A Pattern-based Framework to Address Abstraction, Reuse, and Cross-domain Aspects in Domain Specific Visual Languages*

David Oglesby, Kirk Schloegel, Devesh Bhatt, Eric Engstrom
Honeywell Laboratories
3660 Technology Drive, Minneapolis, MN 55418

August 19, 2001

Abstract
The history of software development has shown a trend towards higher levels of abstraction, code reuse and automatic code generation. This trend has continued with the development of Domain Specific Visual Languages (DSVL). We have developed a framework for the creation, use, and management of Patterns for DSVL environments that supports high levels of abstraction and high degrees of reuse and code generation, as well as multi-domain aspect modeling. We describe the power, flexibility, and use of these patterns and give an example of an implementation of a real-world design pattern under this framework.

1 Introduction
The history of software development has shown a trend towards higher levels of abstraction. Each level allows the developer to focus more directly on solving the problem at hand rather than implementation details. As software developers target more complex problems, it is important to provide better tools that support increasing levels of abstraction. Domain Specific Visual Languages (DSVLs) represent the developers' concerns at a different (usually higher) level of abstraction using concepts from the developers' own domains rather than modeling strictly in terms of software entities (e.g., UML modeling). Automatic code generators then translate domain-specific designs to the software domain. This approach has been demonstrated to be more efficient with respect to development time than designing directly with software entities [5].

Unfortunately, DSVLs are difficult and costly to build, usually requiring their own significant development effort. In addition, each domain supported by a tool is specific to a certain type of problem; thus each tool has a limited market. Metamodeling tools reduce the cost of creating domain specific tools by allowing a user to specify the syntax and semantics of a language in the form of a metamodel and by creating the supporting tools automatically. Thus metamodeling tools lower the barriers to developing domain specific

* This material is based upon work supported by the United States Air Force under Contract No. F33615-00-C-1705. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Air Force.

tools by significantly reducing the time and cost involved in their development and modification.

However, our experience with DSVLs and metamodeling tools suggests three areas for improvement.

1. DSVL environments (e.g. metamodeling tools) should support even higher levels of abstraction by providing a domain-independent capability to compose arbitrary high-level constructs within the domain-specific models from the low-level components. For example, the UML does not itself support hierarchical abstraction. However, we would like to be able to use hierarchical composition within UML models. Such hierarchical models could then be *flattened* into valid UML models prior to any UML-specific use.
2. DSVLs should support greater degrees of reuse within domains. Among existing metamodeling tools, Vanderbilt's GME supports metamodel reuse [4], and DOME's [7] archetypes support reuse within a model (see Section 1.2.1); but none of the metamodeling packages provide much support for reuse of model elements within and throughout modeling domains.
3. DSVL environments should provide support for modeling cross-domain aspects. Complex systems span multiple modeling domains and can contain crosscutting aspects. For example, embedded systems include software entities that consume a certain number of floating-point operations per call and hardware entities that provide a certain number of floating-point operations per second. Modeling this aspect (floating point operations) across both domains is necessary to analyze a system to ensure that the hardware can provide enough power for the software to meet performance requirements. We also discuss cross-domain modeling issues in depth in [6].

1.1 Patterns Provide a Mechanism for Abstraction, Reuse, and Multiple-Domain Compositions

We have developed a framework for the creation, use, and management of *patterns* for DSVL environments that addresses all of the challenges listed above. Patterns are defined in [2] as "solutions to problems in a context." Typical object-oriented patterns prescribe a class structure for solving recurring problems in the software domain. It is left to the developer to create the appropriate corresponding classes for his or her implementation. We are working to implement patterns as first-class objects in arbitrary modeling environments. This will allow developers to easily build up models by simply dropping existing parameterized patterns into domain-specific models and by creating new patterns to be used in other models. Automatic code generation will create the appropriate classes for the developer.

The higher level of abstraction that our pattern-support framework provides comes from the hierarchical composition of domain model elements and/or patterns. A pattern is a node on a graph that has internal details with which the user need not be concerned. Each pattern has one or more *implementations*, which are sets of one or more subdiagrams. A *subdiagram* is a hierarchical construct that allows the representation of a complicated portion of a graph as a simple node or arc. This node or arc then acts as a placeholder,

simplifying the graph for the user who may choose to view the subdiagram at his or her leisure. Each pattern instance uses only one implementation that is selected via parameters on the pattern.

