Mapping UML Designs to Java™

William Harrison Charles Barton Mukund Raghavachari

IBM T.J. Watson Research Center

PO Box 704

Yorktown Heights, NY 10598

harrisn@watson.ibm.com

ABSTRACT

Tools for the generation of code from model descriptions are valuable in helping developers maintain consistency between a model and its implementation. In this paper, we describe a new method for generating Java implementation code from UML diagrams. Our method accepts UML diagrams specified at a higher-level than current tools, and places fewer constraints on the supported UML constructs, like multiple generalizations and association classes. Unlike current tools, it generates implementation code that shields system implementers from the low-level details of how associations and other UML constructs are mapped to Java. Furthermore, it supports the modular design of systems according to concerns[2, 10] by being able to generate code from a set of related UML diagrams. While our discussion is focused on the special problem of generating Java implementation code, the issues discussed in this paper are applicable more generally to object-oriented implementation languages.

Keywords

UML, Java, code generation, design, separation of concerns

1. INTRODUCTION

In order to better design and manage the development of complex software systems, software designers have turned increasingly to modeling languages such as the Unified Modeling Language (UML)[1]. Relationships between the components of a system are grasped more easily when the design is represented graphically using a modeling language. UML modeling tools generally support the generation of skeletal implementation code either directly or by exporting models in a standardized format, such as XMI[25], that can be used by third-party tools. Tools for the generation of code from model descriptions are valuable in helping developers maintain consistency between a model and its implementation, which may involve a large number of source files compared to size of the model.

Developers use modeling languages for various kinds of designs, ranging from providing a description of the implementation of the system to modeling the architecture of the system independent of implementation language or style choices. In the design process, there is always a tension as to the appropriate level at which the system should be modeled so as to attain maximum flexibility, quality, and maintainability of the system, and continuity and cross-reference between the design and its implementation.

When the model represents the design of a specific implementation, the correspondence between the design and the implementation code is apparent and is managed more easily. Several current UML modeling environments provide tools that can generate skeletal code from implementation designs [16, 17, 18, 22, 23, 27, 28]. The system designer, however, is constrained to using a subset or an interpretation of the modeling language that fits the implementation's constraints, whether language, configuration or system model. The design is shaped by the choice of implementation language and style, often obscuring characteristics that are intrinsic to the system/problem being modeled. For example, in most abstract models of systems of any complexity, some entities of the system fit many different generalizations. At the level of an implementation in Java, however, it is difficult to represent these multiple inheritance relationships, since Java restricts multiple inheritance to interfaces and requires single inheritance for their implementations. To model the implementation accurately, the modeler would normally accept the restriction to single inheritance in the design, a restriction that most existing code generation tools impose.

A model of a system expressed independently of a specific implementation provides a clearer and more flexible description of the system's architecture and the relationships between its components. It is clearer because design issues and decisions are not confused with implementation-specific ones and more flexible because it can be mapped more easily to fit many different implementation constraints. On the other hand, the task of validating a particular implementation against the design model is harder, because the constructs at the two levels are no longer necessarily in close correspondence. The problem of designing with higher-level models is further aggravated by the fact that few tools exist for generating appropriate skeletal implementation code from high-level design code.

In this paper, we study the problem of generating object-oriented language implementation code from high-level designs. We present a mapping method that preserves expressive freedom for the designer by allowing the specification of abstract models that make few assumptions about the underlying implementation. Our mapping generates a high-level skeletal implementation that shields implementers from low-level representation choices, thereby facilitating further system development. For example, we present an abstraction, *cursors*, for simplifying the navigation

OOPSLA '00, 10/00 Minneapolis, MN, USA
© 2000 ACM ISBN 1-58113-200-x/00/0010...$5.00

and management of associations. UML, one of the most popular design languages, is the concrete basis of our discussion, although the issues we discuss arise with any design language. While our mapping method is applicable to a broad set of implementation languages, we focus on its particular application in generating Java implementation code. We have implemented a tool based on our mapping method for generating Java code from abstract UML designs, and have used this tool in the design of SAGE (Scalable Adapter Generator), a component of the Message Central project at the IBM T.J. Watson Research Center[21].

An issue with modeling complex systems is that it becomes difficult to view and understand an entire design as the number of design elements and features grow. A solution to this problem is to partition the design according to concerns[2, 10], where each partition represents the slice of the system corresponding to a concern of the system (for example, one concern may be the user interface, another security). Using concerns, the design of a system consists of several partitions, one for each concern or set of concerns. The overall design of the system is obtained by composing the various sub-designs. Our mapping method can handle system designs expressed as a set of UML designs, where each sub-design corresponds to a small number of concerns.

The paper is organized as follows. In Section 2, we discuss the objectives that guided the design of our mapping from UML. In Section 3, we discuss the UML design elements we support, and present our basic mapping method for generating OO code from UML. In Section 4, we discuss issues related to the application of our mapping method to Java. We review previous approaches to the model-to-implementation code generation problem, and highlight the novelties in our approach in Section 5. Finally, in Section 6, we summarize our study.

2. OBJECTIVES IN MAPPING HIGH-LEVEL UML DESIGNS

We imagine a system being designed with respect to a variety of particular concerns[2]. When a developer is constrained to use a single design to model all concerns of a system together (for example, design elements related to user interface, security, etc., are modeled together in a monolithic design), the elements of the design often become scattered and tangled together, with unfortunate consequences[10]. To alleviate this problem, we consider a design process in which designs corresponding to each of the individual concerns may be specified independently, and then, composed to derive the design of the entire system.

