

Exploiting Roles and Responsibilities to Generate Code in a Distributed Design-Pattern-Based Programming System

Jun Chen and Steve MacDonald

David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, CANADA
{stevem, j2chen}@uwaterloo.ca

Abstract

The implementation of a design pattern can be viewed as a process of selecting classes to play the roles needed by the pattern. To ensure that a class can play a role, each role has a set of responsibilities associated with it. When all responsibilities are satisfied, the pattern implementation is complete. Normally a programmer must write code for the responsibilities. However, the implementation of responsibilities can often be derived from information in the application or from user input, and then automatically generated. This paper describes a new approach that uses roles and responsibilities to generate code for an almost-complete application architecture in a pattern-based framework, leaving only application-specific code for the developer. We demonstrate this approach using RMA, an Eclipse plugin for writing Java 2 Enterprise Edition (J2EE) applications based on an existing J2EE framework.

Keywords: Design patterns, Eclipse, Java 2 Enterprise Edition, Frameworks, Code generation

1 Introduction

The implementation of a design pattern can be viewed as one of assigning classes to the different roles that are needed. These roles correspond to the participant classes that are part of a design pattern description [9]. The assigned classes may already exist or may be created for the role. Each role also has a set of responsibilities that must be fulfilled by the assigned class. In the most general sense, a responsibility is a property that the class must exhibit to play a given role. A responsibility can be as simple as having to extend a particular superclass, or as complex as implementing a method in a particular manner.

Since design patterns are design constructs, it normally falls to the programmer to implement the responsibilities in the classes that make up the application.

However, most design pattern implementations are a combination of application-specific code and infrastructural code, where the latter includes class and object interactions that support the intent of the pattern. In many cases, this infrastructural code can be automatically generated based on information gathered from our knowledge of the pattern and its intent, from existing application code, and from simple user input.

This paper explores the use of roles and responsibilities in a design-pattern-based programming tool. We show that it is possible to generate responsibility code using application and user information. Further, we show that it is possible to exploit existing frameworks or pattern languages to create an application architecture from a set of patterns. This capability is especially valuable when the architecture involves a complex structure in a specialized programming domain, like distributed computing, where the developer may lack experience. Instead, the developer can quickly build the architecture from the patterns and then focus on the application code that lies in their area of expertise.

The concepts in the paper are demonstrated using RMA [6, 7], an Eclipse plugin [15] for creating Java 2 Enterprise Edition (J2EE) version 2.1 applications using a framework composed of a set of design patterns [2]. RMA provides an incremental development model that exploits framework, pattern, and application information to derive the implementation of most of the responsibilities in the architecture for a J2EE program. Users can focus on their application code rather than the complexities of the J2EE environment.

The research contributions of this paper are:

- showing how to exploit roles and responsibilities to generate the implementation of a design pattern,
- leveraging an existing framework to not only generate design pattern implementations but also construct a complete architecture, and
- demonstrating these ideas using RMA, an Eclipse plugin for building J2EE applications.

2 Roles and Responsibilities

At its simplest, a *role* is a participant in a relationship between entities. Each role has a set of *responsibilities* that an implementation must implement to successfully play the role. Only when all of the responsibilities are properly implemented is the relationship complete.

As an example of roles and responsibilities, consider two machines transferring a file using FTP. There are two roles in this relationship: a server (that holds the file) and a client (that wants the file). To instantiate this relationship, we must assign computers to each role. When selecting computers for roles, we must ensure that they implement their responsibilities. The server needs an Internet connection and an FTP server. The client needs an Internet connection and an FTP client. Only then is the FTP client and server relationship established.

Pattern relationships can be built in an analogous manner. A role is played by a class and a responsibility is a property of that class that must be true. These properties can range from the class type (concrete or abstract class, or the set of superclasses it inherits from) to the signatures and implementation of the methods.

Given the general nature of responsibilities, it is useful to categorize them. This leads to three forms of responsibility: *static responsibility*, *derivable responsibility*, and *non-derivable responsibility*. A static responsibility is implemented consistently across all applications. For example, a class for an EJB bean (which is the basic remote object abstraction in a J2EE application) must implement the `SessionBean` interface.