Patterns also support domain-level reuse. A given pattern may be applied to multiple models within a domain (some can be applied in multiple domains as well). Instances of patterns may share one implementation, so changes to the implementation will affect all of those instances. In addition, since patterns can be included in the implementation of other patterns, it is easy to create new patterns by extending existing ones.

Patterns can model cross-domain aspects by defining multiple subdiagrams from different domains. Pattern instances sharing an implementation allow for aspect weaving. This provides a single point of concern for the aspect being modeled.

1.2 PREVIOUS WORK

In this section, we describe the previous work done in support of the challenge areas enumerated above.

1.2.1 DOME Support for DSVLs and Reuse

The Domain Modeling Environment [7] (DOME) from Honeywell Laboratories is a metamodeling tool for designing and using domain specific visual languages. Its specification language is graphical—using nodes and edges—and is supported by a special metamodeling editor. Once a visual syntax has been specified, DOME can interpret the specification on the fly. In this respect it is a metamodeling environment (hence the name). This makes it uniquely suited to cross-domain modeling because all models run within the DOME environment making it possible to regulate cross-model interactions.

DOME formalizes support for reuse with its *archetype* and *shelf* concepts. Archetypes support reuse of common structures or templates within any domain-specific visualization. The shelf provides access to the collections of available archetypes. It is a specialized model browser that provides a view of design graphs in terms of component archetypes. Archetypes are automatically created and managed by the component shelf, and archetypes can be reused by cloning instances of them into graphs from the shelf. The component shelf also automatically maintains traceability from archetypes to instances. The shelf and archetype features are the basis for our new expanded and generalized pattern-support tool.

DOME supports code generation and document generation libraries, accessed through its macro language, Alter. These simplify the task of generating deliverable artifacts directly from the model. This approach has the added benefit of increasing consistency throughout the product configuration.

While DOME's archetypes provide a convenient starting point, they fall short of patterns in several respects. They are specific to a model, so an archetype cannot be applied to different models even if they are in the same domain. Furthermore, they cannot be arbi

trarily parameterized to support context-sensitive design details. Similarly DOME's code generation capabilities will have to be extended. Currently, code generation is specific to a single modeling domain, but patterns will introduce cross-domain issues.

1.2.2 Pattern Modeling in the UML

Two UML concepts support pattern modeling: *mechanisms*, and *frameworks* [1]. A mechanism is a way to specify design patterns. It is a *collaboration* among classes that describes both structure (class diagram) and behavior (sequence diagram). A collaboration can be parameterized by the use of template classes as parameters. When a parameterized collaboration is instantiated, specific application classes in the instantiation context can be bound to the specific parameters of the collaboration. Thus, a collaboration can be used to specify additional relationships among application classes. This is also a severe limitation since only simple import-type bindings to classes are supported. Furthermore, inheritance from higher-level abstract classes and other properties of the application classes are often assumed in the collaboration class diagram but not explicitly specified. A *framework* is just a collection of collaborations, without any additional structural attributes.

The limited pattern specification capabilities in the UML allow the capture and automated instantiation of only simple design patterns. Our work presented in this paper overcomes many of these limitations. In particular, support is provided for capturing multiple domains of the design (e.g. class diagrams, data flow, hardware diagrams) in a single pattern. A pattern can be instantiated in different models and behaves according to the syntax and semantics of that model. Different types of entities inside a pattern can be parameterized and attached to pattern portals for different types of bindings in the instantiation context. Both import and export types of bindings are supported. Hence, a pattern can supply parameterized information to complete the instantiation context, and conversely, the instantiation context can supply parameterized information to complete the internal structure of a pattern. This allows for the specification of complex patterns that encompass multiple aspects of software structure and control/data flow.

1.2.3 Template based parameterization in SAGE

The Systems and Applications Genesis Environment (SAGE) tool set [3] provides support for the development of parameterized templates as well as a shelf where these templates can be archived for instantiation in application models. These are extensions of the archetypes and shelf from DOME as described in section 1.2.1. Templates provide two capabilities: (i) the capture of function blocks, their ports and data-striping; and (ii) the hierarchical composition of function blocks. While templates are not powerful enough to allow the capture of arbitrary patterns, they do provide some capabilities desirable in a DSVL. These include (i) alternate structures and code-generation schemes within a template, (ii) the addition of constraints and semantics, and (iii) the automated customization of the instantiation user interface based upon template properties. The SAGE capabilities, although implemented as special cases, give us a flavor of some important features needed in DSVLs as well as the need for underlying generalized mechanisms for implementing such features.

