Better Generative Programming with Generic Aspects

Raul Silaghi and Alfred Strohmeier

Software Engineering Laboratory
Swiss Federal Institute of Technology in Lausanne
CH-1015 Lausanne EPFL, Switzerland

E-mail: {Raul.Silaghi, Alfred.Strohmeier}@epfl.ch

Abstract. After a brief introduction to generative, generic, and aspect-oriented programming, we point out four key elements that appear in the definition of generative programming and that are addressed in this position paper from the perspective of distributed systems development. Then, based on a concrete RMI distribution example, we start motivating how the expressiveness power of generics and the crosscutting modularization power of aspects could be combined in order to achieve highly reusable *generic aspects*. We conclude by presenting how generic concern-oriented model transformations could help in providing the necessary information to *aspect generators* for automatically instantiating our generic aspects before weaving them into concrete applications.

Keywords. Generative Programming, Generics, Aspect-Oriented Programming, AOP, Model Transformations.

1 Introduction

Remarkably, much of software engineering today is still carried out by manual methods. Significant productivity enhancements require automation, which in turn require tools that deeply understand programs. Generative programming is a class of tool technology that captures knowledge about how to generate code, enabling automation.

Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations. In the context of generative programming, the principles of generic programming are applied to the solution space, where we try to come up with a very generic set of implementation components that minimize redundancy, maximize the number of their possible concrete instantiations, and support a vast number of combinations to yield very efficient, concrete applications.

At a more abstract level, generic programming focuses on representing families of domain concepts, whereas generative programming also includes the process of creating concrete instances of concepts.

Separation of concerns is one of the software engineering design principles that is getting more attention from practitioners and researchers in order to promote design and code reuse. However, concern is a broad term, encompassing anything that is of importance to the application, be it infrastructure, code, requirements, design artifacts, etc.

Some concerns, such as data and functions, can be very well encapsulated using object-oriented mechanisms. Others, such as logging, profiling, distribution, transactions, or security, cannot; their implementation is usually expressed by small code fragments scattered throughout the system. Aspect-Oriented Programming (AOP) [KLM$^+$97] provides the user with the ability to modularize such crosscutting concerns into *aspects* in order to solve the code-tangling problem, ease the development and maintenance of applications, and maximize code reusability.

When building distributed systems, different middleware-specific crosscutting concerns need to be integrated along with the core functionalities of the application. While AOP provides a good mechanism to deal with such crosscutting concerns, generative and generic programming can help to deal with the corresponding implementation aspects in an automatic way, completely transparent for the application programmer.

This position paper was inspired by the following definition of generative programming:

> Generative Programming is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configurations knowledge [CE00].

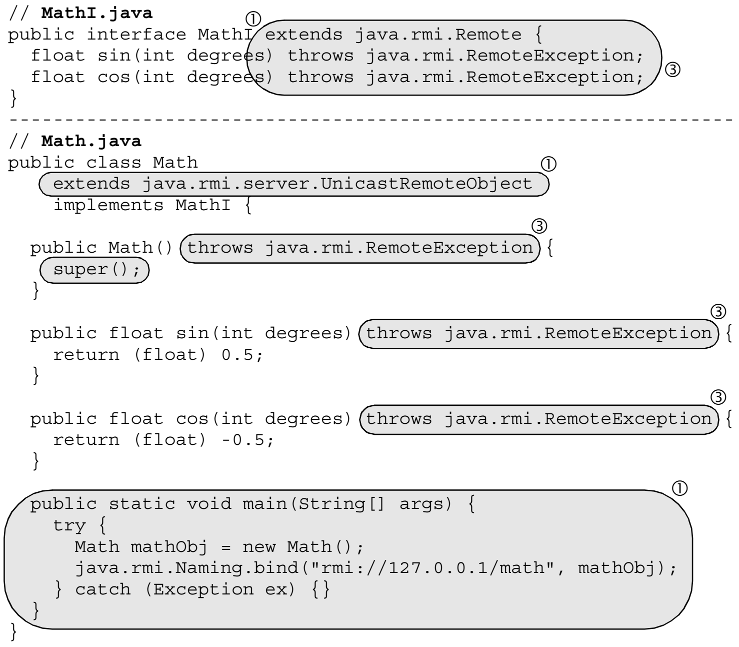
Several key elements on which generative programming is based upon are revealed by this definition. However, in the context of this position paper, we will only focus on four of these elements from the perspective of distributed systems development: (1) families of software systems; (2) reusable implementation components; (3) customization; and (4) automation.