Derivable responsibilities are the most common. The implementation of these responsibilities can be derived from application context. Sometimes this context may not have a unique way to implement the responsibility. When there are multiple possibilities, the user can be called on to select the desired option. If this selection process does not involve coding (*i.e.*, the choices can be selected from a simple GUI dialog), we still consider the responsibility to be derivable. A simple example of a derivable responsibility is the class naming convention used by J2EE for different parts of a bean class. A bean class must end with `Bean` (*i.e.*, `XYZBean`) and must implement two interfaces (named `XYZ` and `XYZHome`). The contents of these interfaces are also derivable responsibilities [6, 7]. Thus, from a bean class name, we can generate these two interfaces.

Non-derivable responsibilities are application-specific code that cannot be generated automatically. The implementation is left to the developer. For example, J2EE programs normally have a database component. Queries to the database are necessarily specific to the application, and thus must be supplied by the user.

3 Use of Roles and Responsibilities in a Programming System

Given a set of pattern relationships expressed as roles and responsibilities, we can now consider how to use them in a design-pattern-based programming system. Before we do, we should examine the characteristics of design patterns our system should preserve.

3.1 Key Design Pattern Characteristics

A brief, commonly-used definition of a design pattern is a solution to a recurring design problem in a particular context [9]. This definition suffices for most needs. When considering tool support, we need to be more thorough. A crucial point missing from the simple definition is that a pattern is not a single solution, but rather is an outline of a solution that can be applied in a given context. This outline must be adapted for the application in which it is applied. The context of a pattern is too broad to be captured by a single solution. For example, the “Implementation” section in the patterns in [9] describes common pattern variations. It is better to think a pattern as a family of design solutions, where the design pattern describes the basic structure of the solution and explains how to create other members of the family.

Unfortunately, this crucial point is sometimes missed in other design-pattern-based tools [14, 3], which only support a single pattern variation. This limits the usefulness of the tool since the single pattern form will not be useful in all cases. Variations of the pattern can be presented as separate patterns, but the resulting explosion in design pattern space is difficult to manage. Adding to the problem is that many of these tools hide the implementation of the pattern, so the single pattern form cannot be changed to create a variant. Instead, users are left to implement most or all of the pattern.

The idea of a pattern as a solution family is important because it impacts the needed roles and responsibilities. Different family members may add or remove roles in a pattern structure, which also changes the responsibilities and their implementation. The Adapter pattern [9], with its object-based and class-based versions, is an example where the roles differ. More commonly, a pattern variation redistributes responsibilities across the roles. The location of the child management methods in the Composite pattern [9] provides an example of such changes.

It is also important that the user be able to use meaningful class and method names in their code. Most classes and method in a pattern are application-specific, where the pattern dictates their interactions.

3.2 Support for Roles and Responsibilities

Before we examine support for roles and responsibilities, it is useful to distinguish between a design pattern and the intermediate pattern form that will be used in a programming system [13]. There are two key differences between the pattern and its intermediate form. First, the intermediate form is more concrete than the underlying pattern. Second, the intermediate form may only provide a subset of the complete family represented by the pattern. We call the intermediate form a *generative design pattern*, using the terminology from [13].

Given the characteristics of design patterns that we want to preserve in our tools, the process of instantiating a pattern consists of three steps: 1) Select the generative pattern corresponding to the desired design pattern, and the specific variant that is most applicable for the application. 2) Based on the pattern variant, select existing classes or create new classes to play the necessary roles. 3) Implement the responsibilities in the classes that make up the pattern implementation.

These steps are detailed in the following sections.

Selecting Design Pattern Variants The first step is to select the generative pattern for the desired design pattern. This is a design decision. As we are focused on development, we assume this choice has been made. In some application domains, frameworks that describe the outline of a complete application may exist. These frameworks can be used during the design phase and may also assist in the implementation of the application.