2 Pattern Support

A number of issues have to be addressed to support powerful and flexible pattern usage. Patterns need to be connected to model entities by arcs defined within the modeling domain. They need to allow for flexible connection types so users can specify the appropriate connection for a given instance of the pattern. Pattern instances need to be tailored to their contexts. Finally, patterns need to support code generation in conjunction with both the code generators of the models in which they are instantiated and the particular details of their implementations. In this section, we describe our scheme for addressing these issues.

2.1 Portals

Semantically, connections to a pattern complete a connection to one or more elements within the pattern. Consider a UML class diagram as an example. Connecting a class to a pattern using a generalization arc has no meaning within the UML domain. Instead, the generalization arc should continue through the pattern edge to a valid entity within the pattern. To make such connections clear, we introduce the notion of portals. The portal is something of a window into the pattern. It provides a limited view of the pattern's implementation, allowing arcs to be connected across pattern edges. It also ensures that these connections are made to only those types of objects the pattern designer has specified. Through portals, a user completes an instantiation of a pattern or completes the instantiation of a model with elements from within the pattern.

In a pattern design, a developer can connect a portal to any entity in any domain by any arc type defined in that domain. A portal on a pattern instance, on the other hand, can be connected only by the kinds of arcs in the directions specified in the pattern design (except for the *be* arc described below). Arcs connecting to one side of a portal should have a corresponding connection on the other side that completes the connection. When the pattern is flattened, the portal is replaced by an arc from the origin node (on one side of the portal) to the destination node (on its other side).

2.2 Be Arc

Some pattern designs may involve multiple arcs connected to either or both sides of a portal. Such a state can lead to difficulty in interpreting composed models and/or constructing flattened diagrams when multiple arcs of different types or directions are connected to one side of a portal.

The rules that govern the connection of arcs to portals indicate that if n nodes are connected (by n arcs) to one side of a portal and m nodes are connected to its other side (by m arcs), then the flattened diagram will contain nm arcs between the $n+m$ nodes. Each node will be attached to all of the others from the other side of the portal. For example, Figure 1(a) shows a case with three UML classes on one side and two classes on the other. The flattened UML class diagram is shown in Figure 1(b).