The roles involved in this design process are as follows. The ***system designer*** specifies the partition of the design into concern designs and how they are to be composed. The ***concern designer*** specifies the entities that effect a particular concern, their attributes, operations, and relationships to other kinds of entities. The attributes, associations, and operations of each entity characterize how that entity can be used by the designers and implementers of other entities in the same concern and of other concerns. The ***object designer*** specifies additional attributes and relationships, and even operations, that are to be visible only within implementations of the entity. The ***behavior implementer*** may specify additional private state information and operations as well as the algorithms for implementing the operations, but should assume no particular implementation of attributes or association. The latter is the province of the ***representation implementer,*** who specifies how attributes and associations are represented to satisfy various other constraints. The work of the concern and object designers is expressed in UML diagrams. The work of the behavior and representation implementers is expressed in the target object-oriented language.

Our goal in the generation of skeletal implementation code from design specifications is to facilitate the design and development process by:

1. Allowing designers the flexibility to model systems with few assumptions about the underlying implementation.

2. Providing designers with a high-level, consistent framework based on the design that supports the use and extension of the system without exposing the implementation of its components.

3. Providing implementers with a framework in which the implementation of each component of the design can be carried out independently of the implementation of the other components.

4. Minimizing designer and implementer effort.

These goals are achieved partly by ensuring that well-known software engineering principles are followed in the generated code and by mapping to a sufficiently high-level implementation design. In this section, we discuss the specific objectives we set for the translation process, and how they enable us to attain the aims set above.

Objective 1: Separate Design from Implementation

Software systems generally undergo change and improvement in response to experience with their implementations and the needs of the customers of the systems. To ease the jobs of developers, the code generated from a design should use a programming style that isolates the various design elements from each other's implementation decisions. Although the implementation of an element may depend on the design of other elements, it should not depend on their implementation. An implementer of a particular element or a designer of objects or concerns can use the elements of the design independently of their implementation.

Objective 2: Separate Behavior from Representation

Even within the implementation of an object, we would like to minimize the effect of data representation decisions made with respect to the storage of attributes or associations. The code that embodies the behavior of the objects should not need to be changed when the data's representation are changed or extended to address various performance or other environmental characteristics. The performance cost of this decoupling should be nominal and within acceptable bounds.

Objective 3: Maximize Type-Safety

Strong type enforcement is an important tool for detecting errors early in the development process. It is important to ensure that

code generated by our mapping retains type information to assist compilers with type-safety checks. Loss of type information not only results in error-prone code, but also results in the investment of considerable programming effort in writing explicit type-casts. Our objective in the mapping process is to preserve strong type checking and avoid forcing system implementers to insert type-cast code when filling in the implementation of the generated skeletal code.

Objective 4: Avoid Round-Tripping

Environments that support generation of code from higher-level descriptions must address the situation that arises when modifications are made to a base of code that was generated from the higher-level description. If the high-level design is then changed, the code must be re-generated from the design. Changes or additions made to the generated code that are unrelated to the higher-level changes need to survive the *round-trip* through the regeneration process. Ideally, the modifications are made not in the generated code, but directly in the high-level model, ensuring that the code generator can mirror changes easily. Realizing this ideal generally requires support of a full software engineering environment. Even so, the modifications are often not expressible in the high-level model. When full control of the generated source is not possible, a common solution to the round-trip problem is to generate code that contains "markers" that delimit regions in which additions and changes may be made, which are then reproduced in the re-generated code. Our objective is to achieve a similar effect in a more robust manner by using language mechanisms to separate the components that contain modifications from those produced by the code generator.

Objective 5: Support for Designing with Concerns

Partitioning a large UML design into separate diagrams makes the overall design easier to understand and possesses the virtue of modularity. The system designer assigns the modeling of relatively separate features or aspects of the design to different UML diagrams, which can be elaborated independently. The composite diagram is defined as the diagram obtained by identifying same-named elements from the separate UML diagrams and merging their features. Our goal is to be able to accept a set of related UML diagrams and generate the code that would have resulted from the composite diagram.

Objective 6: Support for Generalizations (Supporting Multiple Inheritance)

A generalization is a relationship between entities in a design indicating that instances of one kind of entity can be viewed as instances of the associated generalization as well. To model complex systems accurately, it is often necessary to allow components to have more than one generalization. A natural way of representing generalizations in an object-oriented language is through the use of inheritance relationships. Programming languages, however, often impose constraints on the use of inheritance between classes. For maximum flexibility, it is important to shield designers from these constraints. For example, in mapping to a language such as Java, the lack of multiple inheritance among classes ought not to hinder a designer from designing components with multiple generalizations.

Objective 7: Support for Associations

We wish associations to support the following capabilities (they mirror the annotations on associations allowed by UML), and allow our mapping to include them in a smooth manner.

1. Directionality and Roles – An association can be navigated in either direction, or in both. In addition, role names can be attached at the ends (one or both) of the association. A name appearing at one end describes the role played in the association by the entity at that end.

2. Multiplicity – Each end of an association can have an annotation for multiplicity. This constrains the number of entities that may be found by starting from an instance of the entity at the other end and traversing the association in the direction specified by that role. We will support the "to-1" and "to-many" annotations. There are times, like during construction, deletion, or movement of elements, when the multiplicity constraints must be violated in the course of carrying out the operation, but the constraints should be valid at major semantic operational points.

3. Association Classes – In general, associations between entities may have more semantics than specifying that the entities are linked. For example, in representing the marriage between two individuals as a **Marriage** association between **Person** entities, the marriage date and location are more appropriately stored on the association, and not with either of the two **Person**s, especially considering that the same two persons may have had more than one marriage. To this end, an association may have a class linked to it that provides information and behavior relevant to the association. An instance of the association class is attached to a link of the association. In the marriage example, the instance might hold the date of the marriage that the link represents.

4. Subtyping Associations – Assume the model contains entities **A** and **B** connected by a relationship with a role **r** at the **B** end. Assume further that **C** and **D** are entities that are "subtypes", that is, **A** is a generalization of **C**, and **B** generalizes **D**. We want to allow the modeler to require that an instance of **C** may only be linked, in this association, to an instance of **D**, rather than to an arbitrary instance of **B**. The modeler will specify this by drawing a relationship between **C** and **D** with the same-named role **r** at the **D** end.