Once the pattern is selected, the specific variant of the solution family is selected. This can be done by having the generative pattern export a set of options that can be used to construct these variants [13]. Without some means of specializing the pattern at this stage of development, we limit the benefits of design patterns in the programming tool. More importantly, it has been argued that a single framework cannot encompass all variations of design patterns [13]. Thus, some specialization must be done before code is generated.

Assigning Classes to Roles Once the pattern variant is selected, there are a set of roles that must be assigned to classes. This assignment can be done in two ways. First, a new class can be created for the role, which is the simpler approach. The application logic can be inserted into these classes. The user may have existing code with this logic, in which case the new class can delegate its methods. The main drawback to this approach is the large number of classes that can result. Second, an existing class may be assigned to the role, allowing the user to integrate code into a program in a pattern-based tool.

The programming system may provide a method of assigning classes to roles, and can help by pruning the

set of potential classes that are presented for a role. This pruning may be based on knowledge from a framework or pattern language. By presenting only relevant classes, the tool can prevent user error and can ensure that the resulting software architecture is constructed correctly.

Implementing Responsibilities At this point in the process of implementing a pattern, the desired design pattern variant has been selected and classes have been assigned to the various roles. Now, the user must complete the process by implementing the responsibilities for each role. This section focuses on supporting this task through automatic code generation.

When considering automatic code generation, our categories of static, derivable, and non-derivable responsibilities become useful. Generating code for static responsibilities is straightforward since a single implementation suffices for all uses. The necessary code can be inserted into the class playing the role.

However, most code for a design pattern is not static, or a single framework would suffice [13]. The main reason is that design pattern implementations take on application characteristics, like class names and interfaces. These characteristics affect not only the class definition but also the pattern code, as the method interactions between classes must use them. For example, the interface to the composite class in the Composite pattern is dictated by the application. Further, the implementation of its methods is to iterate over its children calling the same application-specific method. These interactions cannot be efficiently implemented with static code, but rather must be generated based on the interface. Or rather, the implementation can be derived from information from the application and from knowledge of the pattern.

Information for derivable responsibilities can also come from the user. This input can be used to resolve ambiguities in the information derived from the application code or to map responsibilities to existing methods. As an example of user input, consider an Adapter generative pattern. The interface to the adapter class may be taken from an existing superclass, and so an outline of the adapter can be quickly created. The mapping between adapter and adaptee methods can be arbitrary, and so requires user input. For simple cases, it can be enough for the user to supply the name of the adaptee method that should be called in each adapter method.

A programming system may also exploit framework or pattern language information to assist the implementation of derivable responsibilities by reducing the set of options for user input. For example, when selecting a method to satisfy a responsibility, we can narrow the list of possibilities if we know that the required method must return a reference to a particular object of a specific type. We will see more concrete examples of this later.

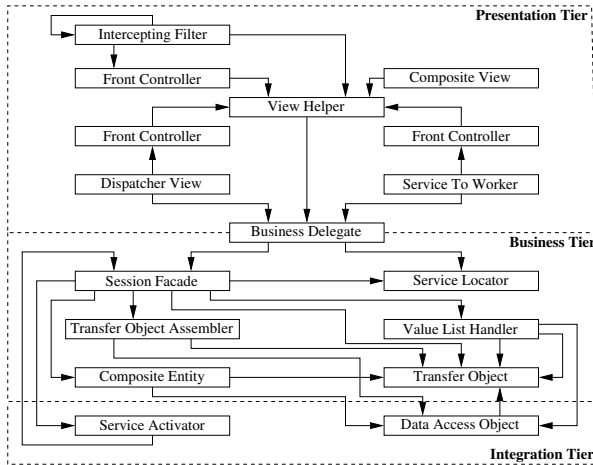


Figure 1. Simplified J2EE Framework [2].

While the static and derivable responsibilities can be automatically generated, non-derivable constraints cannot. However, this code is likely inside the area of expertise of the developer. Also, the non-derivable constraints represent the problem of interest, where the pattern implementation code is necessary infrastructure. Reducing the time to create this infrastructure allows the user to concentrate on their application.

