_mda.htm

[27,28,29,30].

Fiadeiro and Lopes [6] specify cross-cutting functionalities related to system coordination using an algebraic approach. Their approach is applicable to detailed design and code and utilizes a notation not widely known by system developers. Gray et al. [14] use aspects to represent the cross-cutting functionalities in domain-specific models. Their research is part of the MIC initiative that targets embedded software systems. MIC extends the scope and usage of models such that they form the backbone of a development process for building embedded software systems. Requirements, architecture, and the environment of a system are captured in the form of formal high-level models that allow the representation of concerns. The proposed work can complement the MIC efforts by providing UML-based techniques for representing and composing cross-cutting middleware functionalities. Suzuki et al. [33] extend the UML so that it can be used to model code level aspects but not necessarily design level aspects.

France, Ray, and Georg [7,13] used template UML diagrams to describe aspects representing security concerns (e.g., see [11,12,13]). This paper adapts and enhances the work to describe middleware specific aspects.

Recent projects examined the use of AOP to achieve middleware transparency. Bussard [2] encapsulated several CORBA services as aspects using AspectJ to make CORBA programming transparent to programmers. Hunleth, Cytron, and Gill [15] 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 would support several new types of AspectJ *introductions*: interface method and field, interface super class, structure field, oneway specifier, and IDL typedef and enumerations. We have not seen approaches that incorporate aspect oriented modeling to achieve middleware transparency.

3 Overview of Proposed Approach

Consider the scenario where an application developer needs to design a client and a remote service. The developer uses a middleware technology M_1 to implement the remote service discovery (finding and locating services) and connectivity (establishing a connection to the services). A few months later, a new technology M_2 arrives, and business decisions require that M_1 be replaced by M_2. A naive solution is to redesign and reimplement the entire application using M_2. However, at a high level, the design of the application is the same in both cases. It is in the specific details of the use of middleware, (e.g., how the client discovers the remote service and connects to it, and how the remote service advertizes its presence) that the detailed design and code may differ. One way to promote code reuse and automatic transformation (as much as possible) of the application from M_1 to M_2 is to isolate the middleware M_1 features as code aspects from the application and to use different code aspects

corresponding to M_2 to generate the new application code. Similarly, the high-level designs can be reused if design aspects can be developed for each type of middleware and composed with the reused high-level design.

3.1 Code Level Middleware Transparent Development

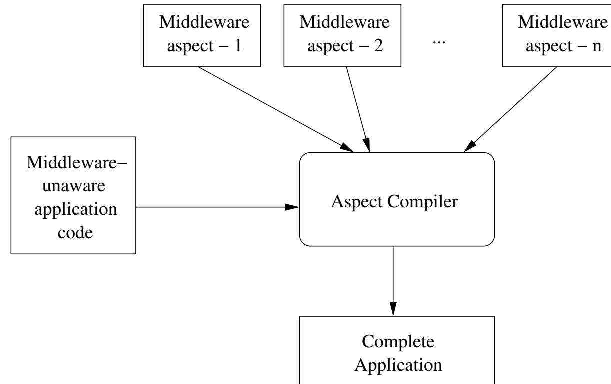


Fig. 1. Application of code level middleware aspects.

Middleware code crosscuts application classes and is scattered across multiple methods in the clients and services in a distributed application. In addition, client classes and services classes each have a specific set of middleware code. In peer-to-peer (P2P) applications, each peer has its own set of middleware code. It is natural to apply aspect-oriented programming to the development of software by encapsulating in aspects the code specific to clients, services and peers. The approach is illustrated in Figure 1. Each middleware feature is encapsulated in *Middleware Aspect-i*. The aspects are compiled with the middleware unaware application code that represents the basic functionality of the client/service/peer. The resulting application code is the complete application containing all the middleware features and services.

In the next section we describe how we successfully isolated aspects for Jini [35]. Jini requires the discovery of a lookup service, listening for the arrival of new services, lease management, and remotely invoking methods on a service. We developed aspects written in AspectJ to represent these features and an application was developed that was unaware of the features. The client in this application contained code that invoked methods on an object representing the service. There was no code that took care of discovering and connecting to the service. In the server code, we provided the implementation of the remote methods, but no lease management or proxy object creation code was provided. When the aspects were woven with the client and server code, the resulting code was a full-fledged Jini application containing all the features.