The outline of the rest of this position paper is as follows: Section 2 starts with a motivating example and describes the important parts of a possible aspect-oriented implementation solution; Section 3 presents generic aspects as a better way to improve reusability and at the same time deal with middleware-specific crosscutting concerns for families of distributed systems; Section 4 makes a one-to-one association between aspects and model transformations in order to show how customization could be automatically achieved, and Section 5 draws some conclusions.

2 Motivating Example

In this section, we present a small example that every developer has to deal with when building distributed systems. Suppose we have an object that provides certain functionalities, like a `math` object that can compute trigonometric functions, and now we would like to make this object's functionalities available in a distributed setting. First, the developer needs to select one of the currently available middleware technologies. Further on, s/he has to embed the `math` object into either an RMI [rmi99] server object, a CORBA [corba02] server object or a CORBA component [ccm02], an EJB [ejb01], a COM/COM+/DCOM [com03] or .NET [net03] server object, or into a Web Service [ws03], depending on the previously chosen technology. To conclude, s/he has to modify all client applications to make them aware of the, by now, remotely available functionalities of the `math` object.

For space reasons only, we have considered the simplest solution, i.e., implement the `math` object as an RMI server object. In Figure 1 and Figure 2, we show the entire



```
// MathI.java                ①
public interface MathI extends java.rmi.Remote {
  float sin(int degrees) throws java.rmi.RemoteException;
  float cos(int degrees) throws java.rmi.RemoteException;   ③
}
----------------------------------------------------------------
// Math.java
public class Math                                   ①
    extends java.rmi.server.UnicastRemoteObject
    implements MathI {
                                                  ③
  public Math() throws java.rmi.RemoteException {
    super();
  }
                                                            ③
  public float sin(int degrees) throws java.rmi.RemoteException {
    return (float) 0.5;
  }
                                                            ③
  public float cos(int degrees) throws java.rmi.RemoteException {
    return (float) -0.5;
  }
                                                              ①
  public static void main(String[] args) {
    try {
      Math mathObj = new Math();
      java.rmi.Naming.bind("rmi://127.0.0.1/math", mathObj);
    } catch (Exception ex) {}
  }
}
```

Fig. 1. Server-Side Code for an RMI-based Distribution Example

code present at the server-, respectively client-side, of the application. On both sides we use rounded rectangles to highlight the changes that need to be performed in order to "transform" a centralized application into an RMI-based distributed one. The strikethrough line in Figure 2 is from the original centralized code and needs to be replaced by the content of the rounded rectangle immediately below it. Please notice that, following the design by contract principles [Mey92, Mey02], even in the centralized application, the client is supposed to have a reference to the interface `MathI` and not to the `Math` class. If the developer does not comply to such programming rules, then the "transformation" might produce unpredictable results.

As one can easily notice from the two figures, the code that handles the RMI-based distribution concern is not at all well localized. Instead, it is scattered throughout the whole application, crosscutting the main functional units of the system but without bringing any new user-defined functionality. As a consequence, an AOP approach seems to be the ideal solution to encapsulate such a crosscutting concern into a separate distribution unit.

AspectJ [KHH$^+$01] is a general-purpose aspect-oriented extension to Java [GJS$^+$00]. It defines one new concept, a join point, and adds a few new constructs, such as pointcut, advice, introduction, and aspect. *Join points* are well-defined points in the program flow; *pointcuts* are a means of referring to collections of join points and con-

```
// Student.java
public class Student {
  private MathI mathI = null;

  public Student() {
  }

  private MathI getMathInterface() {
    if (mathI == null) {
      ~~mathI = new Math();~~
                                                                  ②
      try {
        mathI = (MathI) java.rmi.Naming.lookup("rmi://127.0.0.1/math");
      } catch (Exception ex) {}
    }
    return mathI;
  }

  public void displayValues() {
    float sin=0, cos=0;
    MathI localMathI = getMathInterface();
 ②  try {
      sin = localMathI.sin(30);
      cos = localMathI.cos(120);
    } catch (java.rmi.RemoteException ex) {}
    System.out.println("sin(30) = " + sin + "\n" +
                       "cos(120) = " + cos + "\n");
  }

  public static void main(String[] args) {
    Student s = new Student();
    s.displayValues();
  }
}
```

Fig. 2. Client-Side Code for an RMI-based Distribution Example

text values at those joint points; *advice* defines code that is executed when a pointcut is reached during execution; *introduction* can be used to affect the static structure of Java programs, namely the members of its classes and the relationships between classes; and *aspects* are AspectJ's modular units of crosscutting implementation defined in terms of pointcuts, advices, introductions, and ordinary Java member declarations.