4 J2EE Basics

4.1 J2EE Framework

Frameworks are partial implementations of the structure of a specific application. J2EE has a framework defined by Sun Microsystems [2], shown in Figure 1. It is split across three tiers to separate different parts of the overall application. The presentation tier provides a web-based user interface and displays the results of calls to business logic, and is implemented using servlets and JSPs. The business tier, the focus of this paper, provides business logic that processes client requests, and is usually built using EJBs. The integration or data tier provides database access. Note that the subcomponents of each tier are patterns. For space, the patterns are not described; see [2] for details.

4.2 Important J2EE Characteristics

J2EE has some important characteristics that must be considered in a pattern-based programming tool.

First, few J2EE patterns have significant structural variations. This is a function of the patterns; patterns in other areas have a richer set of alternate structures that a system should support [13]. Instead, it is the composi-

tion of patterns in the framework that makes these patterns useful. Tool support must consider composition.

Second, we have the J2EE framework to guide the implementation. This is useful because most of the patterns are composed with other patterns. Where collaborators are not patterns, they are J2EE entity beans that are easy to identify. For a generative design pattern, we can determine which classes can play its roles.

Third, relationships between the J2EE design patterns in the J2EE framework cannot be maintained by a pure top-down or a pure bottom-up approach. A pattern may not only use other patterns to play some of the roles it needs but it may also play a role in other patterns that already exist. An example will be given in Section 5. Tool support must address these kinds of relationships.

Fourth, although the goal of J2EE is to make web-based distributed applications appear like normal Java code, artifacts of the distributed environment still permeate the code. These artifacts communicate application structure and other information to the J2EE runtime environment. Some of these artifacts include interfaces needed to access methods in EJBs, a name space to locate EJBs, explicit creation of proxies for EJBs, and deployment descriptors that describe the application to the J2EE runtime. As well, the names of the classes, interfaces, and even some methods must obey the J2EE conventions. A good J2EE development environment should remove these artifacts, allowing the developer to concentrate on solving their problem rather than on the distributed J2EE runtime that will execute it.

5 Roles and Responsibilities in J2EE

We will show roles and responsibilities in J2EE design patterns using the *transfer value assembler* pattern. This pattern presents a logical data model to business tier components by aggregating data from multiple sources in the data tier. The *transfer value assembler* contains several *transfer value objects*, where each *transfer value object* contains a row of the database from a data source that makes up the logical data model. This saves clients from making several remote database requests and provides a single, consistent data model to the application.

The *transfer object assembler* pattern has three roles: an assembler role, an aggregator role, and a data source role. We elaborate on each role and its constraints.

The assembler role obtains *transfer value objects* from the data sources that comprise the physical data model. It then constructs an object for the aggregator role, giving it the *transfer value objects*, and returns that object to the client. The assembler must be played by an EJB since it must be accessible to remote clients. This description leads to the responsibilities for the assem-

bler. It must implement the responsibilities needed to be an EJB (Section 4.2). It must access data sources to get database records from the physical data model, and create the aggregator that provides the logical data model.

The aggregator role holds *transfer value objects* from a set of data sources and uses them to provide a logical data model. It acts much like a Facade pattern [9], providing methods to access the fields in the *transfer value objects* to hide the physical layout of the database. This role is usually played by a normal Java object at the client, not an EJB. One responsibility of this role is that it must have an instance variable for each *transfer value object*. Another responsibility is that the aggregator must provide accessor methods that are delegated to accessors on the correct *transfer value object*. The interface to the aggregator is typically the union of the *transfer value object* interfaces.

The data source role provides access to persistent application data. The class playing this role must provide methods to access and update the database using application-specific queries. All query methods have a search criteria, encapsulated as a *transfer value object* that is a parameter to the method. The methods return one or more database rows, where each is encapsulated in a *transfer value object*. In the simplest case, the data source has only one *transfer value object* associated with it, that hold all fields in the database table. However, it can be useful to define several *transfer value objects*, with a different subset of the fields, to reduce communication costs. Regardless, a *transfer value object* is associated with one data source, called its *owner*.