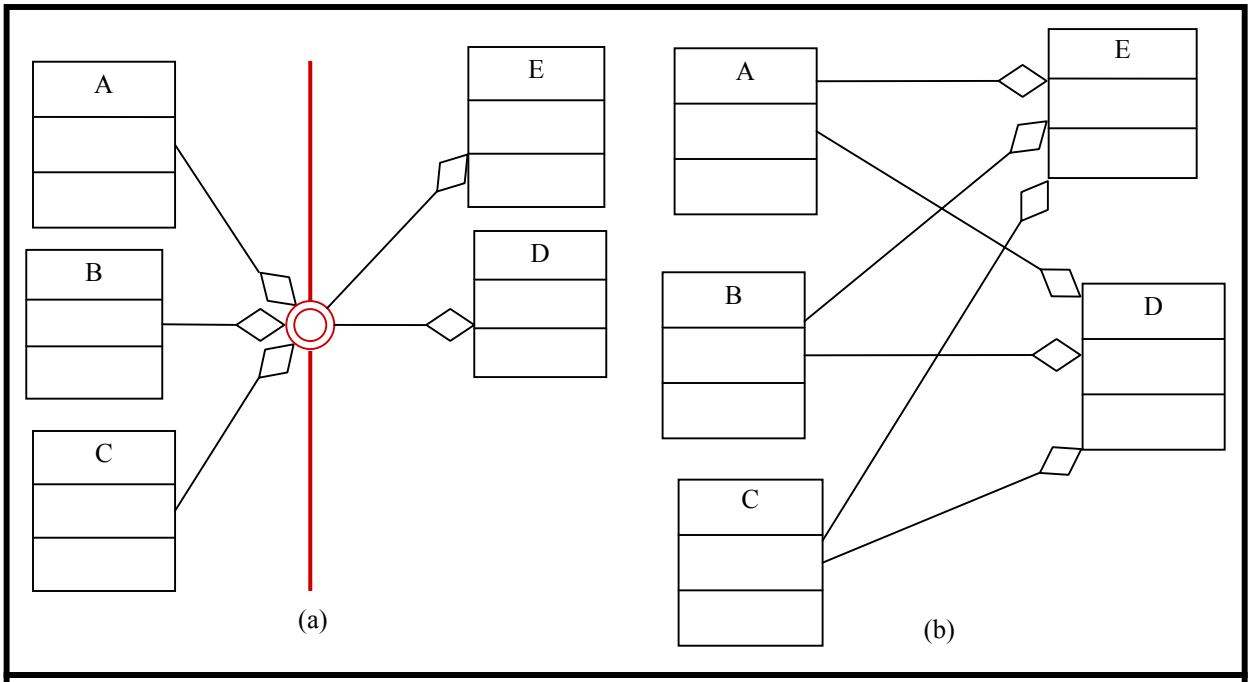

Figure 1: Multiple connections of one type in one direction through a portal.

In this example, all of the arcs are of the same type and direction. If multiple arcs that are not of the same type and direction connect to a portal, then no normal connection can be made to its other side such that the flattened diagram contains *nm* arcs. Figure 2 illustrates this problem. Here, three arcs of different type and direction are connected to one side of a portal. Any one arc can be connected between either D or E and the right side of the portal. However, the resulting flattened diagram will include unmatched arcs. Hence

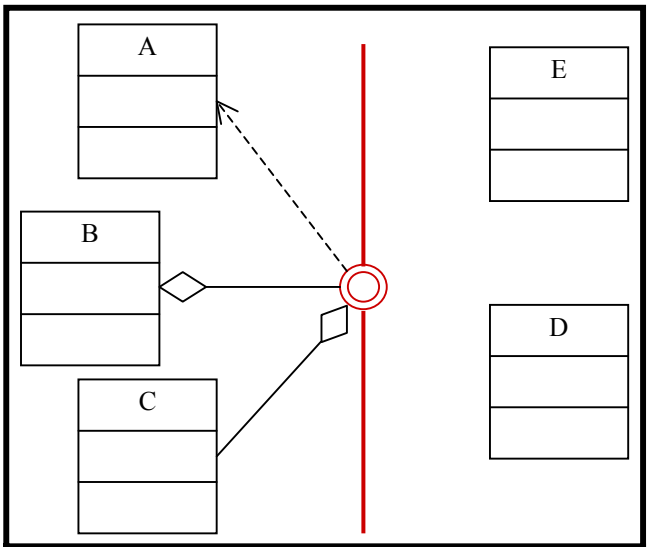

Figure 2: Multiple arcs of different types in different directions through a portal.

the semantic meaning of the pattern will be lost.

In order to address this problem, we have defined a special arc that is usable in any modeling domain to connect nodes to portals. We refer to this as the *be* arc[1]. The *be* arc can be thought of as a wildcard arc. In the flattened diagram, *be* arcs are replaced by every arc that connects to the other side of the portal. Hence, flattened diagrams never contain the *be* arc, and so only contain domain-specific arc and node entities (see Figure 3).

