Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 53 (2004) 51–69

Science of Computer Programming

www.elsevier.com/locate/scico

A Java-based approach for teaching principles of adaptive and evolvable software

Jeff Gray*

Department of Computer and Information Sciences, University of Alabama at Birmingham, 1300 University Blvd., Birmingham, AL 35294, USA

Received 2 September 2003; received in revised form 10 February 2004; accepted 18 February 2004

Available online 10 June 2004

Abstract

The ability to adapt a software artifact is essential toward handling evolving stakeholder requirements. Adaptation is also vital in many areas where software is required to adjust to changing environment conditions (e.g., the growing presence of embedded systems). Current techniques for supporting adaptability and evolvability can be categorized as *static* (happening at compile-time or design-time), or *dynamic* (adaptation during the actual execution of the system). This paper describes a special-topics software engineering course that uses Java as a foundation for teaching concepts of static and dynamic adaptation. The course surveys Java-related research in the areas of meta-programming and reflection, aspect-oriented software development, model-driven computing, and adaptive middleware.

Keywords: Java education; Evolvable software; Aspect-oriented software development; Model-integrated computing; Adaptive middleware

1. Introduction

> . . . program structure should be such as to anticipate its adaptations and modifications. Our program should not only reflect (by structure) our understanding

* Tel.: +1-205-934-8643; fax: +1-205-934-5473.
E-mail address: gray@cis.uab.edu (J. Gray).
URL: http://www.gray-area.org.

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.02.005

of it, but it should also be clear from its structure what sort of adaptations can be catered for smoothly [12].

A long-standing goal of software engineering is to construct software that is easily modified and extended [2]. The desired result is to achieve modularization such that a change in a design decision is isolated to one location of a program [42]. The proliferation of software in everyday life (e.g., embedded systems in automobiles, mobile phones, and television sets) has increased the conformity and invisibility of software [49]. As demands for such software increase, future requirements will necessitate new strategies for improved modularization in order to support the requisite adaptations. Additionally, investigation into pedagogical techniques for introducing these concepts into existing curricula is needed.

The ability to adapt software is typically partitioned among two stages: modifiability during development, and adaptation during execution. The first type of evolution is concerned with design-time, or compile-time, techniques that permit the modification of the structure and function of a software representation in order to address changing stakeholder requirements. To support such evolution, techniques such as aspect-oriented programming [27] and model-driven architecture [5] are but a few of the ideas that have shown promise in assisting a developer in isolating points of variation and configurability within software. The second type of adaptation occurs at run-time during the execution of a system. This type of adaptation refers to a system's ability to modify itself and to respond to changing conditions in its external environment. To accommodate such changes, research in meta-programming and reflection [38] has offered some recourse, especially in the application area of adaptive middleware [55].

This paper provides an overview of a special-topics software engineering course that adopts a Java-based approach for teaching concepts of software adaptation. The course examines the principles of "meta" and reflection as they support the concept of adaptation from several contexts. The general goals of the course are the following.

- Provide an introduction to the history and literature of reflection and meta-programming. This includes the introspective Java Reflection API, as well as research extensions to Java that support more powerful intercessional forms of reflection.
- Motivate the area of aspect-oriented software development (AOSD) as a direct result of past research into reflection. The majority of the semester is spent in this section of the course where students gain experience in applying AOSD ideas using enhanced Java translators, such as AspectJ [26].
- Introduce the issues surrounding meta-modeling and domain modeling within model-integrated computing (MIC) [50]. The generic modeling environment (GME) [32] is used as a platform for exploring model-based synthesis of Java applications.
- Expose students to applications of the above techniques for supporting adaptation of distributed object computing [54]. In particular, various research efforts in the area of reflective middleware are presented [55], in addition to modeling tools for generating middleware configurations [16].

The remaining sections of the paper provide a survey of the four focus areas of the course. The sections follow the chronological sequence of topics introduced in the course. The final section of the paper offers several concluding observations.

Specific details of the course (e.g., course schedule and additional references) can be found at http://www.cis.uab.edu/cs622.

2. Meta-programming and reflection from a Java perspective

Reflection and meta-programming are powerful techniques that provide support for adaptive systems. The general principles of reflective architectures form the underpinnings of most of the concepts introduced in the course. From this foundational perspective, the seminal papers on reflection are introduced early in the course in order to provide students with the proper vocabulary, as well as insight for understanding subsequent topics. The "Smithsonian" notion of reflection initiates students to the concept of having an internal representation of a program and the importance of causal connection between this representation and the external behavior of the program [48]. The important work by Maes is then introduced to students as an extension to the principles of reflection for object-oriented languages [38]. As additional background literature, students are exposed to Kiczales' work on meta-objects and CLOS [25]. A proper understanding of these early influential papers equips students with the knowledge necessary for further study of Java-based approaches supporting reflective adaptation.

2.1. The Java reflection API

Reflection permits a program to inquire about its own state at run-time (called *introspection*), and, in some cases, permits the modification of the semantics of the run-time system itself (called *intercession*). In a Java-based course on software adaptation, it is essential to discuss the introspective nature of the standard `java.lang.reflect` package. As the various features of the classes contained in this package are introduced to students, it has been found helpful to provide short exercises during the lecture to assist students in comprehending the capabilities provided. For example, the students may be asked to "Create a public static method named `NumOfMethods` that takes a single parameter of type `Object`. This method should return an integer value that represents the number of methods available in the `Object` parameter". Listing 2.1 shows that this exercise has a simple solution, but the implementation highlights the fact that an object's meta-class provides access into many of the other introspective capabilities. From initial exercises like this, students can be asked to implement increasingly difficult problems during the lecture period. The goal of these exercises is not to force memorization of the reflection classes and methods, but rather to offer opportunities that motivate the practice and use of

```
public static int NumOfMethods(Object s)
{
  Class c = s.getClass();
  Method[] m = c.getMethods();
  return m.length;
}
```

Listing 2.1. Implementation of a simple reflective in-class exercise.

standard Java reflection. This knowledge can be reinforced through additional questions on quizzes and homework assignments.