In the J2EE framework, the data source role is always played by the *data access object* (DAO) pattern [2]. The data source can range from a simple file to a commercial DBMS. The DAO provides a simple, consistent interface that hides the details of the persistent data store.

In addition to the above roles, the *transfer object assembler* plays a role in the *service locator* pattern [2]. The assembler role is played by an EJB, which must be assigned a name and registered in the Java Naming and Directory Interface service (JNDI) so other EJBs can obtain a reference to it. The responsibilities for this role are implemented by the *service locator*, but it requires that the locator be updated with information about the assembler when a *transfer object assembler* is created. This process is described in [6].

6 RMA

RMA is an Eclipse plugin that supports the J2EE patterns and framework [6, 7]. It provides wizards for the generative J2EE patterns it supports, which gather user input for derivable responsibilities. The generated code

is inserted into new or existing classes in an Eclipse project, and is accessible by the developer. Developers can also leverage features from other Eclipse plugins, including refactoring and code completion. Once the application is complete, RMA packages the application and the generated deployment descriptors into a J2EE Application Archive that can be run on a J2EE server.

6.1 J2EE Patterns and Framework in RMA

RMA supports pattern-based J2EE programming by exploiting structural information in the J2EE framework to help assign roles to the patterns that make up the architecture of an application, generating code for static and derivable responsibilities, and providing an Eclipse task list for non-derivable responsibilities.

RMA promotes an incremental development model to help compose the patterns in the J2EE framework. Applications are generally developed starting from the integration tier to the presentation tier in Figure 1. This allows RMA to gather information about the existing components when requesting user input for a generative pattern, and use that information to generate code for derivable responsibilities. Because RMA uses information from the complete application, it has a large set of derivable responsibilities and can generate a substantial amount of code for an application [6]. However, RMA does not enforce this development strategy. The developer can either implement responsibilities manually or run the RMA wizards several times to generate it.

In addition, RMA provides additional abstractions to simplify J2EE application development, outlined in Section 4.2. These abstractions are described in [6, 7].

6.2 Constructing a *Transfer Value Assembler*

In this section, we will build the *transfer value assembler* described in Section 5 using RMA.

To begin the process, the user selects the *transfer value assembler* generative pattern in the Eclipse interface. RMA then presents a wizard to gather information about the three roles (assembler, aggregator, and data source) for the pattern. First, RMA requires the user to specify the *transfer value objects* that will make up the logical data model. The wizard is shown in Figure 2. In the example application, information about users is held in two tables, Client and Address, holding client password and profile information. The logical model should be the union of these two tables. In Figure 2, the *transfer value objects* for these two tables have been selected.

From the selected *transfer value objects*, we can identify the data sources that hold the records to be aggregated using the ownership relation (from Section 5). This relation is given in the left-most column in Figure 2.

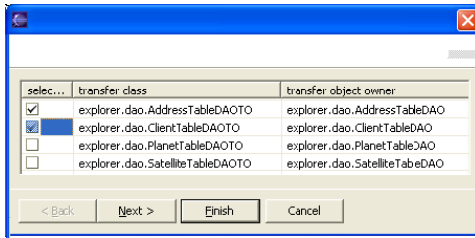


Figure 2. Building the logical data model.

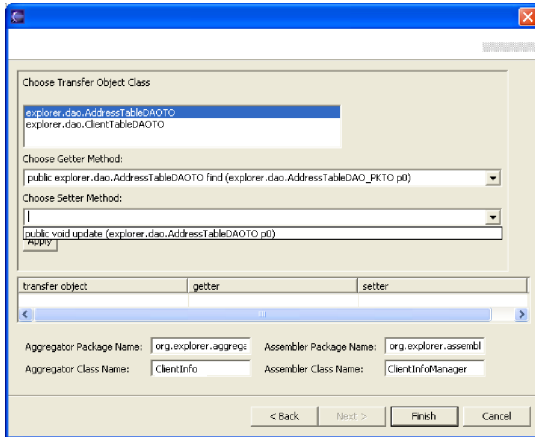


Figure 3. Selecting database methods.