We repeated this exercise using Java RMI, XML/SOAP, and .Net. We wrote sets of aspects for each type of middleware and applied them to the same application. Concerns, such as connecting to services, discovering services, and

advertising services, were successfully implementable as code aspects. Other middleware features pertaining to transactions, group management, and quality of service concerns that were not explored earlier will be a focus of this project. We also made the following observations:

(i) Limitations in the evolving aspect languages prevent us from representing some of the constructs as aspects. AspectJ is still evolving and, at the time of the reported work, did not support inner classes required for lease management. Aspect/C# also had limitations. Some of the limitations will be removed in these languages, but the code-based approach to aspect-oriented software development will always be limited by the types of *join-points* that can be used for weaving in new code. These limitations are non-existent at the design model level because of the abstractions that can be provided. Although other constraints may apply, the higher level abstractions in design models can still describe a larger set of cross-cutting concerns than those at the code level.

(ii) Aspects written to represent specific middleware concerns are highly coupled to the application being developed. For example, a connectivity aspect written for a distributed currency converter application would have to be significantly changed to work with a distributed supermarket management application even though both are Jini applications. The coupling arises out of a need for the aspect to know the names and attributes of several application classes. Thus, although customized aspects can be written for every new application, reuse of the aspects in different Jini applications is limited because of our inability to describe generic aspects that can be instantiated for (or mapped to) different Jini applications.

These observations led us to raise the level of abstraction and apply the approach at the design model level. The proposed approach requires the development of a platform independent model (PIM). PIM developers create an application design or implementation that is independent of middleware considerations. In the previous code example, a PIM would correspond to the model derived from application code that is independent of middleware. However, the PIM does not have to be an implementation model; it may also be a design model.

The approach also requires a mapping that takes a PIM and aspects corresponding to a certain middleware and transforms the PIM to a platform specific model (PSM). Middleware experts isolate and model the middleware specific concerns as aspects. Tools are used to weave the aspects with the application to produce the complete distributed application. Aspects in the previous code example are written in AspectJ (or Aspect/C#), and the AspectJ (or Aspect/C#) compiler was used to weave the aspects. However, using our approach, the aspects may also be design level aspects.

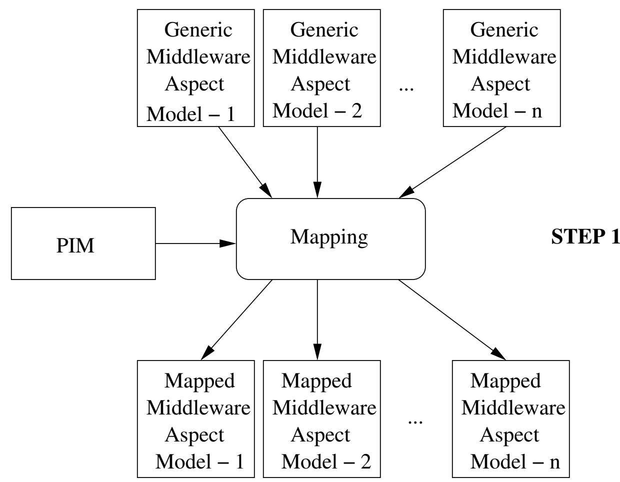


Fig. 2. Generic and mapped middleware aspects.