Having in mind the features offered by AspectJ, we can group the changes highlighted in Figure 1 and Figure 2 into three different categories: changes that affect the static structure of the Java application (①); changes that affect the dynamic structure of the application by changing the way the application executes (②); and changes related to exception throwing declarations, which can be seen as modifying the static structure but were grouped separately due to a reason that will be presented later in this section (③).

Some of these changes can be easily implemented using AspectJ, others require workarounds, while still others require extensions to the current version of AspectJ. For instance, using AspectJ's static crosscutting mechanism, one can introduce new methods and fields to an existing class, convert checked exceptions into unchecked exceptions, and change the class hierarchy by making an existing class extend another one or implement a new interface. Thus, based on simple *introductions*, we can implement all the changes that fall into the first category (①):

```
declare parents: MathI extends java.rmi.Remote;
declare parents: Math extends java.rmi.server.UnicastRemoteObject;
public static void Math.main(String[] args) {...}
```

Regarding the second category (②), one can use *pointcuts* and *advices* in order to dynamically affect the application flow. Besides the code to be executed, the advice declaration also indicates if the code should be executed *before*, *after*, or *around* (instead of) each join point selected by the pointcut definition. Here is a small code snippet that shows how to intercept a call to any constructor of `Math` made from within an instance of `Student`, and make that call return a reference to the remote interface instead of a `Math` object:

```
pointcut mathNews(): call(Math.new(..)) && this(Student);
MathI around(): mathNews() {
  MathI iObj = null;
  try {
    iObj = (MathI) java.rmi.Naming.lookup("//127.0.0.1/math");
  } catch (Exception ex) {}
  return iObj;
}
```

With respect to the third category (③) there is not much to say except that the current version of AspectJ does not support that kind of static crosscutting yet. However, Soares et al. have already submitted a feature request to the AspectJ team on this subject and there are good chances that it will be integrated in the next version of AspectJ [SLB02]. The construct, which would extend AspectJ to allow adding exceptions to a method's `throws` clause, looks like this:

```
declare throws: (* MathI.*(..)) throws java.rmi.RemoteException;
```

This declaration would add the RMI specific exception, `RemoteException`, to the `throws` clause of all the methods of the `MathI` interface. The wildcards `*` and `..` have the same meaning as in any other AspectJ pointcut designator, i.e., match any return type and any method name, and any parameter list, respectively.

Another limitation of AspectJ is related to the return type of an `around` advice. Currently, it is not possible to declare a "general" `around` advice for a group of methods because the return type has to be explicitly specified and it might differ from one method to another. In our particular case, we can write something like:

```
pointcut callsToMath():
             ( call(public float Math.sin(int)) ||
               call(public float Math.cos(int)) ) &&
             this(Student);
float around(): callsToMath() {
  float value = null;
  try {
    value = proceed();
  } catch (java.rmi.RemoteException ex) {}
  return value;
}
```

However, in this example we rely very much on the fact that the two methods of the `Math` object return both a `float` value. If there were a third method to return an `int` value, then we would have to write two different pointcuts and appropriate `around` advices for each one.

Imagine now that the developer has several objects that s/he would like to "transform" into RMI server objects and make them available to remote clients. Using the previously presented approach, s/he would have to rewrite a different aspect for each such object, or at least define new pointcuts, introductions, and advices inside the same aspect but appropriately customized for each of his or her new objects. In both cases, s/he would end up with a lot of duplicate code, which is not a very acceptable solution.

Of course, there are ways to get rid of duplicate code and increase reusability to a certain degree, and one very nice solution is to use AspectJ idioms, such as Abstract Pointcut, Template Advice, Pointcut Method, and so on [HUS03]. Most of these idioms are based on abstract aspects that the developer has to extend and specialize according to his or her specific needs by defining concrete pointcuts or overloading operations. This approach relies on the premise that the developer has deep knowledge about AspectJ, which should not be a requirement as long as we want to make distribution transparent to the application developer. S/he might be requested to customize some aspect parameters (e.g., Java-like variables), but not to write AspectJ code, or even worse, write AspectJ code that contains distribution elements as well.

3 Generic Aspects to the Rescue

After the aspect-oriented solution proposed in the previous section, we look now at how generic aspects can overcome some of the identified drawbacks and how they can help improve reusability. Extensions to AspectJ are also proposed for the previously presented limitations.