5. Stereotypes – Stereotypes can be defined for associations that imply a difference in the semantics of the association.

Objective 8: Behavioral Access for Associations and Attributes

Object implementers must write code to access the associations and attributes they manipulate. Rather than force implementers to express code in terms of the data representation of the associations or attributes in their object, it is useful to separate behavior from representation by defining high-level interfaces for manipulating attributes and associations. A simple get/set interface suffices for attributes. For associations, the implementations of life-cycle behaviors like creation, deletion, navigation, etc., should be generated automatically and be

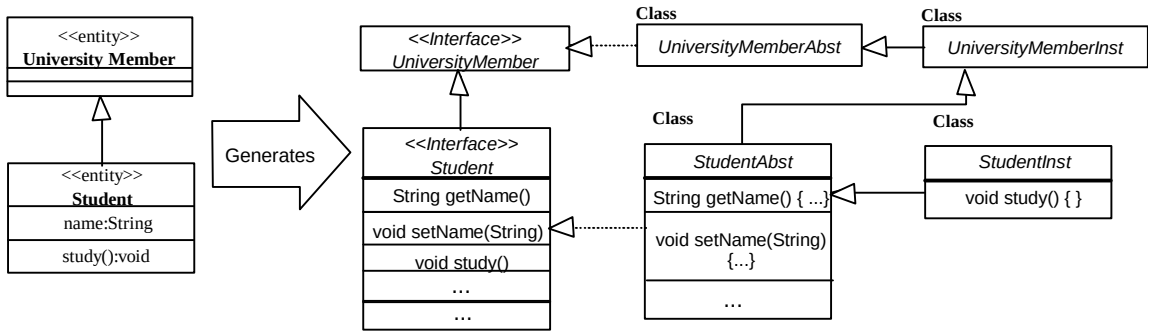

Figure 1. Class hierarchy generated from a UML class diagram.

transparent to the implementers.

Objective 9: Unified Set of Idioms for Accessing/Navigating the Model

It is important that the interfaces to the generated code support common idioms for accessing various components of the model. For example, if an attribute of an entity is realized as a Java variable that is given the same name as the attribute's name in the model, all attributes of all entities should be mapped in a similar way. If these idioms are specified clearly, a system implementer can deduce easily from the model how to navigate and access its various components. A good set of idioms facilitates code development and promotes reuse and extensions of code. For example, a routine written to follow 1-to-many associations should work if reused on 1-to-1 associations. Arbitrary divergence of the idioms, for example, rules like: "associations with multiplicity of three or more will be named by spelling their role name backward," only serve to retard the development process.

Objective 10: Promote Code Reuse

Constructs that are required in the generated code may be similar in semantics to existing types/packages in the implementation language or libraries. To ease the task of an implementer, it would be useful to express generated constructs in terms of these types so that the implementer could use other packages based on the existing types where possible. For example, in Java, expressing lists of elements in terms of *Enumeration* could allow for the use of other third-party routines that expect an *Enumeration*.

3. METHOD FOR MAPPING UML DESIGNS TO OO CODE

In this section, we discuss the basic approach for generating object-oriented code from UML class diagrams to satisfy the objectives we have described.

3.1 UML Class Diagrams

The UML constructs that we support are classes marked with a stereotype **Entity**, with their attributes and operations, generalization relationships, and association relationships with their various adornments including roles, multiplicities, aggregations, and association classes. We recognize that high-level class designs may coexist with designs geared towards a specific implementation in a system design. Classes not marked as belonging to the stereotype **Entity** can be treated as implementation-specific designs and mapped as such. In the remainder of this paper, the term "entity" is used to mean a UML class with the stereotype **Entity**. We consider a collection of UML diagrams as a partitioned representation of a single composite diagram. Entities in different diagrams that share the same name are taken to represent the same entity and their features are merged, also by name matching[2].

3.2 Entities

Each entity is mapped into an interface and a pair of implementing classes. For each entity, **X**, we generate an interface named *X* and classes named *XAbst* (for abstract) and *XInst* (for instantiable). The interface contains operations defined for the entity in the UML model, as well as auxiliary operations for accessing its attributes and associations, and for performing object construction. The *XAbst* class is an abstract class that contains implementations of the auxiliary operations mentioned above, as well as the physical representation of attributes. The *XInst* class extends *XAbst*, and is a skeletal class that provides placeholders that can be used to fill in the implementations of the operations defined for **X**. These two classes sever the generated implementation from the written extensions by playing the roles of the core and extension in the "Generation Gap" pattern[14]. It is neither necessary nor permissible for the behavior implementer to modify the interface *X* or the class *Xabst* generated from an entity. All code written by a behavior implementer is encapsulated in the *XInst* class that does need not be regenerated for unrelated model changes, thus obviating round-tripping, as Objective 4 requires.

To separate behaviors from representation (Objective 2), all accesses to attributes of an entity in the code written by behavior implementers are through auxiliary functions. For each attribute, **attr**, of an entity, **X**, there are two operations in the corresponding *X* interface, *getAttr()* and *setAttr()*. The implementations for these routines are in the *XAbst* generated abstract class as described above. The attributes themselves are declared in the *XAbst* class, and are hidden from other classes in the system. To maximize type-safety (Objective 3), the auxiliary operations use the type of the attribute specified in the UML design.