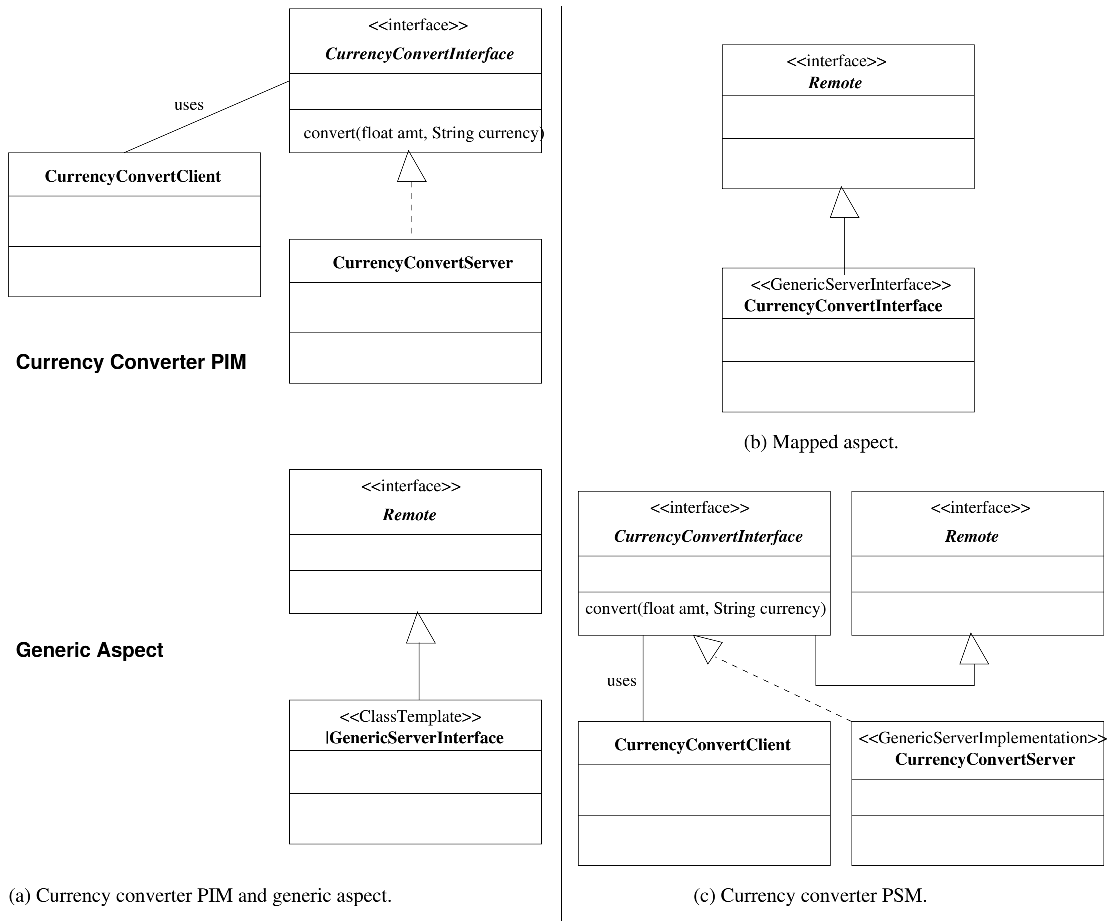
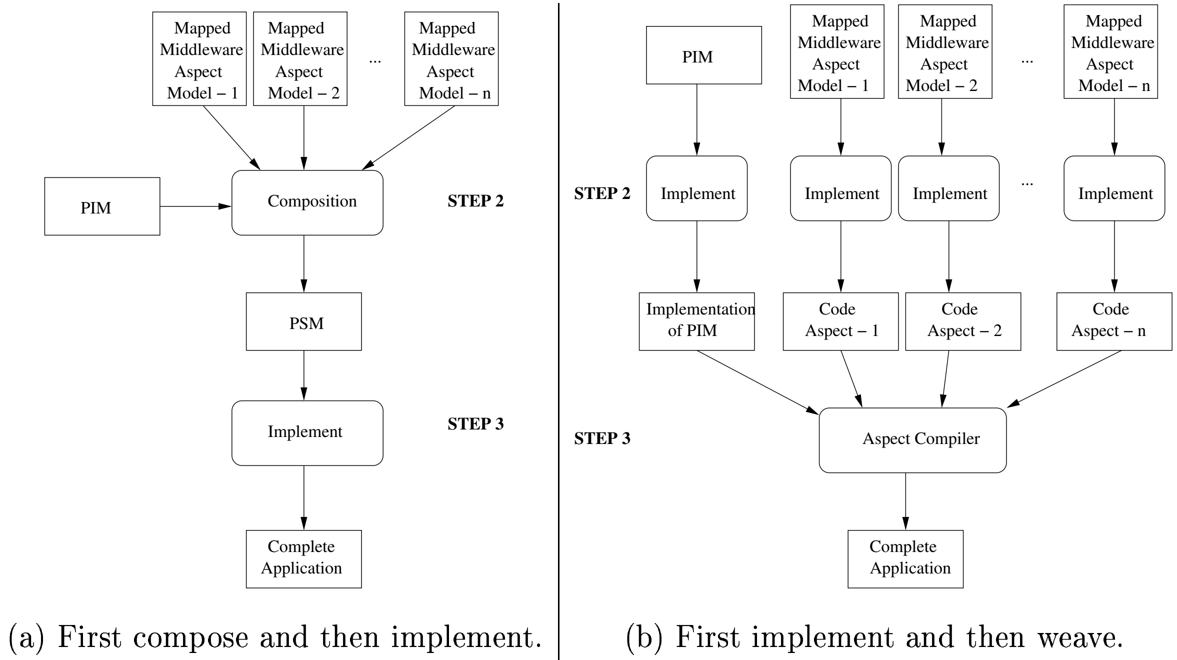


Fig. 3. Illustration of currency converter application.

3.2 Design Level Middleware Transparent Development

The design approach is illustrated in Figures 2 and 4. In Figure 2, the PIM describes the core design elements in terms of UML models (e.g., class and



(a) First compose and then implement. (b) First implement and then weave.

Fig. 4. Application of design level middleware aspects.

sequence diagrams). A *Generic Middleware Aspect Model* describes the UML model of a middleware feature specific to a type of middleware technology. This model is generic in that it can be applied to any application using that particular middleware. Constructs in the generic aspect model need to be mapped to the constructs in the application to get the *Mapped Middleware Aspect Model*. This process needs human input.