Generics, also commonly known as *parameterized types* or *parametric polymorphism*, is a well-established programming language feature whose advantages over dynamic approaches to generic programming (e.g., subtype polymorphism) are well-understood: safety (more bugs caught at compile time), expressivity (more invariants expressed in type signatures), clarity (fewer explicit conversions between data types), and efficiency (no need for run-time type checks).

Generics have been used under several forms in different programming languages for years. To name just a few, we can mention functors (parameterized modules) in Standard ML and Caml, generics in Ada and Eiffel, and, probably what made generics popular the most, the C++ Standard Template Library. Surprisingly, two of the last programming languages on the market, Java and C#, do not support parametric polymorphism yet, but only subtype polymorphism. However, besides the generic extensions that exist on both sides (e.g., GJ [BOS$^+$98], for Java, and Gyro [clrgen03], for C#), there are significant efforts to introduce support for generics in both programming languages [BCK$^+$01, Jcp03, KS01]. In fact, both providers (Sun and Microsoft) offer already prototype implementations of their compilers that support generics.

Generic parameters are type or value parameters about types. They allow us to avoid unnecessary code duplication in statically typed languages. Programming languages may provide generic parameters not only for procedures or functions, but also for classes, modules, packages, and so on. As aspects are just another unit of encapsulation introduced by AOP, we look at the benefits of supporting generic aspects in AspectJ from the perspective of our distribution example.

In Figure 3, we present a generic RMI-based distribution aspect that would trans-

```
  aspect RmiDistributionAspect<Interface, Server, Client>
    where Server implements Interface {

    // Configuration
    String ipAddress = "serverHostIPaddressGoesHere";
    String serverName = "desiredServerNameGoesHere";

    // The developer should not touch anything below this line!!!

    String url = "//" + ipAddress + "/" + serverName;

    // Server-side changes
    declare parents: Interface extends java.rmi.Remote;
①   declare throws: (public * Interface.*(..)) throws java.rmi.RemoteException;

    declare parents: Server extends java.rmi.server.UnicastRemoteObject;
    declare throws: (Server.new(..)) throws java.rmi.RemoteException;
①   declare throws: (public * Server.*(..)) throws java.rmi.RemoteException;

    public static void Server.main(String[] args) {
      try {
        Server serverObj = new Server();
        java.rmi.Naming.bind(url, serverObj);
      } catch (Exception ex) {}
    }

    // Client-side changes
    pointcut serverNews(): call(Server.new(..)) && this(Client);
    Interface around(): serverNews() {
      Interface iObj = null;
      try {
        iObj = (Interface) java.rmi.Naming.lookup(url);
      } catch (Exception ex) {}
      return iObj;
    }
                             ②
    pointcut callsToServer(Type T):
      call(public T Server.*(..)) && this(Client);
②   T around(Type T): callsToServer(T) {
      T obj = null;
      try {
        obj = proceed();
      } catch (java.rmi.RemoteException ex) {}
      return obj;
    }
  }
```

Fig. 3. Generic RMI-based Distribution Aspect

form any centralized application into an RMI-based distributed one, provided that we supply it with three classes corresponding to the `Interface`, the `Server`, and the `Client`. Moreover, the constraint that is set on the type parameters, requiring the `Server` to implement the `Interface`, has to be met as well. Support for such a feature is usually referred to as *constrained genericity*. The syntax used in Figure 3 was inspired from the currently existing syntax proposals for supporting generics in Java and C# programming languages.

The body of the generic RMI-based distribution aspect follows the same ideas as the ones presented in section 2, i.e., introductions, pointcuts, and advices for modifying the static and dynamic structure of the original application.

The syntax proposed by Soares et al. [SLB02] for adding exceptions to a method's `throws` clause was a little bit extended to allow the developer to specify the visibility of the methods as well (Fig. 3 ①). In this way, when modifying the `Server` class, we can specify that only `public` methods should throw `RemoteExceptions`. However, this solution still does not cover another case that might arise: the `Server` class might have `public` methods providing services to some other local classes, and thus those methods should not throw `RemoteExceptions`. To overcome this problem, it would be nice to be able to specify something like:

```
// "All the public methods of the Server
//  that can be found in the Interface as well"
declares throws:
            ( (public * Server.*(..)) &&
              within(Interface) ) throws java.rmi.RemoteException;
```

The `within` keyword specified above would have a different meaning than the one used in current AspectJ pointcut designators. Moreover, the syntax for selecting the methods would support the logical `!`, `&&`, and `||` operators as well.