Consider the example in Figure 1 where an entity, **University**

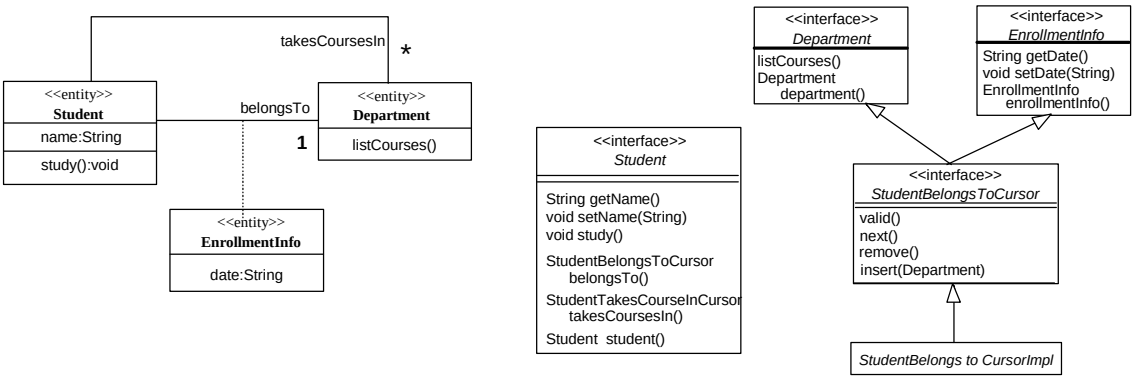

Figure 2. Example of generated code for associations.

Member, is a generalization of another entity, **Student**. The mapping creates the *Student* interface, and *StudentAbst* and *StudentInst* classes. In addition to the **study** operation of **Student**, the *Student* interface contains auxiliary get/set operations to access the **name** attribute, which are implemented in *StudentAbst*. A skeleton of the implementation of **study** is found in *StudentInst*. The figure omits other generated constructs and auxiliary operations that relate to associations, which are discussed in the next section.

As can be seen from the figure, the generalization hierarchy of a UML class diagram is mirrored in the generated implementation. In our mapping, the interface of an entity extends the interfaces of its generalizations. Furthermore, each implementation of the operations of an entity (in the *XInst* class) extends the implementations of its generalizations indirectly through the *XAbst* class. This was a conscious decision to ensure maximum code reuse of implementations, and thus, minimize development effort. We discuss a specific issue that arose as a result of this decision in mapping multiple generalizations to a language with single inheritance such as Java in Section 4.3.2. The alternative mapping of allowing only an *XAbst* class to inherit from the *XAbst* class of its generalizations would have presented different problems, and would have resulted in developers expending much more effort in the common case.

3.3 Associations

As it does with attributes, our mapping method separates the semantics of associations from their representation and implementation. To accomplish this separation, we introduce an abstraction called a *cursor*, which encapsulates the complexity of navigating and updating the association, and hides its implementation. Consider the example in Figure 2, in which we show the interface generated from the **Student** entity, and the cursor corresponding to the **belongsTo** relationship (the figure omits details of the code generated for the other elements of the class diagram). For each of the two named roles accessible from **Student**, its corresponding interface contains auxiliary methods for retrieving the cursors for those roles. The implementation of these methods would be in the *StudentAbst* class, which is not shown. The figure also shows the *StudentBelongsToCursor* interface and its associated implementation For each named role, like **belongsTo**, we generate an interface and an implementation named with the entity and role, in this case, *StudentBelongsToCursor* and *StudentBelongsToCursorImpl*. An association access operation for retrieving a *StudentBelongsToCursor* is included in the interface *Student*. When invoked on an instance of **Student**, it returns a cursor giving access to the collection of **belongsTo** links from that instance.

As suggested by the figure, a cursor has methods for inserting and removing links and for iterating over the set of links. The cursor, when in a valid state, is said to "point to" or to "refer to" the instance at the other end of the current link, which we call the "current target". The cursor is used as a reference to the current target and so we specify that *StudentBelongsToCursor* extends the interface **Department** and delegates the calls on **Department**'s operations to the current target. If there is an association class defined, we similarly simplify manipulation of its instances by requiring that *StudentBelongsToCursor* extend the interface of the association class *EnrollmentInfo*, and by delegating its methods to the current association class instance.

In the following sections, we describe the interface to and the implementation of cursors in greater detail.

3.3.1 Association Management Operations

When a cursor is first constructed by invoking a cursor accessor on an entity, it is set to contain the "first" target entity as its "current target." Note that the iteration order is not specified at this level. If no links have been defined yet, the cursor is set to be *invalid*. All cursors support the following four operations: *valid()*, *next()*, *remove()*, and *insert(t)*. The semantics of these operations is the same for all cursors, subject to multiplicity constraints on the association, to satisfy Objective 9. Cursor interfaces consist of the following methods:

valid	If *valid()* is true, the cursor is said to be valid and has a reference to an instance of the target entity of the association. If the cursor is invalid, attempts to access the target will result in an exception.
next	The *next()* operation advances the cursor so that the "next" target becomes current. If there is none, the cursor becomes invalid. In the case of to-1 relationships, the cursor becomes invalid by definition.
remove	The *remove()* operation removes the current element. If the cursor is not valid, it throws an exception. Prior to the removal, *remove*

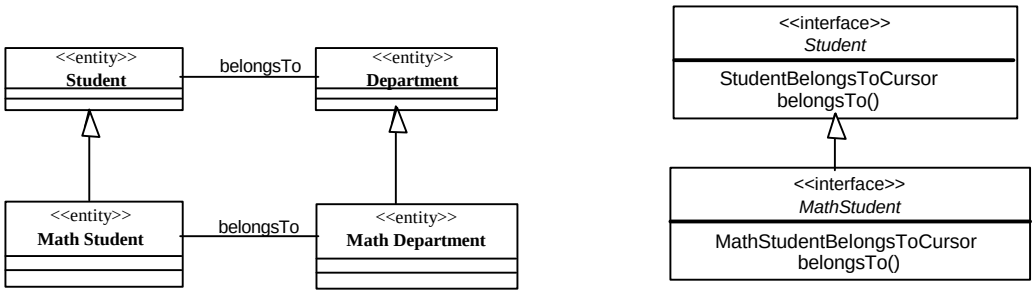

Figure 3. An occurrence of subtyping in a model.

advances the cursor as specified for *next()*.

insert(t) — The *insert(t)* operation inserts a link to the specified target element *t*. The parameter *t* has the type of the interface corresponding to the target of the relationship. In the case of a to-one association, *insert* first performs an implicit *remove()*. After the insert, the current target is *t* and the cursor is valid.