Initially, students may have difficulty grasping reflection's *raison d'etre*. Thus, it is essential to form these exercises around topics to which they can relate. As an example, the two previous offerings of this special topics course have featured guest lectures from developers in the local community who present their own real-world industrial examples that highlight the power of reflection as an aid toward improved implementation and adaptation. As the Java reflection packages are explored during the lecture, the students are instructed on why the core Java reflection capabilities offer only a weak form of introspective reflection. This understanding will motivate the need for later discussions pertaining to compile-time meta-object protocols (MOPs) in Java (see Section 2.4).

2.2. Dynamic class loading and run-time generation of bytecode

The topic of Java class loading presents several opportunities to discuss the benefits of adaptation at run-time. Liang and Bracha provide an excellent description of the capabilities of customized class loaders by introducing topics such as dynamic server evolution and run-time manipulation of bytecode [34]. They also mention the problems with type safety in earlier versions of the JDK and motivate the reason that a type in the JVM is a combination of the class name and the associated loader. These key concepts help students to understand the underpinnings of Java class loaders.

After the students are exposed to the idea of class loaders, a homework assignment can be given that combines the concepts of Java reflection with class loaders in a generative programming style [10]. An outline of a previous assignment is shown in Listing 2.2 (much has been removed to conserve space). The `generateCode` method is not shown here, but the implementation generates a Java source file at run-time by printing code to a `FileWriter` object. After the source file has been dynamically generated, it can be compiled at run-time, loaded into memory, and then invoked via reflective calls. In addition, the students can be asked to write an object browser that exposes the contents of an object in a tree-control using reflection. A student solution to the object browser problem can be found at the course web site (http://www.cis.uab.edu/cs622/sample).

2.3. Load-time manipulation tools

Although the dynamic creation and compilation of Java source code is an interesting idea, it is not always practical to use such techniques, especially when performance penalties are considered. Invasive software composition is a growing research area and is typified by instrumentation of software artifacts at various binding times [1]. As examples of invasive composition techniques, several implementations of bytecode instrumentation tools exist. Specifically, the course introduces the students to the JMangler [29] and JavaAssist [7] class loader manipulation tools. Each of these tools support class file interception and load-time adaptation. During the lectures in this part of the course, a JMangler implementation of a test coverage solution (taken from [30]) is dissected in class. As a homework assignment, students are asked to use JMangler to add basic hooks that capture information fed into a simple performance profiler.

```
public static void main(String args [])
{
  Method fooMain;
  Class fooClass;
  Object newFooInst;

  // generate a .java file at run-time by writing to a
  // FileWriter object
  generateCode("Foo.java");

  // compile the generated code
  Runtime.getRuntime().exec("javac Foo.java");

  // create and start a Foo instance
  fooClass = Class.forName("Foo");
  fooMain = fooClass.getMethod("main", null);
  newFooInst = fooClass.newInstance();
  fooMain.invoke(newFooInst, null);

  // add the Foo object to the object browser
  ObjBrowser.add(fooMain);
}
```

Listing 2.2. Dynamic code generation and reflective object invocation.

2.4. A compile-time MOP for Java

The need for open implementations and open languages [24], in conjunction with the weak form of introspection offered by standard Java, has prompted researches to investigate extensions to Java in order to support structural reflection and behavioral intercession. Such capabilities allow the addition of new methods and fields to a class structure, as well as a facility for trapping method calls to provide adaptation to behavior not defined in a "plain old Java object" (POJO). A representative example of this kind of Java extension is OpenJava, which is a compile-time MOP [53]. Although JMangler and JavaAssist operate on Java class files, OpenJava is purely a source to source translator.

OpenJava parses a Java source file and provides meta-objects for classes, methods, and fields. By overriding the default behavior of the associated meta-object, structural and behavioral adaptations can be made to the parsed code. After parsing and attaching meta-objects, OpenJava then invokes the translation methods on each meta-object. The process of meta-object translation generates modified source that can then be compiled by the traditional Java compiler to produce bytecode. By introducing OpenJava into the course, students are exposed to a more powerful form of reflection in Java that also has the capability of introducing language extensions.

As a homework exercise, students can be asked to add adaptations that write tracing information to the screen upon each entry and exit of a method. An outline of the solution

```
public class AddEntryExit instantiates MetaClass extends OJClass {

  public void translateDefinition() throws MOPException {

    OJMethod[] methods = getDeclaredMethods();
    String methodSig;
    String printString = "java.lang.System.out.println(" ;

    for (int i = 0; i < methods.length; ++i) {
      methodSig = methods[i].toString();
      Statement before = makeStatement(
            printString + \"" + methodSig + " entered\" );" );
      Statement after = makeStatement(
            printString + \"" + methodSig + + " exited\" );");
      methods[i].getBody().insertElementAt(before, 0);
      methods[i].getBody().add(after);
    }
  }
}
```

Listing 2.3. An OpenJava transformation rule for adding method entry/exit traces.

to this problem is shown in Listing 2.3, where an `AddEntryExit` meta-class is defined that inserts `println` statements at the beginning and end of a method. For each class to which the `AddEntryExit` meta-class is attached, the `translateDefinition` method will be called by OpenJava during meta-object transformation. After OpenJava has performed its translation, all classes that instantiate `AddEntryExit` will have the tracing behavior added to each method body.

3. Aspect-oriented software development

To support software adaptation and evolution, new paradigms such as aspect-oriented software development (AOSD) (http://aosd.net) have shown initial promise in assisting a developer in isolating points of variation and configurability. It has been observed that most programming languages provide modularization mechanisms that force other non-orthogonal concerns to be scattered and tangled across a code base [52]. Aspects are a new language construct for cleanly separating concerns that, heretofore, crosscut the modularization boundaries of an implementation [27]. In a fundamentally new way, aspects permit a software developer to quantify, from a single location, the effect of a concern across a body of code [14], thus improving overall modularization. A translator called a weaver is responsible for merging the separated aspects with the base code. After completing the reflection portion of the course, students are equipped to understand how AOSD mechanisms improve upon the capabilities provided by reflection alone.