Figure 3 illustrates the mapping and composition for the *CurrencyConverter* application. The example in this figure uses the notation developed in [9,10,21,22]. This notation uses parameterized UML artifacts referred to as templates. Consider a simple application (see Figure 3(a)) that contains a Java server class called *CurrencyConvertServer* that implements an interface called *CurrencyConvertInterface*. This interface has a method called `float convert(float amt, String currency)`. The Java client class is called *CurrencyConvertClient*, and it invokes methods on the server. To convert this setup to a distributed Java RMI-based application, the *CurrencyConvertInterface* should extend the *Remote* interface. The generic aspect model for Java RMI would specify a generic server interface that extends the special Java interface *Remote*. As part of the mapping process, we need to specify that the generic server interface be mapped to our *CurrencyConvertInterface* (see Figure 3(b)).

Figure 4 describes how the mapped aspects can be used in two different ways. The mapped aspects can be composed together with the PIM to produce the PSM as shown in Figure 4(a). This PSM can be implemented by software developers with the help of code generators. In the example (see Figure 3(c)), the resulting class model from the composition would contain the

CurrencyConvertServer and *CurrencyConvertClient* classes, and the *CurrencyConvertInterface* and *Remote* interfaces with the appropriate associations. Composition of aspect models is a complex process and will be investigated in this research.

An alternative approach is presented in Figure 4(b), where the mapped aspects are first implemented as code aspects. Finally, these aspects are woven together with the implementation of the PIM. In the example, this would involve writing an AspectJ aspect that would effectively declare the interface *CurrencyConvertInterface* with the `extends Remote` clause using the following AspectJ notation:

```
declare parents: CurrencyConverterInterface extends Remote;
```

A design aspect model cannot always be converted to a code aspect because current code aspect languages may not have the constructs to capture and refine the rich abstractions at the design level.

Using the design approach involves the following tasks:

(i) Identify types of middleware functionalities and services to be represented as design or code aspects.
(ii) Specify design level middleware aspects in the UML.
(iii) Compose aspects with design model and then implement.

3.2.1 Identify Aspects Representing Middleware Functionalities and Services

In the early stages of design, software designers specify the qualitative properties required from the service, e.g., fault tolerance, availability, and security. These services are ultimately realized in some specific object or component-middleware technology. Developers specify PIMs that are independent of the middleware technology. However, these PIMs may need to be refined and details specific to a middleware technology may need to be added. The refinement may be done at the design level or, in some cases, at the code level because of idiosyncrasies in the middleware technology.

3.2.2 Specify Design Level Aspects

Generic aspect models are described using the UML-based notation. Standard UML notation will be used to depict the context-specific (mapped) aspects. The generic aspect models are reusable models that specify patterns of structural and behavioral UML models of mapped aspects. In our earlier work [8,9,10,21,22], we described a new UML-based pattern specification notation called *Role Models*. These pattern specifications define constrained forms of the UML metamodel. The UML metamodel specifies valid forms of UML models and consists of a UML class diagram, well-formedness rules expressed in the Object Constraint Language (OCL) [36], natural language, and informal descriptions of semantics. Adding constraints to the UML metamodel results in a specification of a subset of valid UML models. In this case, the generic aspect models describe valid realizations of mapped aspects.

Descriptions contain a *static structural diagram* to show the structural features of the generic aspect and an *interaction diagram* to show the behavioral features. The two diagrams can be used to stamp out valid mapped aspect models. Constructs in the mapped aspect model play various *roles* that are specified in the generic aspect model.

3.2.3 Develop a Composition Technique

Assuming that the software developer has a core decomposition of the design model in the form of a primary model (PIM), the generic aspect first needs to be mapped to the primary model, thereby producing the mapped aspect model (see Figure 2). The next step is to merge the mapped aspect model with the primary model (see Figure 4(a)). If there are corresponding elements in the primary model and aspect model with the same syntactic type, they can be merged. Merging may necessitate addition or removal of elements from the primary model. Preconditions and postconditions are appended to matching operations in the composed model. Constraints are appended to matching attributes in the composed model. For matching associations, the stronger multiplicities are used in the composed model.

4 Jini Case Study

We used the developmental framework shown in Figure 5. The framework consists of the following:

(i) *PIM*: The platform independent model of the application.
(ii) *Aspects* — MFA and QoSA: A collection of aspects that capture the middleware requirements (services, facilities, and QoS) for an application, for example leasing, transactions, and security. MFA means middleware functional aspects, and QoSA means quality of service aspects. MFA includes all non-QoSA aspects.
(iii) *Converters* - Standardized mappings that transform PIMs and middleware aspects to their enhanced versions. Separate converters are required for each middleware. Converters perform architectural, design or code transformations to prepare a generic model (PIM, MFA, and QoSA) for a specific context.
(iv) *Enhanced Models* - EPIM, EMFA, EQoSA: Independent models (PIM, MFA, and QoSA) are developed without regard for a target middleware or a target application, and have to be transformed by a converter before aspect weaving is possible.
(v) *Aspect Weaver*: The weaver produces the final platform specific model(PSM) by combining the enhanced aspects (EMFA and EQoSA) and the EPIM.
(vi) *PSM*: The final and complete model output by the framework with all the required middleware elements. In this case, the PSM describes the code model.