For each *transfer value object* selected in the first part of the wizard, we need to issue database queries that return the desired record from the associated data source. In addition, the *transfer value assembler* supported by RMA allows the data source to be updated, so an update method can also be chosen. This is the second part of the wizard, shown in Figure 3. The *transfer value objects* selected in the first part of the wizard are listed at the top, and the developer can select query methods from the owner data source for each.

This wizard shows how RMA uses application information. The getter method must be a method in the owner data source that returns an instance of the *transfer value object* class selected at the top of the wizard. The setter must have a parameter of this type. This information can be used to filter out the list of candidate methods that are available for selection. This filtering is not foolproof but suffices for many applications.

From the wizard, RMA generates an almost-complete implementation of the *transfer value assembler*. The wizard gathers information to generate the derivable responsibilities. The non-derivable responsibility is initializing the database query criteria, which is application-specific. The pattern implementation totals 118 lines of code, of which the user writes 7.

An important consideration is that the *transfer value assembler* is played by an EJB bean. This requires

the class have extra responsibilities, such as extra interfaces (which include remote exceptions and are not implemented by the application classes, which is counter-intuitive) and registration with a name server (which requires the construction of a name space). Further, instantiating an EJB bean takes several steps. These requirements are complicated by J2EE naming conventions, which include renaming certain methods in the interfaces while maintaining the rest of the signature. RMA automatically handles these extra complications in the generated code, as detailed in [6, 7].

6.3 Initial Evaluation of RMA

To more fully evaluate RMA, we implemented a sample application based on an astronomy club. The application required user creation and management (login, user profile) before displaying the contents of a simple database. The application required four database tables, two for user management and two for application data. The application database presented the data on a page-by-page basis as it was too large for a single page.

The complete application required 2039 lines of code, of which RMA generated 1507 (74%). Of the 532 lines of user code, 438 (82%) was needed for database queries, which is necessarily application-specific. The remaining 94 lines of code mainly consisted of method co-ordination code that could not be automatically generated. Overall, it took less than 10 minutes to assemble the patterns using RMA wizards, and less than 2 hours to complete the non-derivable responsibilities. It must be noted that these numbers are a function of the application, but they do suggest reduced development effort.

A more detailed description of the application and analysis of the code can be found in [6, 7]. A more thorough evaluation of the benefits of RMA, using a larger application, is future work. We believe the benefits of will scale to larger applications [6].

7 Related Work

The relationships between patterns, pattern languages, and frameworks/middleware has been explored [4, 16]. Frameworks and middleware often use pattern-based components, and a pattern language may guide the process of assembling them into an application. RMA provides tool support for the assembly process.

A number of other tools that generate pattern implementation code exist. One of the first was the work of Budinsky *et. al.* [5] for generating code for the patterns in [9]. This tool accounts for the structural alternatives in the patterns. However, it generates generic code as it does not consider application-specific interfaces in the

created code. Further, it generates all classes for individual patterns; it does not allow existing classes to be assigned to pattern classes and does not have a framework for describing a complete application architecture. The CO₂P₃S system for parallel design patterns also generates individual patterns, though the generated code also incorporates some application-specific interface information. RMA uses the J2EE framework and application interfaces when generating code, expanding the set of derivable constraints compared to the above systems.

Fred (later renamed JavaFrame) [10] and MADE [11] both use roles and constraints (almost identical to our responsibilities) to guide developers through the framework construction process. Both tools use this information primarily to produce a task list that the developer follows to complete the application. While both tools have some code generation capabilities, they only generate code for static constraints and for a small subset of derivable constraints. The scope of derivable information appears to be limited to that from the current class and the J2EE runtime library. RMA uses the task list to highlight non-derivable constraints in its generated code. Compared to Fred and MADE, RMA has a larger set of derivable responsibilities for generating code. Information is derived from the all classes in the program and from user input, and is used to generate compositional code and method-level interactions.