3.1. Motivating examples

To motivate the benefits of AOSD, students must see several examples of poor modularity that exist in real software. These examples should be presented to the students before any technical solutions are proposed. An appeal can be made that these examples violate fundamental principles of software engineering (e.g., cohesion and coupling) and that traditional languages lack supporting constructs for coping with such problems. For instance, one of the earliest studies that documented the benefits of AOSD was presented in [36]. In this study, an application composed of almost 50K lines of Java source code was refactored using aspects. The results that were reported are substantial with a reduction in the amount of exception handling code by a factor of 4. Further redundancy was removed from the code by aspectizing common pre- and post-conditions that were spread across the code base. It was reported that 56% of the 375 post-conditions were redundant, and 1510 simple pre-conditions were reduced to just 10 pre-conditions using aspects. As another case study, the benefits of aspectizing four concerns within the FreeBSD operating system can be introduced to students [9]. For a commercial example, four example aspects from a client-server application are documented in [17].

3.2. AspectJ: Java-based tool support for AOSD

After the motivating examples are presented, the students should be able to understand the foundational principles of aspect-orientation [39]. Fortunately, almost all of the major research efforts in AOSD have produced tools that support the integration of the research ideas with Java. Perhaps the most mature of the AOSD tools is AspectJ [26] (see http://www.eclipse.org/aspectj/ for download information). In this course, students are given several weeks of lectures on AspectJ along with a plentiful selection of exercises during each lecture. As an aid for preparing lectures on AspectJ, a large collection of presentations are available at the original AspectJ site at Xerox PARC (please see http://www.parc.com/research/csl/projects/aspectj/default.html).

The primary concepts of AspectJ should be introduced first, such as joinpoints, advice, and introductions. A joinpoint represents a specific location in the execution of a Java application. A pointcut is a declarative specification that contains a collection of joinpoints. A pointcut typically defines the various locations where a concern appears. The complimentary notion of advice prescribes the behavior that is to be associated with a particular pointcut. As the lecture sequence progresses, students are eventually exposed to all of the syntax and semantics of AspectJ.

The homework assignment for this part of the course should emphasize the power of AOSD approaches over the techniques that were introduced in the previous section on reflection. To further emphasize the benefits of AOSD, students can be asked to perform adaptations on medium-sized open-source projects. JHotDraw is an excellent choice for such an assignment. The JHotDraw project is a Java framework for constructing graphical editing tools (see http://jhotdraw.sourceforge.net/). As an easy first exercise, students can be asked to consider the tracing problem that was previously implemented in OpenJava in Listing 2.3. Each student can also be asked to compare the two different solutions, and how each solution is integrated into JHotDraw to trace all method entry and exits.

```
aspect AddEntryExit {

  pointcut allMethodCalls(): call(* *.*(..));

  before(): allMethodCalls() {
     System.out.println(thisJoinPoint + "entered");
  }

  after(): allMethodCalls() {
     System.out.println(thisJoinPoint + "exited");
  }

}
```

Listing 3.1. AspectJ code to add method entry/exit logs.

As shown in Listing 3.1, the AspectJ solution is very concise. In fact, it is quite elegant when compared to the corresponding solution in Listing 2.3. The single pointcut declaratively specifies all calls to every method (note that the asterisks represent wild card designations within the signature for the return type, class name, and method name). The `before` advice specifies that the name of the method signature (designated by thisJoinPoint) is to be printed before the method is called. The `after` advice is similarly represented, but the affect is rendered after the method call returns. As comparisons are made between the OpenJava and AspectJ solutions, students report that the AspectJ implementation is much more comprehensible. This observation was also found among professional software developers [44]. Additionally, the AspectJ solution requires no manual changes to the base code. For the OpenJava solution, it is necessary to tag each class declaration with "`instantiates AddEntryExit`" to associate the meta-class that performs the adaptation. For a system with hundreds of classes, the OpenJava solution is rendered impractical, but for the AspectJ solution the weaver will include the tracing behavior without any manual addition to base code. As a more lengthy homework exercise, students can be asked to search JHotDraw for code that could be aspectized and provide the required separation into an aspect. They may also be asked to consider the ramifications that aspects have on refactored code [21].

3.3. Other approaches to AOSD

Unfortunately, this section only offers a shallow introduction to the important AOSD topics that are covered in the actual course. Apart from the multiple lectures on AspectJ, a single lecture is devoted to each of the other main approaches to AOSD, such as Hyper/J (now a part of the Concern Manipulation Environment) [41,52], DemeterJ [35], and ComposeJ [4]. Each lecture considers the examples posed in the original papers. For comparative purposes, the idea of a global tracing concern for method entry/exit is implemented in each approach. This gives a single frame of reference for understanding commonalities and differences among the syntax and semantics of each technique.

The load-time adaptation tools presented in Section 2.3 are actually considered techniques for supporting dynamic weaving. During the AOSD section of the course, the concepts of dynamic weaving are revisited, and another technique is described from [43].

Several researchers have also identified the advantages of language-independent aspect weaving, which brings the benefits of AOSD to languages other than Java. In [31], a language-independent approach for C# is presented. The technique weaves concerns into the common language infrastructure (CLI) of .Net. Another language-independent approach, which is also platform independent, is described in [17]. This approach uses a program transformation system, called the design maintenance system (DMS) [3], to build an aspect weaver for Object Pascal.

3.4. AOSD across the lifecycle

Proper separation of concerns is beneficial at all levels of the software life-cycle [19]. Although the idea is not specific to Java, students are asked to think about the benefits that AOSD would bring to approaches of analysis and design. To motivate example research in this area, the work of Clarke and Walker is introduced to the students as a technique for improving modularization of properties in UML class diagrams [8]. Correspondingly, constraints in models of embedded systems can be separated as a type of higher-level aspect [19]. The examination of AOSD at the design and modeling levels provides a great transition into the third section of the course, which is focused on meta-modeling tools and program synthesis from model interpreters.