With respect to the `around` advice limitation, we considered the `System.Type` construct that is present in C# and mainly serves reflection purposes. In our case, we define our pointcut that selects all the join points corresponding to calls to the `Server` `public` methods that have been made from within `Client` objects, and moreover, we bind the method return type `Type` at those join points (Fig. 3 ②). This way we know what return type to specify for the `around` advice.

With the current approach, having a concrete instantiation of the generic RMI-based distribution aspect, the developer still needs to customize some configuration parameters with respect to the host where the `Server` object will be deployed and the name that will be used to identify it. However, the instantiation of the generic RMI-based distribution aspect is not yet fully automatic. It still requires the developer to analyze his or her models, to figure out by him- or herself which are the classes corresponding to the `Interface`, the `Server`, and the `Client`, and only then perform a concrete instantiation.

4 Generic Model Transformations and Generic Aspects

In this section, after a brief presentation of the current trends in the domain of model transformations, we show how generic concern-oriented model transformations could help automatically customize generic aspects according to the particular needs of each application.

In the context of Model Driven Architecture (MDA) [MM01], model-to-model transformations play a very important role since they are responsible for the possible refinements that may occur between Platform Independent Models (PIMs), Platform Specific Models (PSMs), and in-between the two in both directions. Moreover, the automatic generation of application code from a PSM is viewed as a model-to-model transformation as well.

In the absence of a standard language for defining such model transformations, different approaches and technologies are appearing. For instance, one approach proposes to encode model transformations in a procedural language, the models being retrieved from some UML-tool repository using a well-defined API. Another proposal, which promises to raise the level of abstraction of operations on UML models, is to use UML's action language [uml03] as a way to procedurally define UML transformations [SPH+01]. One interesting technique is to treat UML models as graphs and use the body of knowledge in graph transformation theory to define model transformations [SWZ97, BEW02, Fuj03, SPG+03]. Logic programming languages have also been used in the context of model transformations [Whi02, CDE+01]. [Mil02] proposes a graphical language for specifying model transformations based on extended UML object diagrams. Approaches based on XMI [xmi02] and XSLT [xslt03] to describe model transformations exist as well [Wag01, PZB02]. UML's OCL [WK98] has also been proposed as a way to declaratively describe UML model transformations [PVJ02, SPL+01, KWB03]. A brief overview of some of the previously mentioned approaches to model transformations and some recommendations on the desirable characteristics of a language for describing model transformations can be found in [SK03]. OMG [Omg03] has also posted a Request for Proposals, called MOF 2.0 Query/Views/Transformations RFP [mofqvt02], in order to fill this model transformation language gap and add the much needed keystone to the MDA vision.

Generic concern-oriented model transformations, which were first introduced in [Sil03] and then integrated as a basic constituent part of the Enterprise Fondue software development method [SS03], propose to drive the refinement of models according to the different concerns that the final application needs to incorporate. The genericity is required in order to deal with the imminent differences that appear from one application to an other.

In our particular example, refinement needs to be performed along a middleware-specific *concern-dimension*, namely RMI-based distribution, and the genericity has to take care of those model elements in the RMI-based distribution *concern space* that have to be customized for our particular application, namely the classes corresponding to the `Interface`, the `Server`, and the `Client`. In Figure 4, we illustrate how a concrete RMI model transformation affects the design of our originally centralized application when migrating to an RMI-based distributed one. Even though the notation used in Figure 4 is fully UML compliant, the representation of model transformations is a pure intuitive one that, we believe, serves best the point that we would like to make. We used a parameterized stereotyped class to indicate the generic RMI model transformation. This class is further specialized into a concrete RMI model transformation by binding its type parameters. A constraint (not OCL compliant) is also enforced on the type parameters of the generic RMI model transformation.

As one can easily notice, each change introduced by the concrete RMI model transformation at design level has a corresponding element in the generic RMI-based distribution aspect (presented in Figure 3) that implements that change at code level once it is customized for that particular application. Moreover, provided that tool support is of-