3.3.2 Target Operations

Our cursor abstraction, designed to fulfill Objectives 8 and 9, is closer in spirit to Eiffel's Iterators[8] than to Java's Enumerations. Compared to a solution based on Enumerations, it results in simpler and more transparent code for the common operation of traversing a 1-1 relationship to perform an operation on its target. To achieve this goal, a cursor extends the interface defined by the target entity. These operations are implemented in the cursor by delegating them to the current target. This is in partial realization of Objective 8, the simplification of the behavior of associations. Implementers have available a variety of expressions. For example, if *x* supports the *Student* interface, the *listCourses()* method of the *Department* reached by following *belongsTo* from *x* can be invoked by *x.belongsTo().listCourses()*, or by:

```
StudentBelongsToCursor p = x.belongsTo();
p.listCourses();
```

Iterative processing of a to-many association is handled similarly:

```
for (StudentTakesCoursesInCursor p = x.takesCoursesIn(); p.valid();
p.next()) {p.listCourses();}
```

Since cursors extend the interface of the entity to which they refer, one can use them as one would use references. They are, however, more volatile than ordinary references because a cursor could be advanced to its next target without software using it being aware of the fact that its point of reference is changing. To make available a stable reference to the current target, we add to the interface for each entity **X** a "self-reference" operation *X x()* which returns a reference to itself. When invoked through a cursor, this method will return a reference to the entity that is the current target of the cursor. To maximize type-safety (Objective 3), the returned reference is of the appropriate type.

3.3.3 Association Class Operations

When an association has an association class, the cursor supports an additional operation, *insert(TargetType t, AssociationType a)*, where *TargetType* is the type of entity appropriate for the role's target and *AssociationType* is the type of the interface corresponding to the entity of the association class.

insert(t,a) — *insert(t)* inserts a link to the specified target element, *t*, as a new target of the association with a specified association class instance *a*. In the case of a to-one association, *insert* first performs an implicit *remove()*. After *insert*, the cursor points to *t*.

Furthermore, the cursor produced for the association extends the interface of the association entity, as it does for the target entity. When both the target entity and the association entity have attributes or operations of the same name, it is ambiguous as to which entity the cursor should delegate when that operation or attribute is accessed through the cursor interface. We take the approach of dropping ambiguous operations from a cursor interface, and forcing implementers to retrieve explicitly the references to the target or association entity using the "self-reference" operation explained previously when invoking those operations.

3.4 Stereotyped Association Implementations

A discussion of the specific details of the implementations of cursors and associations is beyond the scope of this paper. We note that we support stereotyped associations, where the stereotype is used to determine how cursors are implemented, though the interface for using the association remains the same. This helps us to support associations without compromising Objectives 1 and 2, that is, the separation of design from implementation and representation. We have also found it to be of great use for specifying associations that have extended behavior and for providing implementations that reuse a variety of pre-existing specialized implementations of associations.

4. SPECIAL ISSUES FOR JAVA

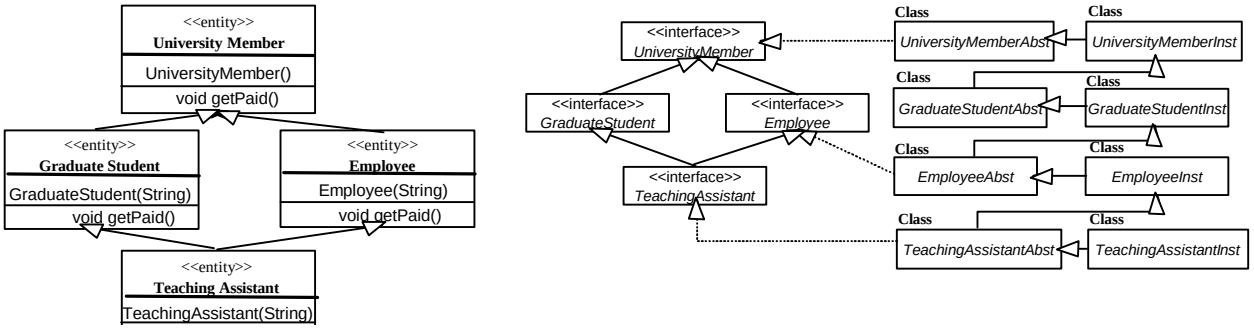

Figure 4. Resulting class hierarchy from mapping multiple generalizations to Java.

IMPLEMENTATIONS

Our tool converts UML class diagrams expressed in XMI into Java classes following the methodology stated in the previous section. In this section, we discuss some Java-specific issues that were addressed during the implementation of our UML-to-Java tool.

4.1 Cursors as Enumerations

Cursors behave much like the classes satisfying the interface *java.util.Enumeration*. To maximize code reuse (Objective 10), *Cursor* interfaces also extend that interface, allowing them to be passed to pre-existing code expecting enumerations of objects. *Cursor* interfaces, therefore, also support the following operations:

hasMoreElements	is the same as *valid()*
nextElement	performs a *next()* but returns the target of the cursor that was current when *nextElement()* was called.

4.2 Subtyping

Figure 3 illustrates a configuration that can arise in a model and that can cause difficulty in translation to Java. It shows four entities: **Student**, **Math Student**, **Department**, and **Math Department**, linked by an association role **belongsTo**. A **Math Student** is a kind of **Student**, and **Math Department** is a kind of **Department**. A **Student** belongs to a **Department**, and a **Math Student** belongs to a **Math Department**.