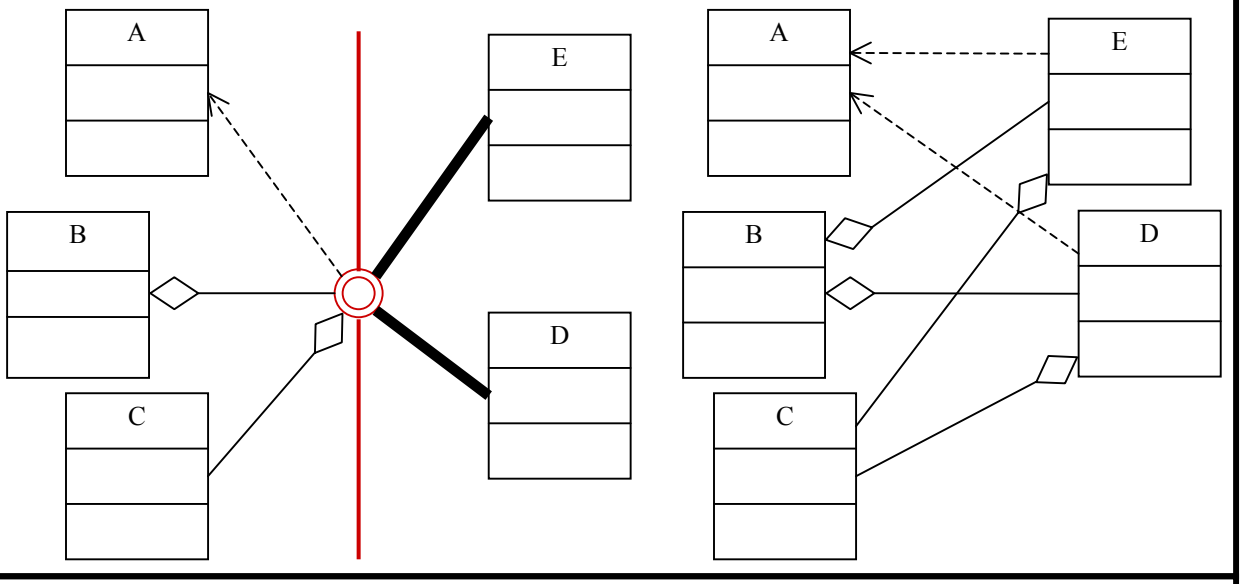

Figure 3: Using the *be* arc to connect multiple types of connectors in different directions through a portal.

The *be* arc can also be used within a pattern as a kind of an *export arc*. This is because the use of the *be* arc gives the semantic appearance that the portal is equivalent to the entity to which it is connected. Therefore the user is allowed to connect to the portal in whatever manner makes sense for that application. The reverse situation is also true whenever the *be* arc is attached to the outside of a pattern. Here, the appearance is that information propagates into of the pattern. Also, using the *be* arc outside of a pattern allows a user to connect to the pattern with less concern for the details of its implementation.

When *be* arcs attach to both sides of a portal, the nodes on either side are considered to be the equivalent. This feature is especially useful for specifying a pattern inside another pattern from the outside of it (see section 3 Examples).

[1] The *be* arc got its name because it indicates that the portal should appear to **be** the object to which it is connected.

2.3 Parameters

Some pattern implementations may be dependent upon certain instance-specific details. Consider the watchdog communication pattern (see Figure 4). In this pattern, communications are monitored. If an error occurs, the watchdog thread puts the system into a safe state. This pattern is flexible to a degree in that the communication may occur over TCP or UDP. We would like to be able to support such flexibility. Therefore, we allow patterns to have user-defined properties. These are used to select among optional implementation details. In the watchdog pattern, the code generator can select among communication protocols based on a quality-of-service property (for example, the maximum communication error rate) specified on the pattern.