4. Model-driven synthesis of Java applications

From a modeling perspective, expressive power in software specification is often gained from using notations and abstractions that are aligned with the problem domain. In domain-specific modeling [18], a design engineer describes a system by constructing a visual model using the terminology and concepts from a specific domain. Analysis can be performed on the model, or the model can be synthesized into an implementation [40]. Model-integrated computing (MIC) has been refined over many years to assist in the creation and synthesis of complex computer-based systems [23,50]. A key application area for MIC is those systems that have a tight integration between the computational structure of a system and its physical configuration (e.g., embedded systems) [49]. In such systems, MIC has been shown to be a powerful tool for providing adaptability in changing environments [51]. The generic modeling environment (GME) [32] is a meta-configurable modeling tool for realizing the principles of MIC. The GME provides meta-modeling capabilities that can be configured and adapted from meta-level specifications (representing the modeling paradigm) that describe the domain.

In this section of the course, students apply the concepts of reflection and meta-ideas to higher levels of abstraction, as represented by meta-modeling tools. The students learn to design the meta-model for a domain using the GME. In the GME, the meta-model is specified using UML class diagrams and OCL constraints. The OCL constraints are actually executed within the GME modeling engine to ensure the domain rules are consistently applied. In addition to the introduction of meta-modeling principles,

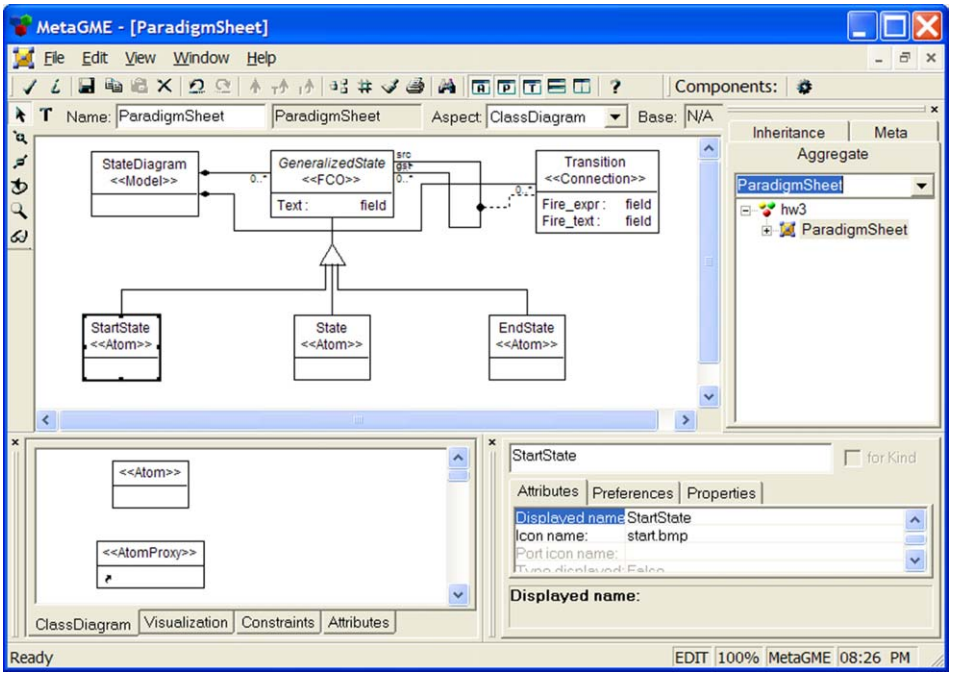

Fig. 4.1. Simple meta-model for a finite-state machine.

students are instructed in the techniques of generative programming [10] and asked to develop a Java code generator from the domain models. To illustrate the concepts that are taught, a student solution to an actual homework problem is presented here (the complete example is available from the course web site).

4.1. An exercise in meta-modeling

Fig. 4.1 illustrates a screen-shot of the GME showing a simple meta-model for a finite-state machine (FSM) that was submitted by a student in response to a homework problem. In this meta-model, a `StateDiagram` is specified as containing `GeneralizedStates` and `Transitions`. The `GeneralizedState` entity is a generalization of the three types of state that may appear in a FSM (i.e., start states, end states, and intermediate states). Each state has a text attribute that represents the string that is to be printed to the screen upon entry of the state. `Transitions` are modeled as connections between `GeneralizedStates`. Each transition has two attributes representing the conditional expression causing the transition to fire, and the text string that is to be displayed when a transition is enabled. As can be seen from the bottom right of the figure, visualization attributes can also be associated with each modeling entity (e.g., the `StartState` icon will be rendered from the “start.bmp” graphic file).

In addition to the class diagram from Fig. 4.1, a meta-model also contains constraints that are enforced whenever a domain model is created as an instance of the meta-model.

Constraints in the GME are specified in a different context from that shown above, but the following is an example constraint specification:

```
parts ("StartState") -> forAll (x | x.connectedFCOs ("dst") ->
                                size()=1) and
parts ("State") -> forAll (x | x.connectedFCOs ("dst") ->
                           forAll (y | y.kindName <> "StartState"))
```

The above constraint captures the idea that "There can only be one transition leaving the start state and no transitions coming into the start state". The first line of the above constraint specifies that all `Transitions` coming out of a `StartState` must only have one destination. The second and third lines of the constraint state that, for all of the transitions coming out of a `State`, none of them can be connected to a `StartState`. The constraint is executed each time a new transition is added to a domain model.

4.2. Creating domain models as instances of the meta-model

After creating the meta-model for the FSM, students can then asked to create a domain model that is based upon the FSM meta-model. As an example, Fig. 4.2 shows a domain model for an Automated Teller Machine (ATM). In the bottom right of this screenshot, the attributes of the transition connecting the `CardInserted` and `ValidUser` states are shown. The fire expression here reveals that this simple example has a hardcoded identification code of "777". That is, the machine enters the `ValidUser` state only when the user enters the correct identification code. As the transition is fired, the associated text string that is to be displayed is "Valid User".

4.3. Model interpreters that generate Java applications

The final part of the student meta-modeling project is to create a model interpreter that generates Java code from the FSM models. In the GME, a model interpreter is a type of plug-in that is associated with a particular meta-model and can be invoked simply by pressing a single button (the single button is represented by the italicized "C" on the "Components" toolbar in Fig. 4.2). To write an interpreter, the GME provides an API for accessing the internal structure of the model. Interpreters can be compiled as Windows DLLs and registered to the GME. From this API, an interpreter walks the tree representation of the model and generates code at each node. Thus, for the FSM meta-model, Java code is generated that simulates the execution of an FSM. The FSM interpreter assumes that the user will enter input from the keyboard in response to the text string that is displayed from a transition firing. From this input, the interpreter generates code that determines which transition is to be fired next. Note that the code generator assumes that transition firing expressions consistently reference a variable named "input" (see the fire expression in Fig. 4.2).