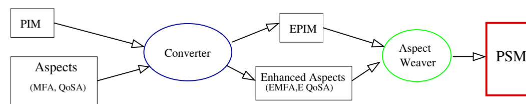


Fig. 5. Developmental Framework

Figure 5 shows that there are two main inputs to the framework - a PIM and some aspects (MFA and QoSA). The figure also shows that there are enhanced models for each PIM and each aspect. Each PIM has a particular architectural signature (structure) consisting of a collection of classes with specific relationships, a collection of interfaces used by the classes, and a collection of methods particular to each class and interface. Although all PIMs are middleware independent, one PIM may differ significantly from another PIM in its architectural signature. This difference and the fact that middleware aspects require a PIM with a specific architecture in order for aspect weaving to be successful, require the use of an Enhanced Platform Independent Model (EPIM). An EPIM is simply a PIM that has been refactored to produce the appropriate architectural signature required for a specific middleware aspect. The particular refactoring algorithm applied depends on the selected aspect (e.g., transaction) and the selected middleware (e.g., Jini). For a given middleware, different aspects will require different architectural signatures and for a specific aspect (e.g. transaction) the required architectural signature will differ from middleware to middleware. In our framework the refactoring algorithm is encapsulated in a *Converter*.

There are significant benefits to this separation. These refactoring transformations cannot be specified in the PIM as that would make it platform specific. They cannot be specified in the weaver (e.g., AspectJ) as this would make the weaver application specific which is completely unacceptable. Specifying the transformation in a separate component is therefore the only sure way to be MDA compliant and provide the flexibility and structural coherence needed for the framework. A similar argument can be provided for the need of enhanced aspects (EMFA, EQoSA). A generic middleware aspect (MFA, QoSA) is intended to be used for any relevant application. At the time of its creation it is unnecessary to determine the specific application to which a generic aspect will be applied. For practical purposes, this information would be unavailable for numerous situations. Generic middleware aspects are, therefore, application independent. The application independent nature of generic aspects coupled with the fact that each application has its own architectural signature, necessitates that generic aspects are transformed before usage to make them application specific.

We used the following three-stage design process to create the application. The activities of stage 1 can be done concurrently as can the activities of stage 2.

Stage 1 Tasks:

(a) Create the PIM for the server/client - no middleware consideration

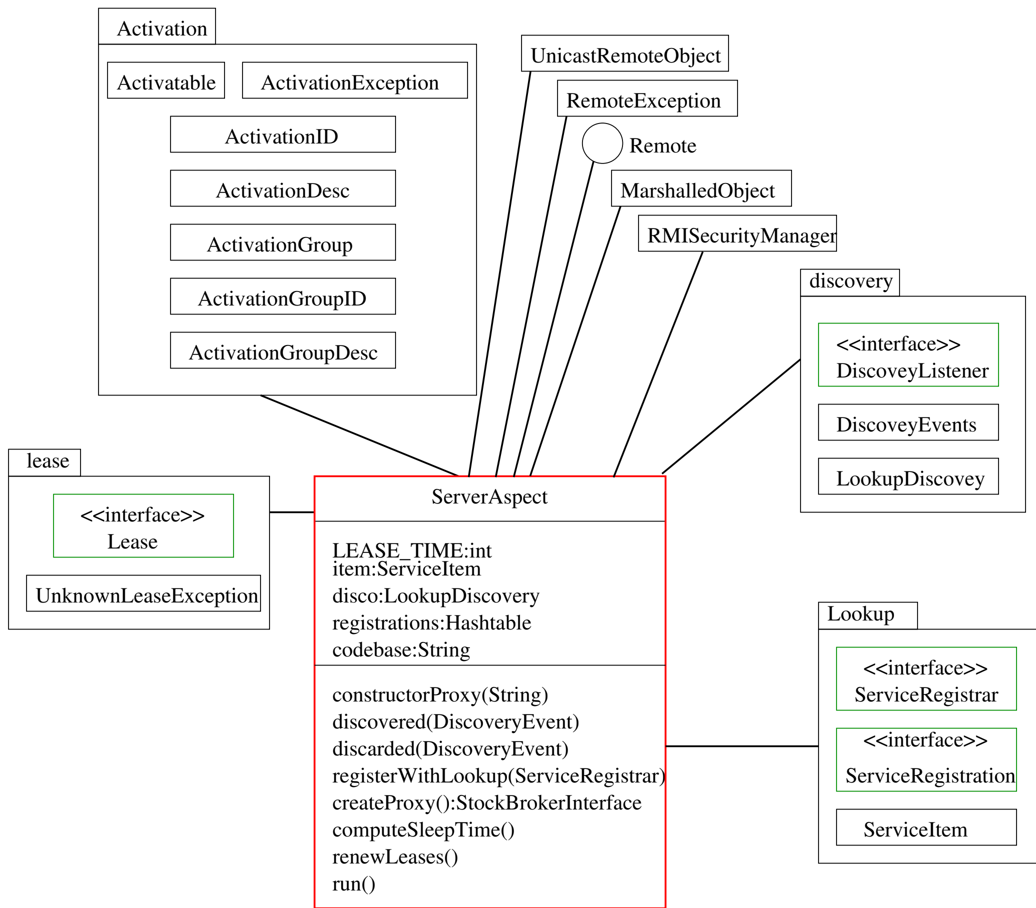


Fig. 6. Jini StockBroker MFA

necessary.

(b) Select the target middleware and create the generic middleware aspects (MFA, QoSA).

Stage 2 Tasks:

(a) Transform the PIM to EPIM using the application converter.

(b) Transform the generic aspects to enhanced aspects (EMFA, EQoSA) using the aspect converter.

Stage 3 Tasks:

Weave the enhanced aspects into the EPIM to produce the PSM. A separate PSM is produced for each client and for each server.

For our case study, we selected a simple distributed stock broker application. We did not consider quality of service aspects. Our only concern was to provide connectivity and develop a Jini-compliant application. The functional requirements of this application included providing the ability to clients to register with the stock broker service, and buy and sell stocks. We wrote the application in Java and used AspectJ as the aspect language.

We developed the Jini converters and MFA as a one-time exercise. We also developed the PIM classes and interfaces for the application. All other

steps in the process are completely automated. Figures 6, 7, 8, and 9 give a graphical view of the MFA, PIM, EPIM and the application PSM respectively. The specific activities that produced these models during the design phase are described below:

(i) *Create the PIM and MFA:* The PIM shown in Figure 7 was developed and tested as a stand alone Java application. It consists of a single interface and three classes. The MFAs are shown in Figure 6.

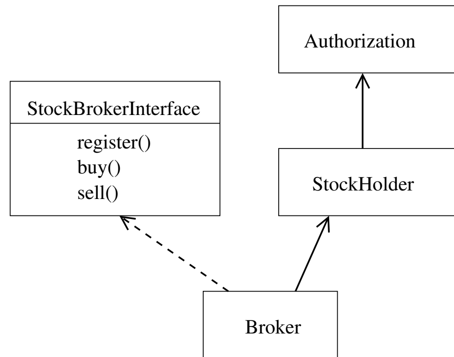


Fig. 7. Jini StockBroker PIM

(ii) *Generate the Enhanced Models (EPIM, EMFA)*

A number of Jini design models are possible. The one we chose to use required that the services to be made available by the server be captured in a inner class and that a proxy for this inner class be created as well. The aspect converter implemented a simple string matching algorithm that replaces generic class names and class attributes in the aspects with the actual class names and attributes from the Stock Broker application. The application converter was written using the Java Tree Builder [16]. It generated the EPIM (see Figure 8) by refactoring the PIM to include seven new components all of which are required by Jini. The tasks performed by the application converter are as follows:

(a) Create a remote interface to be used by the server
(b) Create two inner classes: a server (from the PIM) and a proxy for the server.
(c) Create a wrapper class as the outer class for the two classes just mentioned.
(d) Insert import statements from the interface into the wrapper class.
(e) Add throws clauses and/or exceptions statements specific to Jini to the client and server code.

(iii) *PSM Generation:* The AspectJ compiler was used as the weaver. This is both an asset and a challenge: AspectJ is Java compatible but our directives are limited to those of AspectJ.