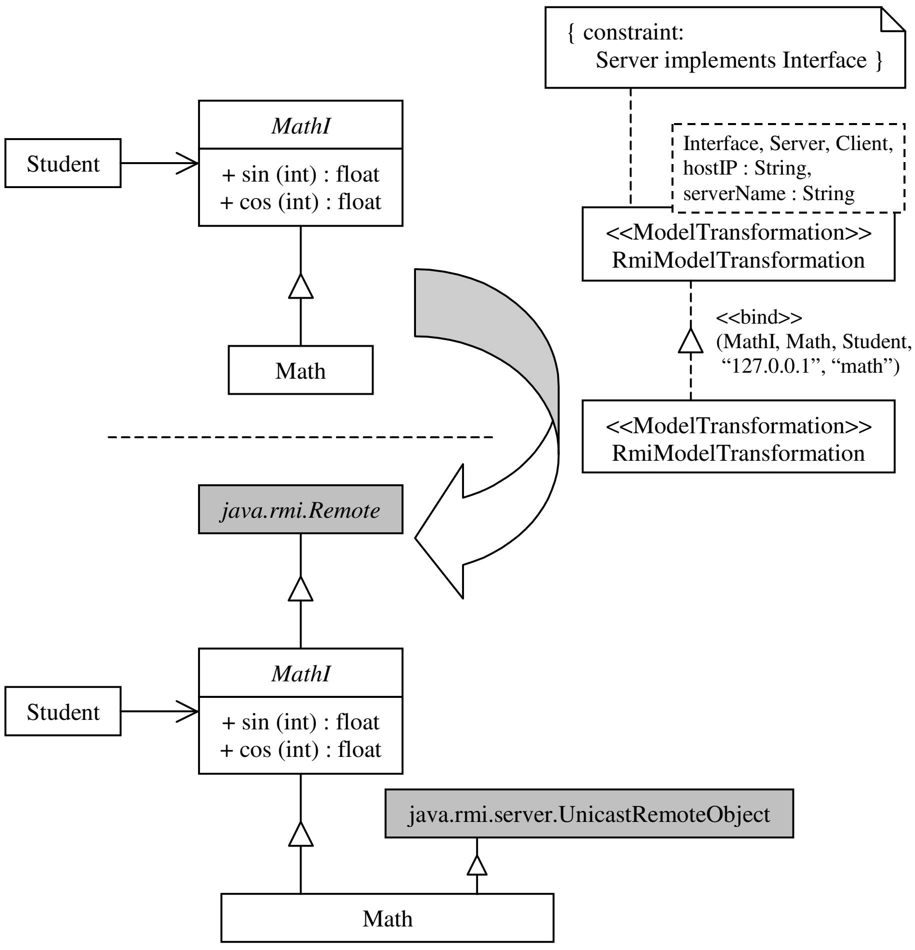


Fig. 4. A Generic/Concrete RMI Model Transformation

fered, customization of the generic aspect can be performed automatically based on the binding of the type parameters of the generic model transformation, since both the model transformation and the aspect use the same type parameters, namely the `MathI`, `Math`, and `Student` classes.

To conclude, a one-to-one mapping should exist between generic concern-oriented model transformations and generic aspects, and *aspect generators* should be used to instantiate generic aspects based on the information used to specialize the corresponding generic concern-oriented model transformations. In this way, we address a specific concern, in our case RMI-based distribution, at two different levels (abstract and concrete) and at two different layers (design and code) by applying model transformations specific to each layer (concern-oriented model transformations and aspects, respectively).

5 Conclusions

Based on a very simple example that transforms a centralized application into an RMI-based distributed one, we emphasized the benefits that the developer could gain from generics at the level of aspect-oriented programming. To automate the instantiation of *generic aspects*, we proposed to have a one-to-one association between generic concern-oriented model transformations and generic aspects, and to use *aspect generators* to instantiate generic aspects based on the information used to specialize the corresponding generic concern-oriented model transformations.

Looking back at the four key elements that we identified in the definition of generative programming quoted in the introduction, i.e., (1) families of software systems, (2) reusable implementation components, (3) customization, and (4) automation, we claim to have them addressed all to a certain degree in this position paper. As families of software systems, we addressed the narrow domain of RMI-based distributed systems. Generic aspects are the reusable implementation components that can be used to create several concrete instances of applications in the considered domain. Customization relies on the support for generics and is automated by using aspect generators out of generic concern-oriented model transformations.

References

[BCK+01] Bracha, G.; Cohen, N.; Kemper, C.; Marx, S.; Odersky, M.; Panitz, S. E.; Stoutamire, D.; Thorup, K.; Wadler, P.: *Adding Generics to the Java™ Programming Language.* Participant Draft Specification, April 2001.

[BEW02] Bardohl, R.; Ermel, C.; Weinhold, I.: *AGG and GenGED: Graph Transformation-Based Specification and Analysis Techniques for Visual Languages.* Electronic Notes in Theoretical Computer Science, **72**(2), Elsevier Science B.V., 2002.