Listing 4.1 shows Java code that was generated from a student-written FSM interpreter. This particular piece of generated code represents the `CardInserted` state. The first half of the method displays the Text string of the `CardInserted` state (e.g., "Please Enter Pin") and receives the user input. The last half of the method determines which transition out of

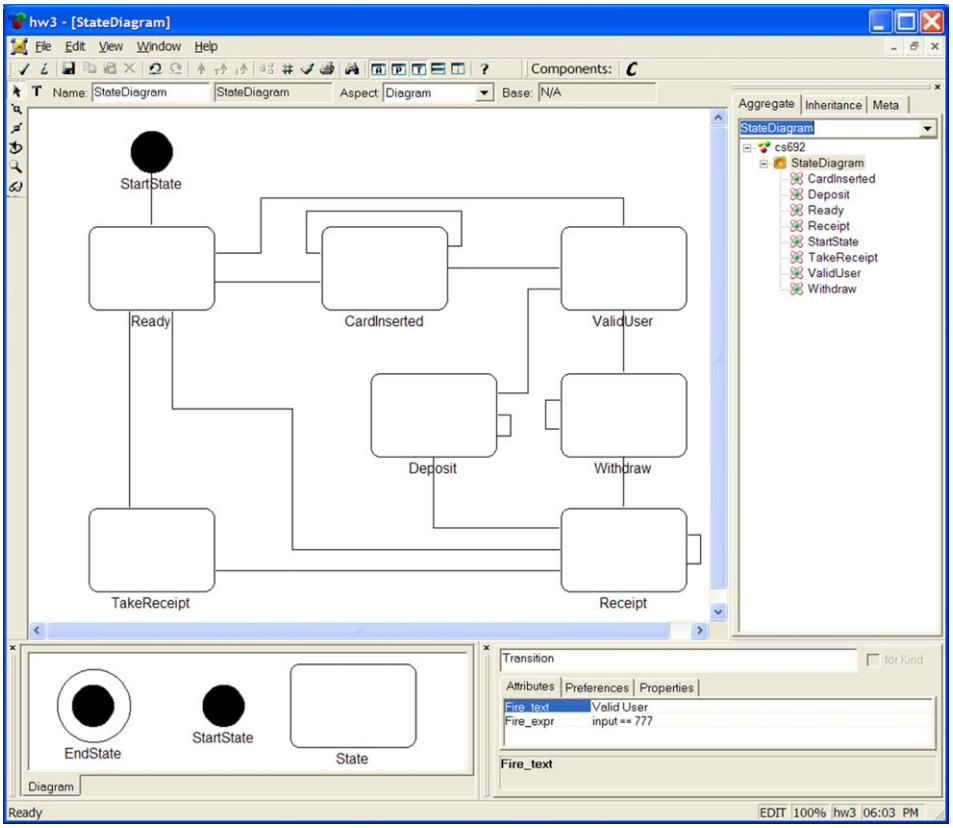

Fig. 4.2. An instance of the meta-model of Fig. 4.1 that represents an ATM.

the state is to be fired. Other code is generated similarly for each state and transition that is specified in the model.

Model-based approaches can improve the ability to adapt to changing requirements [49]. Small modifications to a model often result in multiple changes to the lower-level representation. Students learn the benefits of model-integrated techniques by understanding that meta-ideas and reflection can also be applied to higher levels of abstraction. It is also possible to combine concepts from AOSD and modeling as a way to separate modeling properties that are crosscutting [19].

5. Adaptive, reflective, and aspect-oriented middleware

The final topic area that is covered in the course sequence represents a practical examination of adaptive techniques applied to distributed computing. Middleware researchers are increasingly looking toward reflective and AOSD solutions to better modularize middleware implementations [56,57]. For instance, a major goal of Zen,

```
static void CardInserted()
{
  int input = 0;
  try
  {
   BufferedReader reader;
   String inputStr;
   reader = new BufferedReader(new InputStreamReader(System.in));
   System.out.println("Please Enter Pin");
   inputStr = reader.readLine();
   input = new Integer(inputStr).intValue();
  }
  catch (Exception e) {  }
  if(input != 777)
  {
    System.out.println("Invalid Pin");
    CardInserted();
  }
  else if(input == 777)
  {
    System.out.println("Valid User");
    ValidUser();
  }
}
```

Listing 4.1. Sample output from student-written Java code generator.

an open-source CORBA ORB implemented in Real-Time Java, is "to eliminate common sources of overhead and non-determinism in middleware implementations, as well as to be space-efficient to meet the needs of embedded applications" [28]. Perhaps the best evidence of the impact that AOSD approaches are having on middleware research can be found in the full embrace of such techniques by one of the most popular open-source Java application servers—JBoss [54].

The remainder of this section introduces a case study that is presented to the students, as well as issues surrounding model-driven configuration of middleware.

5.1. Quality of service adaptation in distributed objects

The ability to adapt is an essential trait for distributed object computing (DOC) middleware solutions. In real-time embedded systems, the presence of quality of service (QoS) requirements demands that a system be able to adjust in a timely manner to changes imposed from the external environment [46]. Students should understand that, to provide adaptability within distributed real-time systems, there are three things that must be present: (1) the ability to express QoS requirements in some form, (2) a mechanism to monitor important conditions that are associated with the environment, and (3) a causal

relation between the monitoring of the environment and the specification of the QoS requirements in such a way that there is a noticeable change in the behavior of the system as it adapts [22].