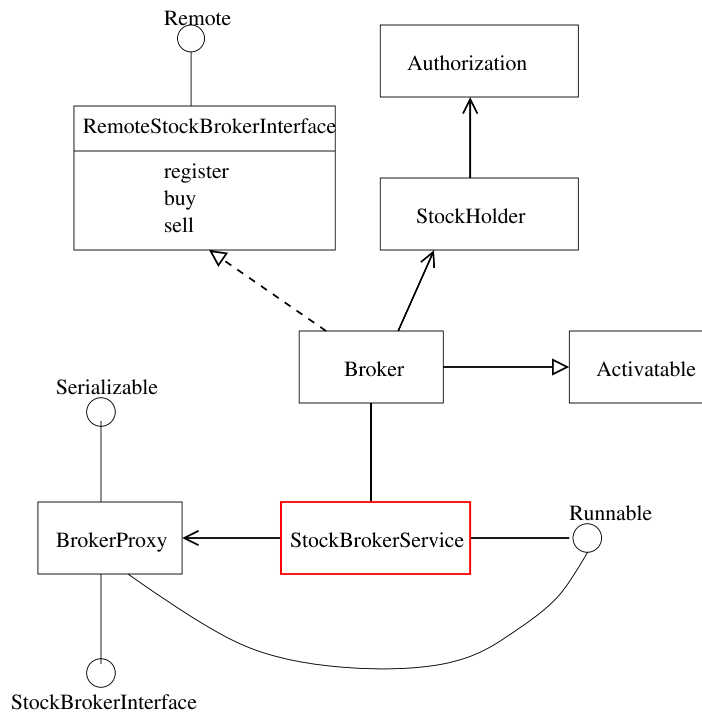


Fig. 8. Jini StockBroker EPIM

5 Conclusions and Future Work

We presented a middleware transparent approach incorporating aspect-oriented software development techniques to decouple the design of an application from the specifics of the middleware technologies. This approach evolved from our current research and is based on the MDA vision. The approach will provide many benefits such as reduced design complexity and development time when incorporating new middleware technologies, maintainability, and adaptable software evolution.

We are investigating several middleware technologies to identify what features can be easily isolated as aspects. We are also evaluating which aspects are best specified and composed at the design level and which ones at the code level. Specifying aspects at the design level removes the limitations of aspects at the code level, raises the level of abstraction, and permits early design decisions. We are developing composition techniques that can be used for aspect-oriented design models. We are developing static and dynamic analysis techniques to validate the designs and code resulting from the weaving of middleware aspects with the rest of the application. We also plan to evaluate the impact of using an aspect-oriented approach on the maintenance of middleware-based applications.