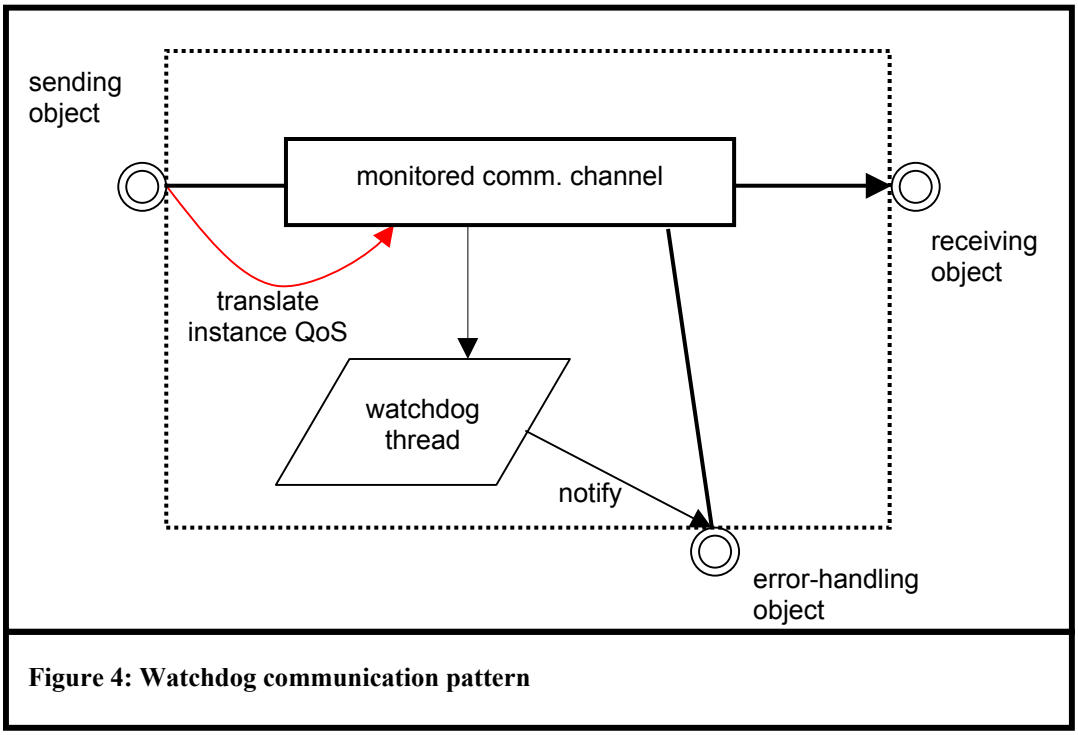

Figure 4: Watchdog communication pattern

Portals can have parameters also. These parameters can be propagated across and even between patterns. For example, a portal may have a periodicity attached to it. This value could propagate through the pattern to the entities connected to it through a corresponding output portal. The property could follow a chain of portals in order to complete the instantiation of a model (see Figure 5).

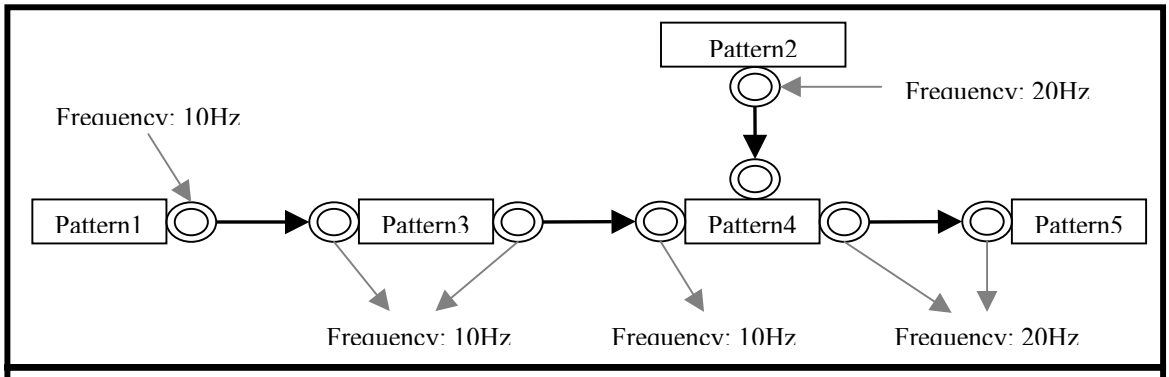

Figure 5: The Frequency specified on Pattern1 propagates to the left portal on Pattern4. Pattern4, having two different frequency inputs propagates the higher frequency (determined by the pattern design) to its output portal, which propagates it on to Pattern5.

2.4 Multiple Implementations

Many patterns have a number of different implementations. Logically, a pattern has one specific meaning, but its implementations may vary depending on the circumstances of its use. While parameters allow for adaptations to an implementation of a pattern (such as TCP versus UDP), multiple implementations accommodate fundamentally different implementation approaches. For example, a pattern may be implemented differently on different kinds of middleware or a "notification" pattern can use two implementations to allow for lazy or eager evaluation. These differences need to be captured for code generation and system analysis. The context in which a pattern is used may change the implementation significantly, but not the logical use of the pattern. Implementations can be selected on a pattern instance via parameters.

2.5 Pattern Interaction

An important aspect of patterns is the ability to combine them. Many of the examples in [2] show how one pattern can be used to help implement another. Similarly, our patterns can both be embedded in other patterns and connected at the portals. While the ability to tie pattern instances together can be powerful, it is important to consider the side effects that might occur since the user may not see the underlying implementation. In general, patterns are self-contained with objects interacting only though portals. This encapsulation simplifies pattern interaction. However, portal parameter propagation across patterns needs to be considered.