[BOS+98] Bracha, G.; Odersky, M.; Stoutamire, D.; Wadler, P.: *Making the Future Safe for the Past: Adding Genericity to the Java Programming Language.* Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Vancouver, British Columbia, Canada, October 18-22, 1998. SIGPLAN Notices, **33**(10), ACM Press, 1998, pp. 183 − 200. More documents on GJ (including the specification) can be found at http://www.research.avayalabs.com/user/wadler/gj/.

[ccm02] Object Management Group, Inc.: *CORBA Components Specification*, v3.0, June 2002.

[CDE+01] Clavel, M.; Durän, F.; Eker, S.; Lincoln, P.; Marti-Oliet, N.; Meseguer, J.; Quesda, J.: *Maude: Specification and Programming in Rewriting Logic*. Theoretical Computer Science, 2001.

[CE00] Czarnecki, K.; Eisenecker, U. W.: *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[clrgen03] Microsoft, Inc.: *Generics for C# and .NET CLR.* http://research.microsoft.com/projects/clrgen/, August 2003.

[com03] Microsoft, Inc.: *COM (Component Object Model)*, *COM+*, *DCOM (Distributed COM).* http://www.microsoft.com/com/, August 2003.

[corba02] Object Management Group, Inc.: *The Common Object Request Broker: Architecture and Specification*, v3.0, July 2002.

[ejb01] Sun Microsystems: *Enterprise JavaBeans^{TM} Specification*, v2.0, August 2001.

[Fuj03] Fujaba Project Web Site, http://www.fujaba.de, August 2003.

[GJS^{+}00] Gosling, J.; Joy, B.; Steele, G.; Bracha, G.: *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[HUS03] Hanenberg, S.; Unland, R.; Schmidmeier, A.: *AspectJ Idioms for Aspect-Oriented Software Construction*. Proceedings of the 8th European Conference on Pattern Languages of Programs, EuroPLoP, Irsee, Germany, June 25–29, 2003.

[Jcp03] Java Community Process: *Add Generic Types To The Java^{TM} Programming Language*. Java Specification Request, JSR#14, http://www.jcp.org/jsr/detail/14.jsp, August 2003.

[KHH^{+}01] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G.: *An Overview of AspectJ*. Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP, Budapest, Hungary, June 18-22, 2001. LNCS Vol. **2072**, Springer-Verlag, 2001, pp. 327 – 353.

[KLM^{+}97] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J.-M.; Irwin, J.: *Aspect-Oriented Programming*. Proceedings of the 11th European Conference on Object-Oriented Programming, ECOOP, Jyväskylä, Finland, June 9-13, 1997. LNCS Vol. **1241**, Springer-Verlag, 1997, pp. 220 – 242.

[KS01] Kennedy, A.; Syme, D.: *Design and Implementation of Generics for the .NET Common Language Runtime*. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, Snowbird, UT, USA, June 20-22, 2001. SIGPLAN Notices **36**(5), ACM Press, 2001, pp. 1 – 12.

[KWB03] Kleppe, A.; Warmer, J.; Bast, W.: *MDA Explained: The Practice and Promise of Model Driven Architecture*. Addison-Wesley, 2003.

[Mey92] Meyer, B.: *Applying Design by Contract*. IEEE Computer, 1992, pp. 40 – 51.

[Mey02] Meyer, B.: *Design by Contract*. Prentice Hall, 2002.

[Mil02] Milicev, D.: *Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments*. IEEE Transactions on Software Engineering, **28**(4), 2002, pp. 413 – 431.

[MM01] Miller, J.; Mukerji, J.: *Model Driven Architecture (MDA)*. Object Management Group, Draft Specification ormsc/2001-07-01, July 9, 2001.

[MMS02] Mili, H.; Mcheick, H.; Sadou, S.: *CorbaViews – Distributing Objects that Support Several Functional Aspects*. Journal of Object Technology, **1**(3), Special Issue: TOOLS USA 2002 Proceedings, pp. 207 – 229.

[mofqvt02] Object Management Group, Inc.: *MOF 2.0 Query/Views/Transformations RFP*. http://cgi.omg.org/cgi-bin/doc?ad/02-04-10, 2002.

[net03] Microsoft, Inc.: *.NET*. http://www.microsoft.com/net/, August 2003.

[Omg03] Object Management Group, Inc., http://www.omg.org/, August 2003.

[PVJ02] Pollet, D.; Vojtisek, D.; Jézéquel, J-M.: *OCL as a Core UML Transformation*

Language. Workshop on Integration and Transformation of UML Models, WITUML, at ECOOP, University of Málaga, Spain, June 10-14, 2002.