As a case study for adaptive middleware, students learn about a prototype application for an unmanned aerial vehicle (UAV). A UAV is an aircraft that performs surveillance over dangerous terrain and hostile territories. The UAV streams video back to a central distributor that forwards the video on to several different displays [37]. In the presence of changing conditions in the environment, the fidelity of the video stream must be maintained according to specified QoS parameters. The video must not be stale, or be affected by jittering, to the point that the operator cannot make an informed decision. Within the UAV implementation, a *contract* assists the system developer in specifying QoS requirements that are expected by a client and provided by a supplier. Each contract describes operating regions and actions that are to be taken when QoS measurements change. A domain-specific language exists to assist in the specification of contracts; the name of this DSL is the contract description language (CDL) [13]. A generator translates the CDL into code that is integrated within the runtime kernel of the application. As students consider the relationship of the CDL contracts to the dynamic modification of observable behavior, they can relate back to the earliest parts of the course regarding the essence of causal connection.

5.2. Model-driven middleware configuration

The object management group (OMG) has sponsored an initiative that is focused on modeling the differentiating characteristics of distributed applications. The model-driven architecture (MDA) provides the capability to separate the essence of an application (specified in platform-independent models) from specific bindings to middleware solutions (specified in platform-specific models) [5,6]. The final topic of the course is an instance of MDA that focuses on the generation of middleware components from MIC models [16]. The models of a real-time embedded system are presented to the students along with the mechanisms for adapting software at multiple levels of abstraction. Boeing's BoldStroke is a product-line architecture for mission computing avionics software that is implemented on top of real-time middleware [47]. The concept of two-level weaving is introduced to the students, such that properties woven into domain models trigger the generation of AspectJ code that configures a real-time event channel within BoldStroke [20].

6. Summary of course results

Within a year, the course described in this paper has become one of the most popular offerings in the computer science graduate curriculum at UAB. Toward the end of the AOSD section of the course, many students typically begin to formulate their own research questions to be investigated. As a result of the first year of the course, six Ph.D. students and two Masters students have chosen to explore dissertation and thesis topics in this area. As a transition to their own research, several students published a paper that compared the ideas presented in the course (e.g., a comparative survey found that experienced Java

developers prefer AspectJ over OpenJava in terms of comprehensibility [44]). This section gives an overview of the research that has resulted from this course.

6.1. Two-level aspect weaving

In system modeling, constraints may be specified throughout the nodes of a model to stipulate design criteria and limit design alternatives. A lack of support for separation of concerns with respect to constraints can pose a difficulty when creating domain-specific models [19]. The scattering of constraints throughout various levels of a model makes it hard to maintain and reason about their effects and purpose. In conventional system modeling tools, any change to the intention of a global property requires visiting and modifying each constraint, for every context, representing the property. This requires the modeler to "drill down" (i.e., traverse the hierarchy by recursively opening, with the mouse, each sub-model), manually, to many locations of the model. It is common for system models to contain thousands of different modeling elements with hierarchies that are ten or more levels deep.

To provide better support for exploring design alternatives in the presence of crosscutting model properties, an aspect-oriented approach to modeling has been investigated. The C-SAW weaver framework serves as a generalized transformation engine for manipulating models. C-SAW is a plug-in for the GME. When C-SAW is invoked from the GME toolbar, the user is asked to provide a set of files that specify modeling aspects that describe the location and behavior of the transformation to be performed on the model. The result of model weaving is a new model that contains adaptations that are spread across the model hierarchy. These adaptations can be undone and new concerns can be woven from simply selecting different model aspects. The concept of a model weaver can be used in many ways beyond the application of constraints. For example, a weaver can be used to distribute any system property endemic to a specific domain across the hierarchy of a model. A weaver can also be used to instrument structural changes within the model according to the dictates of some higher-level requirement that represents a crosscutting concern.

The concept of aspect model weaving, when combined with the idea of model-driven program transformation, provides a powerful technology for rapidly transforming legacy systems from high-level properties described in models. The goal is to have small changes at the modeling level trigger very large transformations at the source level. This can be achieved by applying aspect-oriented techniques and program transformation concepts. At one level, model transformations allow alternative design configurations to be explored using an aspect weaver targeted for modeling tools. From generative programming techniques, the models can be used to generate program transformation rules to adapt legacy source on a wide scale. The initial description of C-SAW is given in [20]. The C-SAW web site (http://www.gray-area.org/Research/C-SAW/) contains the plug-in for GME, as well as various video demos illustrating the approach.

6.2. Language-independent aspect weavers

Much of the research in AOSD has been concentrated on Java-based tools, as described in Section 3. Yet, the vast majority of legacy systems are written in languages other

than Java. To apply AOSD principles to legacy systems, initial work has been performed to harness the power of a commercial program transformation engine (PTE) in order to construct aspect weavers for other languages. Commercial PTEs typically provide mature lexers and parsers for several dozen programming languages. Additionally, a PTE offers the ability to transform a parse tree through rewrite rules. Thus, the core features that are needed by an aspect weaver exist in a PTE. The preliminary work on this research idea can be found in [17]. A web site with video demonstrations of the aspect weavers that were constructed from this approach can be found at http://www.gray-area.org/Research/GenAWeave.

6.3. Broader impacts

In addition to graduate research, two Honors undergraduate students at UAB conducted research into model-based synthesis of Java software to control multi-agent robots [11]. Using the GME, a meta-model represents a hostile environment containing land mines, rescue targets, and robots. The model interpreter for this project generates Java code that will control multiple LEGO robots in a rescue mission. This work is sponsored by a fellowship from the Computing Research Association (CRA) special Collaborative Research Experience for Women (CREW). The project deliverables are available at the UAB CREW web site (http://www.gray-area.org/Research/CREW).

The introduction of this course also prompted the formation of a new conference mini-track related to the course topics (please see http://www.cis.uab.edu/HICSS-AESS/), which is now in its second year.

7. Conclusion

Software developers continue to face serious challenges in the presence of changing stakeholder requirements, which require facilities for effectively evolving software artifacts. Moreover, the majority of the total global computational cycles today are spent on controlling real-time and embedded systems, including cell phones, automobile engines and brakes, chemical factories, and avionics applications. In fact, it has been reported that over 90% of all of the world's microprocessors are used in systems that are not "traditional" computers [45]. Physical mechanical controls are being replaced every day by software controllers [33] that must adapt to varying environmental conditions. The reliance on these new devices has increased the quality of our lives in many ways, yet also has created a critical dependence on technology. Therefore, additions to computer science curricula are needed so that future software developers are exposed to techniques that support software evolution. This paper presented an overview of a course that embraces the concept of software adaptation by using Java-based technologies as a platform for study.