Using the mapping discussed for associations to generate Java code would result in an illegal Java program because the *belongsTo()* operation has the same signature but different return types (*MathStudentBelongsToCursor* and *StudentBelongsToCursor*) in the *Student* and *MathStudent* interfaces. *MathStudentBelongsToCursor* ought to enforce the type constraints implied by the model by "extending" *StudentBelongsToCursor* in two ways: it should extend the interface *MathDepartment* instead of just *Department*, and its insert method should require parameters of type *MathDepartment* and not merely of type *Department.* To achieve this goal while observing Java's restrictions, we do the following:

1. The same return type, *StudentBelongsToCursor*, is used for the *belongsTo()* methods in both the *Student* and *MathStudent* interfaces.

2. We broaden the definition of *StudentBelongsToCursor* to extend the interface *MathDepartment*.

3. In the class *MathStudentAbst*, we override *StudentAbst*'s implementation of *belongsTo()* to return a different implementation of *StudentBelongsToCursor*. The instance of *StudentBelongsToCursor* returned to a *MathStudent* checks in its *insert* method that the parameter is actually of class *MathDepartment*. If a *Student* instance is not of class *MathStudent,* the returned *StudentBelongsToCursor* traps any invocations of *MathDepartment* methods that are not defined in *Deptartment*. These differing implementations provide dynamically the type safety that we cannot obtain statically from Java.

4.3 Reducing Multiple Inheritance to Single

In order to support generalizations, we need to map multiple generalizations of an entity to interfaces and implementation classes in the Java realization. Since Java allows an interface to extend many other interfaces, that part of the mapping is natural. The Java interface for an entity is defined to extend the Java interfaces of all of its generalizations. However, although code inheritance is very important for reuse, Java does not support multiple inheritance of implementation classes. To enable implementation inheritance, we produce an inheritance tree for the generated implementations by linearizing the inheritance hierarchy. Consider the example in Figure 4. The multiple generalization hierarchy shown can be mirrored exactly among the interfaces corresponding to the entities. For the implementation classes, however, we generate the classes as though **Graduate Student** had been a generalization of **Employee** in the UML class diagram. In general, we convert an inheritance hierarchy that is a directed, acyclic graph into a tree, and ensure that the classes generated for any entity inherit from all its generalizations in the original UML class diagram. A side-effect of this approach is that spurious subclassing relationships are introduced among the classes (for example, between *EmployeeAbst* and *GradStudentAbst*). We now address two issues, construction and proper choice of method implementations, that arise when mapping multiple generalizations to Java.

4.3.1 Construction and Initialization

We require a construction model that is different from Java's because our generated implementation code must allow independent superclasses corresponding to multiple generalizations of an entity to participate in the initialization of the classes corresponding to that entity. Since, as described in the previous section, we simulate multiple generalizations using single inheritance, we need a construction model in which the construction of an instance of an entity results in the individual constructors for all classes corresponding to generalizations of that entity ***in the UML model*** being called, but not those of other Java classes that happen to become superclasses in the generated implementation. Our construction model, however, ought to be as consistent as possible with Java's model.

In this section, we describe how the construction specified in a UML design is mapped to Java. Like Java's built-in model, our construction model follows the builder pattern[4] for object creation. Creation involves four parts: the factory, the call for construction, the control of the construction sequence, and the individual constructors. These parts all have analogies in Java's construction model. In Java, the factory is built in and control of the construction plan is tailored for single inheritance. As in Java, each entity may have several creation methods, each of which has a different signature. In Figure 5, we depict the sequence of method calls made to perform the initialization of the **Employee** entity in the multiple generalization example of Figure 4. The creation operations for an entity, **X**, are specified in the UML model as operations of the entity, all named **X**. For example, in Figure 4, the entity **Employee** has a single constructor that takes a **String** as an argument.

4.3.1.1 Factory

There are various strategies that could be adopted by a client to locate the factory for creating a new object. The simplest method employs a single known static variable or method for obtaining a reference to a single factory. More complex strategies are needed when more than one factory might be in use, the most complex of which involve parameterized factory-finding functions. We employ a middle-course and specify that every entity implements a method, declared as *ModelFactory modelFactory()*, that returns a reference to an appropriate factory. Each factory supports the creation of *all* entities.

4.3.1.2 Call For Construction

The methods identified in the *ModelFactory* interface are named *newX*, where *X* takes on the names of the entities specified in the UML model. An implementer, wishing to create an instance of an *Employee*, whose construction requires a *String* parameter specifying the name of the employee, using a reference *foo*, to an instance of an arbitrary entity, would write:

```
foo.modelFactory().newEmployee("Lou G.");
```

This would retrieve a model factory from the entity identified by *foo*, and invoke the factory's *newEmployee* method. This method uses the native Java *new* to create an entity (e.g., *new EmployeeInst("Lou G.");*), and then, begins the initialization of the new *Employee* instance as we now describe.

4.3.1.3 Control of the Construction Sequence

In Java, the order in which the constructors of a class and its superclasses are invoked is specified clearly and is straightforward because it has a single-inheritance model that makes the ordering of individual constructors simple. As a result, the *construction plan* is implicit and built into the language. In the conversion of UML designs to Java, we need to ensure that during the creation of an instance of an entity, the initializers of that entity and the appropriate superclasses (those corresponding to entities that are generalizations of the entity being created) are called in the appropriate order, and that no initializer is called more than once.

In our mapping, each entity contains two kinds of methods for construction, construction plans and individual constructors (described in the next section). A construction plan controls the sequence of initializations, by individual constructors, of the parts of the object that belong to its own class and to its superclasses. For an entity **X** and for each constructor with a different signature in **X**, an operation *XConstructPlan* is added to the corresponding *XAbst* class, and an implementation is generated in the *XAbst* class. After the factory's *newX* uses Java *new* to create an instance of the entity, it calls the version of *XConstructPlan* that has the same signature as the factory's method and passes on its arguments. That *XConstructPlan* passes these arguments to the individual constructors it calls. If no version of *XConstructPlan* has a matching signature, one with an empty signature is used.