2.6 Code Generation

Automatic code generators are extremely important components of graphical modeling tool sets. Current state-of-the-art generators typically utilize code generation *templates* that are executed (or interpreted) to build up generated code. Such templates consist of (i) text that is copied directly into the generated code, (ii) special purpose subroutine calls that output text into the generated code, (iii) template-file include statements that can be used like subroutine calls, and (iv) comment text that is ignored by the code generators. As an example, consider the following two templates.

```
THIS IS A TEMPLATE TO GENERATE C++ HEADER FILES.

// c++ header file
// class declarations

#include <stdio.h>
#include <math.h>

INCLUDE classDeclarationTemplate
```

```
THIS IS A TEMPLATE CALLED classDeclarationTemplate.
IT GENERATES CLASS DECLARATIONS.

for-each node in model.nodeList
  if node.isA(classEntity)
    class node.name : public {
    private:
    for-each attribute in node.attributeList
      attribute.type attribute.name = attribute.initialValue ;

    public:
    for-each method in node.methodList
      method.returnType method.name method.parameters.format() ;
    }
```

The first is the top-level template for generating C++ header files given UML class diagrams. It contains a comment section (bold) followed by straight text to be copied directly into the generated code. It ends with a template include statement (italics). This template include statement directs the code generator to load and execute the template named *classDeclarationTemplate*. The purpose of this utility template is to write out individual class declarations given a specific UML model. It contains a comment section followed by template subroutine calls (underlined) interspersed with straight text.

It is our position that such code generation templates should be encapsulated within patterns. This approach has a number of advantages.

- Such encapsulation is beneficial from a software-engineering standpoint as code generation details for a pattern are hidden within it. Typically, the pattern developer will take care of the code generation details associated with the specific pattern during its design.
- Encapsulation explicitly links templates to model entities. This removes the requirement that top-level templates contain hard-coded knowledge of every possible utility template. Instead, the model is simply traversed, and as each pattern is encountered, its template is executed.
- Encapsulation allows for the dynamic resolution of multiple pattern implementations during code generation.

Managing complexity is a key issue for the design of code generation tools. This is because the complexity of the model increases with the richness and flexibility of the

desired level of code generation. That is, if you want to generate code for many different attributes of a system, then all of these attributes must be modeled. Our framework lets the user control the internal pattern complexity by allowing for multiple implementations of a single pattern as well as multiple instances and types of subdiagrams within various implementations. In their most basic form, patterns can be used as simply as other first-class modeling entities to build up complex models. On the other hand, they can allow for arbitrary amounts of additional specification (by means of multiple implementations and subdiagrams) in order to support rich code generation. However, even with multiple implementations and subdiagrams built from diverse modeling domains, pattern interfaces remain simple and straightforward to use. As such, patterns give us nice, small, easily understood, compact units of composition that can specify arbitrarily complex, multi-attribute, and hierarchical structures to maximize code generation while also managing complexity.

3 Examples

The three patterns below cooperate to implement the Composite pattern described in [2]. The Composite pattern allows a client to treat groups of components as individual components, presenting a single interface to the client. The essential structure of the pattern is shown in figure 6(a). The Component inherits from another class that is outside the pattern. This external class is an abstract class that is specific to a context and specifies the operations that the components must implement. The Composite class contains another pattern, MethodImpl. This pattern may expand to any number of methods, and expects to be supplied with method implementations. This use of the MethodImpl pattern allows the Composite pattern to compose components with an arbitrary number of methods. It also allows for different kinds of reductions across the corresponding child methods. Consider the instance in figure 6(b). In this example, the Draw() operation on the composite can simply iterate over all children and call Draw() on each of them. However, if the components provided a getSize() operation, then the composite operation would need to calculate its size based on the sizes of its children. Hence it would not only need to call the operation on all children, but also accumulate their return values. (This would require a slightly different implementation of the SimpleIteration pattern than the first case.) The MethodImpl pattern combined with the portal to which it connects allow us the flexibility to specify an arbitrary number of operations with multiple types of iteration or reduction across the children while still supporting rich code generation.