A single foundational language from which to explore new concepts can be beneficial toward student comprehension and understandability. By fixing Java as the core throughout the course, students are not forced to relearn other languages for each new concept that is presented. Correspondingly, much of the interesting research in the area of adaptive software is being conducted within the context of Java. These factors made the selection of

Java an obvious and powerful choice for this course. Other languages, such as C#, do not provide the breadth across the spectrum of topics considered in this course.

In the initial offerings of this course, students were assigned research papers that were representative of the topics being studied. These papers typically came from software engineering journals and conference proceedings. In particular, the October 2001 issue of the *Communications of the ACM* featured a special issue on AOSD. Similarly, a special issue on adaptive middleware was published in the June 2002 issue of *Communications of the ACM*. In the future, a book that surveys AOSD will replace many of the papers found in Section 3 [15]. A suggested schedule for introducing the four research areas can be found at http://www.cis.uab.edu/cs622/Fall2003/schedule.htm.

References

[1] U. Aßmann, Invasive Software Composition, Springer-Verlag, 2003.
[2] D. Batory, J.N. Sarvela, A. Rauschmeyer, Scaling step-wise refinement, in: International Conference on Software Engineering, Portland, Oregon, May, 2003, pp. 187–197.
[3] I. Baxter, C. Pidgeon, M. Mehlich, DMS: program transformation for practical scalable software evolution, in: International Conference on Software Engineering, ICSE, Edinburgh, Scotland, May, 2004.
[4] L. Bergmans, M. Aksit, Composing crosscutting concerns using composition filters, Communications of the ACM (2001) 51–57.
[5] J. Bézivin, From object composition to model transformation with the MDA, in: Technology of Object-Oriented Languages and Systems, TOOLS, Santa Barbara, CA, August, 2001, pp. 350–354.
[6] C. Burt, B. Bryant, R. Raje, A. Olson, M. Auguston, Quality of service issues related to transforming platform independent models to platform specific models, in: The 6th International Enterprise Distributed Object Computing Conference, EDOC, Switzerland, September, 2002, pp. 212–223.
[7] S. Chiba, Load-time structural reflection in Java, in: European Conference on Object-Oriented Programming, ECOOP, Cannes, France, June, LNCS, vol. 1850, Springer-Verlag, 2000, pp. 313–336.
[8] S. Clarke, R.J. Walker, Composition patterns: an approach to designing reusable aspects, in: International Conference on Software Engineering, ICSE, Toronto, Ontario, Canada, May, 2001, pp. 5–14.
[9] Y. Coady, G. Kiczales, Back to the future: a retroactive study of aspect evolution in operating system code, in: International Conference on Aspect-Oriented Software Development, Boston, MA, March, 2003, pp. 50–59.
[10] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
[11] R. Dennison, B. Shah, J. Gray, A model-driven approach for generating embedded robot navigation control software, in: 42nd Annual ACM SE Conference, Huntsville, Alabama, April 2–3, 2004.
[12] E.W. Dijkstra, Notes on structured programming: on program families, in: Structured Programming, Academic Press, London, 1972, pp. 39–41.
[13] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, J. Zinky, Building adaptive distributed applications with middleware and aspects, in: International Conference on Aspect-Oriented Software Development, AOSD, Lancaster, UK, March 22–27, 2004, pp. 66–73.
[14] R. Filman, D. Friedman, Aspect-oriented programming is quantification and obliviousness, in: OOPSLA Workshop on Advanced Separation of Concerns, Minneapolis, MN, October, 2000.
[15] R. Filman, T. Elrad, M. Aksit, S. Clarke (Eds.), Aspect-Oriented Software Development, Addison-Wesley, 2004.
[16] A. Gokhale, D. Schmidt, B. Natarajan, J. Gray, N. Wang, Model-driven middleware, in: Q. Mahmoud (Ed.), Middleware for Communications, John Wiley and Sons, 2004.
[17] J. Gray, S. Roychoudhury, A technique for constructing aspect weavers using a program transformation system, in: International Conference on Aspect-Oriented Software Development, AOSD, Lancaster, UK, March 22–27, 2004, pp. 36–45.
[18] J. Gray, J.-P. Tolvanen, M. Rossi (Guest Eds.), Domain-Specific Modeling with Visual Languages, Journal of Visual Languages and Computing (2004) (special issue).