[PZB02] Peltier, M.; Ziserman, F.; Bézivin, J.: *On Levels of Model Transformation*. Proceedings of XML Europe, Paris, France, June 12-16, 2000.

[rmi99] Sun Microsystems: *Java^{TM} Remote Method Invocation Specification*. Revision 1.7, Java^{TM} 2 SDK, Standard Edition, v1.3.0, December 1999. http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html.

[Sil03] Silaghi, R.: *Generic Concern-Oriented Model Transformations Meet AOP*. Proceedings of the 1st International Workshop on Model-driven Approaches to Middleware Applications Development, MAMAD, at the ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16-20, 2003. Middleware 2003 Companion, PUC-Rio Press, 2003, pp. 307 – 311.

[SK03] Sendall, S.; Kozaczynski, W.: *Model Transformation – the Heart and Soul of Model-Driven Software Development*. IEEE Software, **20**(5), Special Issue on Model-Driven Development, 2003, pp. 42 – 45. An extended version is available as Technical Report, EPFL-IC-LGL N° IC/2003/052, July 2003.

[SLB02] Soares, S.; Laureano, E.; Borba, P.: *Implementing Distribution and Persistence Aspects with AspectJ*. Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Seattle, WA, USA, November 4-8, 2002. SIGPLAN Notices **37**(11), ACM Press, 2002, pp. 174 – 190.

[SPG$^+$03] Sendall, S.; Perrouin, G.; Guelfi, N.; Biberstein, O.: *Supporting Model-to-Model Transformations: The VMT Approach*. Proceedings of the 1st International Workshop on Model Driven Architecture: Foundations and Applications, MDAFA, University of Twente, Enschede, The Netherlands, June 26-27, 2003. Published as Technical Report, TR-CTIT-03-27, University of Twente, 2003, pp. 61 – 72.

[SPH$^+$01] Sunyé, G.; Pennaneac'h, F.; Ho, W-M.; Le Guennec, A.; Jézéquel, J-M.: *Using UML Action Semantics for Executable Modeling and Beyond*. Proceedings of the 13th Conference on Advanced Information Systems Engineering, CAiSE, Interlaken, Switzerland, June 4-8, 2001. LNCS Vol. **2068**, Springer-Verlag, 2001, pp. 433 – 447.

[SPL$^+$01] Sunyé, G.; Pollet, D.; Le Traon, Y.; Jézéquel, J-M.: *Refactoring UML Models*. Proceedings of the 4th Conference on the Unified Modeling Language: Modeling Languages, Concepts, and Tools, UML, Canada, 2001. LNCS Vol. **2185**, Springer-Verlag, 2001, pp. 134 – 148.

[SS03] Silaghi, R.; Strohmeier, A.: *Integrating CBSE, SoC, MDA, and AOP in a Software Development Method*. Proceedings of the 7th IEEE International Enterprise Distributed Object Computing Conference, EDOC, Brisbane, Queensland, Australia, September 16-19, 2003. IEEE Computer Society Press, 2003, pp. 136 – 146. Also available as Technical Report, EPFL-IC-LGL N° IC/2003/057, September 2003.

[SWZ97] Schürr, A.; Winter, A.; Zündorf, A.: *The Progres Approach: Language and Environment*. In Chapter 13 of Rozenberg, G. (eds), Handbook of Graph Grammars and Computing by Graph Transformation: Volume II Applications, Languages, and Tools. World Scientific Publishing, 1997.

[uml03] Object Management Group, Inc.: *Unified Modeling Language Specification*, v1.5, March 2003.

[Wag01] Wagner, A.: A Pragmatic *Approach to Rule-Based Transformations within UML using XMI.difference*. Workshop on Integration and Transformation of UML Models, WITUML, at ETAPS, 2001.

[Whi02] Whittle, J.: *Transformations and Software Modeling Languages: Automating Transformations in UML*. Proceedings of the 5th Conference on the Unified Modeling Language: The Language and its Applications, UML, Dresden, Germany, September 30 - October 4, 2002. LNCS Vol. **2460**, Springer-Verlag, 2002, pp. 227 − 242.

[WK98] Warmer, J.; Kleppe, A.: *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[ws03] World Wide Web Consortium: *Web Services*. http://www.w3.org/2002/ws/, August 2003.

[xmi02] Object Management Group, Inc.: *XML Metadata Interchange (XMI) Specification*, v1.2, January 2002.

[xslt03] World Wide Web Consortium: *eXtensible Stylesheet Language Transformations*, v1.0. http://www.w3.org/TR/xslt/, August 2003.