4.3.1.4 The Individual Constructors

These methods are analogous to ordinary Java constructors except that they do not invoke, implicitly or explicitly, individual constructors for their superclasses. The generated individual constructor for an entity *X* is called *constructX* and is added to the *XAbst* class. There is one version of *constructX* for each overloading of *newX*. *constructX* may assume that *XConstructPlan* has already called the individual constructors corresponding to the generalizations of **X**. The generated implementation of *constructX* initializes the representations for attributes and associations. If the instantiable class introduces additional state variables or performs more complex initialization, it can override the generated individual contructor, calling it via "super" to accomplish the model initialization for the entity.

Figure 5 shows the construction operations that would be generated for the multiple generalization hierarchy in Figure 4, and the steps taken in the construction of an instance of **Employee**. First, the generated implementation of *EmployeeConstructPlan()* is invoked (Step 1). This routine ensures that *constructUniversityMember()* and *constructEmployee(String)* are called in that order (skipping over *constructGraduateStudent(String)*).

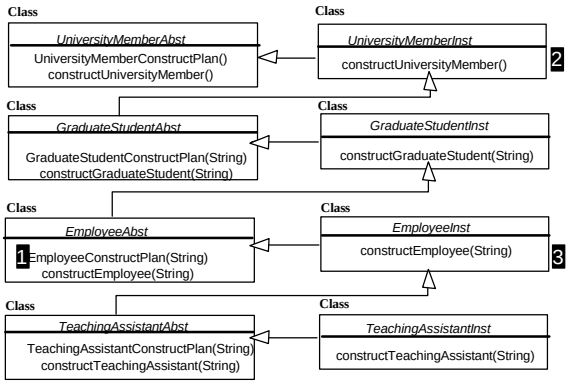

Figure 5. Construction code generated for the example in the previous figure.

4.3.2 Proper Choice of Method Implementations

The spurious subclassing that may be caused by our mapping of multiple generalizations to Java could result in undesired method inheritance. Consider the example in Figure 4, in which an implementation for *getPaid()* is provided in *UniversityMemberInst*. If this implementation is overridden in *GraduateStudentInst*, perhaps to reflect that a graduate student may be paid out of a research grant, *EmployeeInst* inherits this overridden method unwittingly, and not the desired one from *UniversityMemberInst*.

We solve this problem by exploiting the fact that in our UML-Java mapping, the instantiable (*Inst*) class always specifies the *Abst* class as its superclass. When necessary, we generate in the *Abst* class an override for methods of its *Inst* class. The override (in the *Abst* class) catches the call (from the *Inst* class) and redirects it to the appropriate superclass via a renamed diversion method. For example, observe in Figure 6 how *EmployeeInst*'s invocation of *getPaid*() reaches *UniversityMemberInst*'s implementation. These indirections are only necessary when we determine that spurious method inheritance may occur in the generated code. The appearance of a "diamond shape" in the model, as in Figure 4, signals the possibility of this problem.

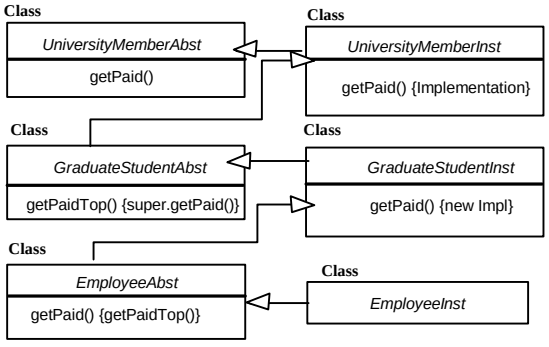

Figure 6. Avoiding spurious method inheritance.

5. RELATED WORK

As discussed in [6], one of the important issues for software engineering in the near term is better integration of the various concerns and artifacts used through the life cycle. The fact that the recent past has seen some convergence and more widespread use of design tools and the emergence of generators for producing object-oriented code signals that it is appropriate to focus attention not only on appropriate mappings, but on the concrete objectives by which to evaluate them. In this respect, the exercise here is similar to the valuable exposition of the design rationale for programming languages like C++ [12]. Some of the work described here addresses similar issues as other work on levels of abstraction, design composition, UML Code Generators, and operational models for relationships.

Levels of Abstraction

The Reference Model for Open Distributed Processing[20] supports the description of consistency relationships among several different viewpoints that together describe a body of complex software. Models like RM-ODP emphasize the need for tool support to aid in managing the separation and integration of the various concerns addressed in the different viewpoints, which are specified at varying levels of abstraction.

One approach to managing models at multiple levels of abstraction is to provide a Software Engineering Environment (SEE) that supports a refinement methodology for realizing high-level designs as lower-level ones, and for expressing how each level relates to the other. Catalysis[3] is a modern commercially available SEE of this sort, of which RPDE^3[9], Arcadia[13], and IDE[15] were earlier SEEs with similar goals.

Catalysis is an SEE of particular interest because of its support of UML. Unlike Catalysis, which permits a variety of lower level design elaborations for each construct in the higher-level design, we simplify the developer's task by restricting the form of the elaborations, and then, generating them from the higher-level design automatically. Instead of dealing with a pair of independently-created designs or implementations, we consider the more restricted situation in which there is a governing model into which design decisions at different levels all fit.

Design Compositors

Our UML-to-Java generator performs a UML design composition prior to generating the Java code for the design. Although the use of composition for program development is becoming better supported and more widespread [5, 6, 10], its impact on design is only beginning to be examined [2]. The information model underlying our work on SAGE was designed initially as a paper design, without anticipating subsequent automation of its conversion to code. Perhaps in consequence, we found that it uses "merge" composition more broadly than in [2], even including the merging of generalization relationships across diagrams. Tools like Rational/Rose [26] also provide design composition, although generally as an ancillary to version management rather than as a persistent feature of the design itself. Prior to undertaking the implementation of the compositor part of our generator, we investigated using these tools instead, but we found that they generally lacked several features we needed, including "by-name" composition of classes and merging

of the operations and relationships associated with the classes.