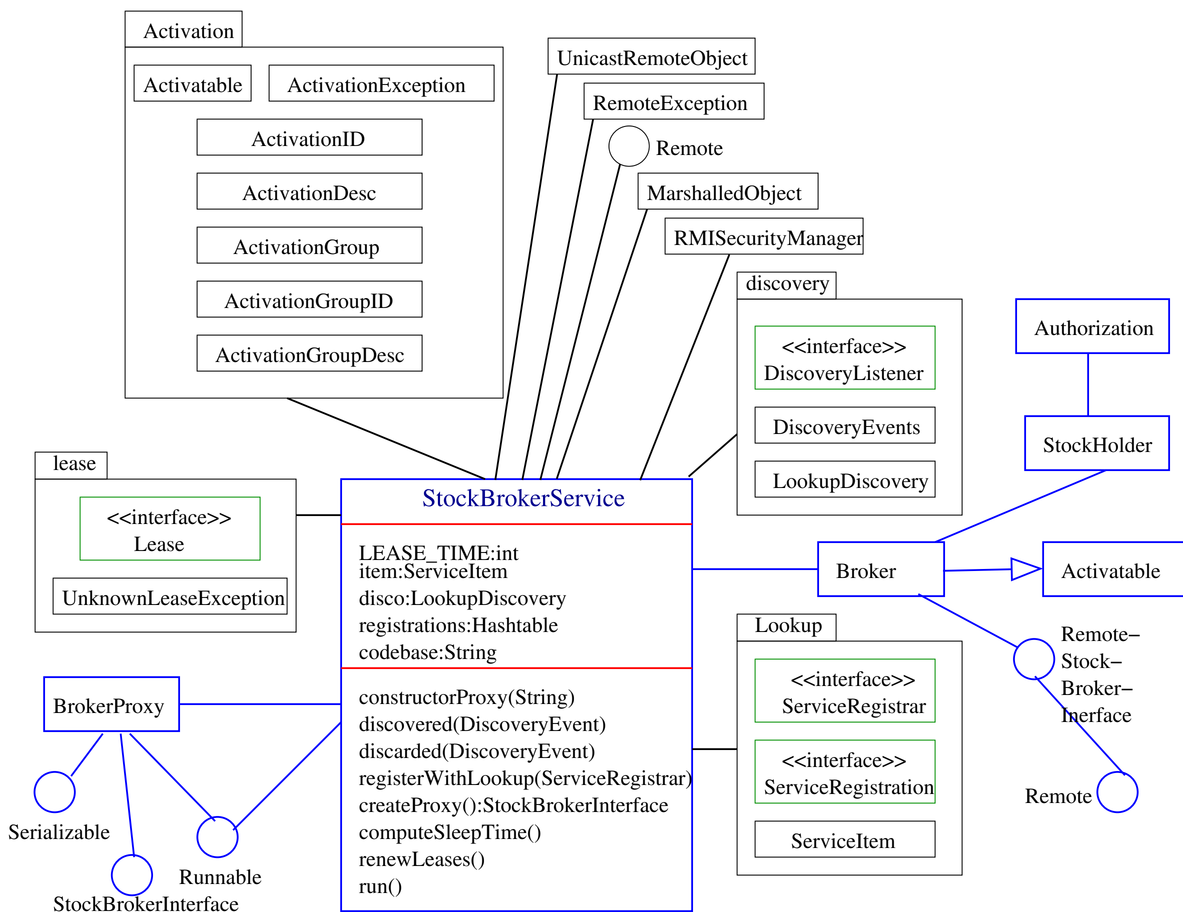


Fig. 9. Jini StockBroker PSM

6 Bibliographical references

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, 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.

[4] 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.

[5] M. Corporation. .Net. *URL* `http://www.microsoft.com/net/`, 2003.

[6] J. L. Fiadeiro and A. Lopes. Algebraic semantics of co-ordination or what is it in a signature? In A. Haeberer, editor, *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 293–307, Amazonia, Brasil, January 1999. Springer-Verlag.

[7] R. France and G. Georg. Modeling fault tolerant concerns using aspects. Technical Report 02-102, Computer Science Department, Colorado State University, 2002.

[8] R. B. France, S. Ghosh, E. Song, and D. K. Kim. A Metamodeling Approach to Pattern-Based Model Refactoring. *IEEE Software Special Issue on Model-Driven Development*, 20(5):52–58, September 2003.

[9] R. B. France, D. K. Kim, E. Song, and S. Ghosh. Using Roles to Characterize Model Families. In *Proceedings Tenth OOPSLA Workshop on Behavioral Semantics: Back to the Basics*, Portland, Oregon, October 2001.

[10] R. B. France, I. Ray, G. Georg, and S. Ghosh. "An Aspect-Oriented Approach to Design Modeling". *Submitted to IEE Proceedings — Software, Special Issue on Early Aspects, Aspect-Oriented Requirements Engineering and Architectural Design.*

[11] G. Georg, R. France, and I. Ray. An Aspect-Based Approach to Modeling Security Concerns. In *Proceedings of the Workshop on Critical Systems Development with UML*, Dresden, Germany, 2002.

[12] G. Georg, R. France, and I. Ray. Designing High Integrity Systems using Aspects. In *Proceedings of the Fifth IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS 2002)*, Bonn, Germany, November 2002.

[13] G. Georg, I. Ray, and R. France. Using Aspects to Design a Secure System. In *Proceedings of the Interational Conference on Engineering Complex Computing Systems (ICECCS 2002)*, Greenbelt, MD, December 2002. ACM Press.

[14] 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.

[15] 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.

[16] Jens Palsberg. JTB: Java Tree Builder. *URL* `http://www.cs.purdue.edu/jtb/`.

[17] 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.

[18] 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.

[19] 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.

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

[21] D. K. Kim, R. B. France, S. Ghosh, and E. Song. Using Role-Based Modeling Language (RBML) as Precise Characterizations of Model Families. In *Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, Greenbelt, MD, December 2002.

[22] D. K. Kim, R. B. France, S. Ghosh, and E. Song. A Role-Based Metamodeling Approach to Specifying Design Patterns. In *COMPSAC 2003 (to appear)*, Dallas, TX, November 2003.

[23] Microsoft Inc. Component Object Model (COM) Website. *URL* `http://www.microsoft.com/com/`.

[24] OMG — The Object Management Group. *Common Object Request Broker Architecture CORBA/IIOP 2.6.* OMG, 2002.

[25] 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.

[26] 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.

[27] A. Rashid. A Hybrid Approach to Separation of Concerns: The Story of SADES. In *3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), Springer-Verlag Lecture Notes in Computer Science 2192*, pages 231–249, Kyoto, Japan, September 25–28 2001.

[28] A. Rashid and R. Chitchyan. Persistence as an Aspect. In *2nd International Conference on Aspect-Oriented Software Development, ACM*, pages 120–129, Boston, March 2003.

[29] 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.

[30] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo. Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In *IEEE Joint International Conference on Requirements Engineering, IEEE Computer Society Press*, pages 199–202, Essen, Germany, September 9–13 2002.

[31] 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.

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

[33] 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.

[34] W3C. Simple Object Access Protocol (SOAP). *URL* `http://www.w3.org/TR/SOAP/`, 2003.

[35] J. Waldo. "Alive and Well: Jini Technology Today". *IEEE Computer*, 33(6):107–109, June 2000.

[36] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.