Work is underway on version 3.0 of the J2EE specification, which removes some of the limitations in version 2.1 [8]. Version 3.0 uses the new Java metadata annotation facilities [12] to remove many of the distributed artifacts in J2EE programs.

8 Conclusions and Future Work

This paper showed that the implementation of a design pattern can be viewed as the process of assigning roles to classes and implementing the responsibilities of the role. A role is a participant in a relationship, a class in our case. A responsibility is a property the class must exhibit to play the role. Responsibilities can be categorized as static, derivable, and non-derivable.

From this viewpoint, we argued that tool support for pattern-based programming involves three steps: selecting a pattern variant, assigning roles to classes, and implementing responsibilities. We use information from the application code and user input to generate code for static and derivable responsibilities. These ideas were demonstrated using RMA, an Eclipse plugin that supports pattern-based J2EE programming. RMA exploits J2EE patterns and the J2EE framework, which shows how to construct a complete application architecture.

RMA currently supports patterns from the business

and integration tiers. Future work could include support for the presentation tier to complete the development process. In addition, these ideas should be applicable to other application domains, patterns, and frameworks. We could consider building similar plugins for frameworks like ACE [16]. With an extensible version of RMA (like MetaCO₂P₃S for creating generative patterns in CO₂P₃S [13]), it might be possible for framework developers to create their own plugins.

Acknowledgements

This research was supported by the Natural Science and Engineering Research Council of Canada and the University of Waterloo.

References

- [1] G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering Methodology*, 4(4):319–364, 1995.
- [2] D. Alur, J. Crupi, D. Malks. *Core J2EE Patterns, Best Practices and Design Strategies*. Sun Microsystems Press, 2nd ed., 2003.
- [3] D. Astels. Introduction to pattern automation. <http://bdn.borland.com/article/0,1410,29927,00.html>.
- [4] D. Brugali, K. Sycara. Frameworks and pattern languages: An intriguing relationship. *ACM Computing Surveys*, 32(1es), 2000.
- [5] F. Budinsky, M. Finnie, J. Vlissides, and P. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.
- [6] J. Chen. RMA: A pattern-based J2EE development tool. Master's thesis, School of Computer Science, Univ. of Waterloo, 2004.
- [7] J. Chen and S. MacDonald. RoadMapAssembler: A new pattern-based J2EE development tool. In *Proc. CASCON 2005*, pages 113–127, 2005.
- [8] EJB 3.0 Expert Group. *JSR 220: Enterprise JavaBeans, Version 3.0: EJB 3.0 Simplified API*. <http://java.sun.com/products/ejb/docs.html>, 2005.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [10] I. Hammouda and K. Koskimies. A pattern-based J2EE application development environment. *Nordic Journal of Computing*, 9(3):248–260, 2002.
- [11] I. Hammouda, J. Koskinen, M. Pussinen, M. Katara, and T. Mikkonen. Adaptable concern-based framework specialization in uml. In *Proc. 19th Intl. Conf. on Automated Software Engineering*, pages 78–87, 2004.
- [12] Java Community Process. *A Program Annotation Facility for the Java Programming Language*. JSR 175. <http://jcp.org/aboutJava/communityprocess/review/jsr175/index.html>.
- [13] S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling, and K. Tan. Generative design patterns. In *Proc. 17th Intl. Conf. on Automated Software Engineering*, pages 23–34, 2002.
- [14] ModelMaker Tools. Design patterns in modelmaker. <http://www.modelmakertools.com/modelmaker/design-patterns.html>.
- [15] Object Technology International, Inc. *Eclipse Platform Technical Overview*, February 2003. Available at: <http://www.eclipse.org/whitepapers/eclipse-overview.pdf>.
- [16] D. Schmidt and F. Buschmann. Patterns, frameworks, and middleware: Their synergistic relationships. In *Proc. 25th Intl. Conf. on Software Engineering*, pages 694–704, 2003.