UML-to-Code Generators

The work described here differs from other code generators available for UML designs in that it is intended to map from high-level designs. The code generators in [16, 17, 18, 22, 23, 27, 28] all expect designs that are expressed in terms of Java constructs, including the mapping of attributes and associations into Java variables, etc. Taking this course would not have fulfilled many of the objectives discussed in Section 2.

Object Models for Associations

Object models for associations, such as the OMG Relationship Services[24], the java.util collection classes, and the IBM OpenClass library[19], are generally expressions of the iterator pattern [4]. The iterator pattern treats the iterator (or cursor) as a separate object from which the "current item" must be extracted and then processed. In our definition and use of cursors, we intentionally avoid exposing the implementation of associations. Our cursor model reduces the amount and unfamiliarity of client code, increases the robustness of the design under change, and permits us to extend the design in ways that use information not only about an object but about how it was reached. It does this by extending the iterator concept to allow the cursor to act as a transparent reference to the current item or to the related association class item.

6. SUMMARY

We have presented a new method for generating Java implementation code from UML designs. Our mapping method was designed to support sound software engineering principles, and possesses the following novel features:

It places fewer constraints on the designs it accepts compared to current tools, allowing the use of UML constructs such as multiple generalization and association classes.

It generates high-level code that shields developers from the details of how UML constructs, such as associations, are mapped to Java. The generated code supports a high-level implementation model that is faithful to the UML design.

It allows for modular design of systems according to concerns by supporting code generation from a set of related designs. The resulting code is that which would have been generated if a composition of the diagrams were presented to the tool.

The generation of Java code from UML designs was influenced by certain characteristics of the Java language. The lack of multiple inheritance of implementations of classes, and the inability to subtype, overload, or override return types of functions required extensions to our mapping model. A certain "redundancy" is present in our generated code as a result of our goal of supporting strong static type checking. This redundancy would not have resulted had Java supported parametric types.

7. REFERENCES

[1] Booch, G., J. Rumbaugh, and I. Jacobson, "The Unified Modeling Language User Guide," Addison Wesley (1999).

[2] Clarke, S., W. Harrison, H. Ossher, and P. Tarr, "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code," *Proceedings of the 1999 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver (November, 1999).

[3] D'Souza, D., and Wills, A. C., " Objects, Components, and Frameworks with UML – The Catalysis Approach," Addison-Wesley (1998).

[4] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, "Design Patterns Elements of Reusable Object-Oriented Software," Addison Wesley, p. 99ff (1995).

[5] Harrison, W., and H. Ossher, "Subject-Oriented Programming (a critique of pure objects)," Proceedings of the 1993 Conference on Object-Oriented Programming, Systems, Languages and Applications, Washington, D.C. (October, 1993).

[6] Harrison, W., H. Ossher, and P. Tarr, "Software Engineering Tools and Environments: A Roadmap," in "Future of Software Engineering," Anthony Finkelstein (Ed.), ACM Press (June, 2000).

[7] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, Springer-Verlag, LNCS 1241 (June 1997).

[8] Meyer, B., "Reusable Software: The Base Object-Oriented Component Libraries," Prentice Hall (1994).

[9] Ossher, H., and Harrison, W. "Support for change in RPDE^3," Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments (December 1990).

[10] Ossher, H., P. Tarr, W. Harrison, and S. Sutton, "N Degrees of Separation: Multi-Dimension Separation of Concerns," Proceedings of 1999 International Conference on Software Engineering (May 1999).

[11] Roth, W., " An Introduction To Enterprise Java Beans Technology," Java Developer Connection, http://developer.java.sun.com/developer/technicalArticles/Beans/IntroEJB/

[12] Stroustrup, B.J., The Design and Evolution of C++, Addison Wesley, Reading, MA.

[13] Taylor, R. N., Belz, F. C., Clarke, L.A., Osterweil, L. J., Selby, R. W., Wileden, J. C., Wolf, A. L., and Young, M., "Foundations for the Arcadia Environment Architecture," Proceedings of SIGSOFT'88: Third Symposium on Software Development Environments (November 1988).

[14] Vlissides, J., " Pattern Hatching – Design Patterns Applied," Addison-Wesley (1998).

[15] Wasserman, A. I., Pircher, P. A., Shewmake, D. T., and Kersten, M. L., "Developing Interactive Information Systems with the User Software Engineering Methodology," In *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 326-345 (February 1986).

[16] –, Aonix, Software through Pictures (MetaEdit+), http://www.metacase.com/meplus30index.html.

[17] –, Advanced Software Technologies, GDPRO: http://www.advancedsw.com/.

[18] –,AppBuilder, http://www.devdaily.com/AppBuilder/.

[19] –, S/390 V2R4.0 C/C++ IBM Open Class Library Reference, Document Number: SC09-2364-02, http://www.redbooks.ibm.com/cgi-bin/bookmgr/BOOKS/CBCOCR03/CCONTENTS.

[20] –, Reference Model of Open Distributed Processing, ISO/IEC Document 10746,

[21] –, MessageCentral Home Page, IBM T.J. Watson Research Center, http://www.research.ibm.com/messagecentral/.

[22] –,No Magic, Magicdraw, http://www.nomagic.com/magicdrawuml/features.htm.

[23] –,Object International Software, Together/J, http://www.togethersoft.com/together/togetherJ.html.

[24] –, OMG CORBA Services, Relationship Service, Version 1.0 , http://www.omg.org/technology/documents/formal/relationship_service.htm.

[25] –, OMG XMI Revised Submission, cgi.omg.org/cgi-bin/doc?ad/99-10-13

[26] –, Rational Rose '98, Using Rose, Rational Software Corporation (1998).

[27] –, Rational Software, Rational Rose, http://www.rational.com/products/rose/index.jtmpl.

[28] –, Softera, SoftModeler, http://www.softera.com/manual/UserGuide.htm.