[19] J. Gray, T. Bapty, S. Neema, J. Tuck, Handling crosscutting constraints in domain-specific modeling, Communications of the ACM (2001) 87–93.
[20] J. Gray, J. Sztipanovits, D.C. Schmidt, T. Bapty, S. Neema, A. Gokhale, Two-level aspect weaving to support evolution of model-driven synthesis, in: R. Filman, T. Elrad, M. Aksit, S. Clarke (Eds.), Aspect-Oriented Software Development, Addison-Wesley, 2004.
[21] S. Hanenberg, C. Oberschulte, R. Unland, Refactoring of aspect-oriented software, in: Net.ObjectDays 2003, Erfurt, Germany, September 22–25, 2003.
[22] D. Karr, C. Rodrigues, J. Loyall, R. Schantz, Y. Krishnamurthy, I. Pyarali, D. Schmidt, Application of the QuO quality-of-service framework to a distributed video application, in: International Symposium on Distributed Objects and Applications, Rome, Italy, September, 2001, pp. 299–309.
[23] G. Karsai, A configurable visual programming environment: a tool for domain-specific programming, IEEE Computer (1995) 36–44.
[24] G. Kiczales, Beyond the black box: open implementation, IEEE Software (1996) 8–11.
[25] G. Kiczales, J.M. Ashley, L. Rodriguez, A. Vahdat, D.G. Bobrow, Metaobject protocols: why we want them and what else can they do? in: A. Paepcke (Ed.), Object-Oriented Programming: The CLOS Perspective, 1993, pp. 101–118.
[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, Getting started with AspectJ, Communications of the ACM (2001) 59–65.
[27] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: European Conference on Object-Oriented Programming, ECOOP, Jyväskylä, Finland, June, LNCS, vol. 1241, Springer-Verlag, 1997, pp. 220–242.
[28] R. Klefstad, S. Rao, D.C. Schmidt, Design and performance of a dynamically configurable, messaging protocols framework for real-time CORBA, in: 36th Hawaii International Conference on System Sciences, Big Island, Hawaii, January, 2003.
[29] G. Kniesel, P. Costanza, M. Austermann, JMangler—a framework for load-time transformation of java class files, in: IEEE Workshop on Source Code Analysis and Manipulation, SCAM, November, 2001.
[30] G. Kniesel, P. Costanza, M. Austermann, JMangler—a powerful back-end for aspect-oriented programming, in: R. Filman, T. Elrad, M. Aksit, S. Clarke (Eds.), Aspect-Oriented Software Development, Addison-Wesley, 2004.
[31] D. Lafferty, V. Cahill, Language-independent aspect-oriented programming, in: Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Anaheim, CA, October, 2003, pp. 1–12.
[32] Á. Lédeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai, Composing domain-specific design environments, IEEE Computer (2001) 44–51.
[33] E. Lee, What's ahead for embedded software? IEEE Computer (2000) 18–26.
[34] S. Liang, G. Bracha, Dynamic class loading in the Java virtual machine, in: Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Vancouver, B.C., Canada, October, 1998, pp. 36–44.
[35] K. Lieberherr, D. Orleans, J. Ovlinger, Aspect-oriented programming with adaptive methods, Communications of the ACM (2001) 39–41.
[36] M. Lippert, C.V. Lopes, A study on exception detection and handling using aspect-oriented programming, in: International Conference on Software Engineering, ICSE, Limmerick, Ireland, June, 2000, pp. 418–427.
[37] J. Loyall, R. Schantz, M. Atighetchi, P. Pal, Packaging quality of service control behaviors for reuse, in: 5th IEEE International Symposium on Object-Oriented Real-time Distributed Computing, ISORC, Washington, DC, April 29–May 1, 2002, pp. 375–385.
[38] P. Maes, Concepts and experiments in computational reflection, in: Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Orlando, FL, December, 1987, pp. 147–155.
[39] H. Masuhara, G. Kiczales, Modeling crosscutting in aspect-oriented mechanisms, in: European Conference on Object-Oriented Programming, ECOOP, Darmstadt, Germany, July, 2003, pp. 2–28.
[40] G. Nordstrom, J. Sztipanovits, G. Karsai, Á. Lédeczi, Metamodeling—rapid design and evolution of domain-specific modeling environments, in: International Conference on Engineering of Computer-Based Systems, ECBS, Nashville, Tennessee, April, 1999, pp. 68–74.
[41] H. Ossher, P. Tarr, Using multidimensional separation of concerns to (re)shape evolving software, Communications of the ACM (2001) 43–50.
[42] D. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM (1972) 1053–1058.

[43] A. Popovici, G. Alonso, T. Gross, Just-in-time aspects: efficient dynamic weaving for Java, in: International Conference on Aspect-Oriented Software Development, Boston, MA, March, 2003, pp. 100–109.
[44] S. Roychoudhury, J. Gray, H. Wu, J. Zhang, Y. Lin, A comparative analysis of meta-programming and aspect-orientation, in: 41st Annual ACM SE Conference, Savannah, Georgia, March 7–8, 2003, pp. 196–201.
[45] B. Santo, Embedded battle royale, IEEE Spectrum (2001) 36–42.
[46] D. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, L. DiPalma, Toward adaptive and reflective middleware for network-centric combat systems, Crosstalk: The Journal of Defense Software Engineering (2001) 10–16.
[47] D. Sharp, Reducing avionics software cost through component based product-line development, in: Software Technology Conference, Salt Lake City, Utah, April, 1998.
[48] B.C. Smith, Reflection and semantics in lisp, in: Annual Symposium on Principles of Programming Languages, Salt Lake City, Utah, 1984, pp. 23–35.
[49] J. Sztipanovits, Generative programming for embedded systems, in: Keynote Address: Generative Programming and Component Engineering, GPCE, Pittsburgh, PA, October, 2002, LNCS, vol. 2487, Springer-Verlag, 2002, pp. 32–49.
[50] J. Sztipanovits, G. Karsai, Model-integrated computing, IEEE Computer (1997) 10–12.
[51] J. Sztipanovits, G. Karsai, T. Bapty, Self-adaptive software for signal processing, Comunications of the ACM (1998) 66–73.
[52] P. Tarr, H. Ossher, W. Harrison, S. Sutton, N degrees of separation: multi-dimensional separation of concerns, in: International Conference on Software Engineering, ICSE, Los Angeles, CA, May, 1999, pp. 107–119.
[53] M. Tatsubori, S. Chiba, M.-O. Killijian, K. Itano, OpenJava: a class-based macro system for Java, in: W. Cazzola, R.J. Stroud, F. Tisato (Eds.), Reflection and Software Engineering, LNCS, vol. 1826, Springer-Verlag, 2000, pp. 117–133.
[54] E. Tilevich, S. Urbanski, Y. Smaragdakis, M. Fleury, Aspectizing server-side distribution, in: IEEE International Conference on Automated Software Engineering, Montreal, Canada, October, 2003.
[55] N. Wang, D.C. Schmidt, O. Othman, K. Parameswaran, Evaluating meta-programming mechanisms for ORB middleware, in: B. Opdyke, A. Pakstas (Eds.), IEEE Communication Magazine, Evolving Communications Software: Techniques and Technologies, October, 2001, pp. 102–113 (special issue).
[56] E. Wohlstadter, S. Jackson, P.T. Devanbu, DADO: enhancing middleware to support crosscutting features in distributed, heterogeneous systems, in: International Conference on Software Engineering, Portland, Oregon, May, 2003, pp. 174–186.
[57] C. Zhang, H.-A. Jacobsen, Quantifying aspects in middleware platforms, in: Proceedings of the International Conference on Aspect-Oriented Software Development, Boston, MA, March, 2003, pp. 130–139.