The SimpleIteration pattern shown in figure 6(b) is an example of a pattern that takes one or more operations as input and generates the corresponding composition operations according to some rule. For example, iterate over all children and call the operation on each child. This pattern exports one operation for every imported operation.

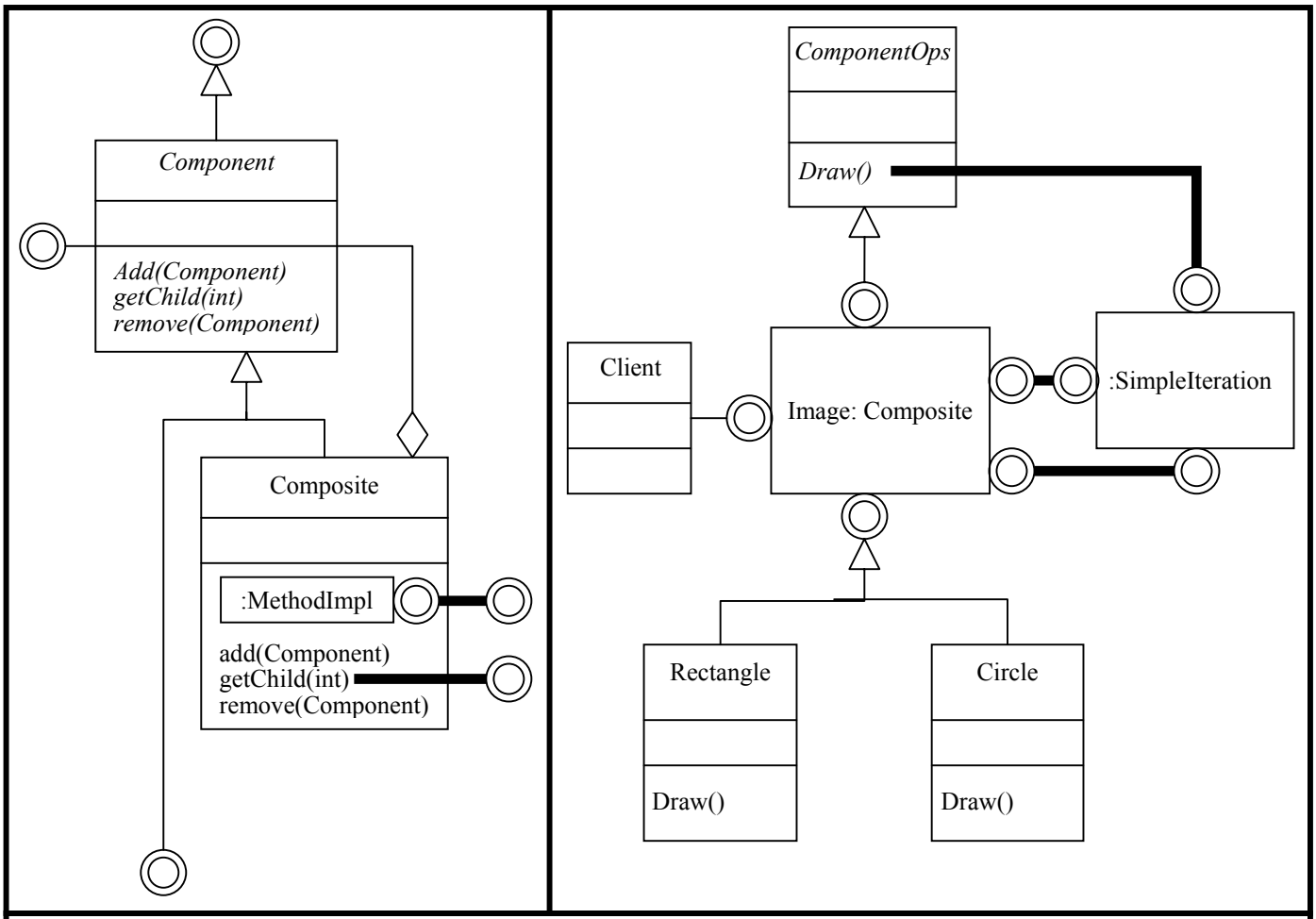

Figure 6: The internal details of the Composite pattern (on the left) and an example of external connectivity to it (on the right).

4 Conclusion

There are synergies between domain specific visualizations and patterns that have not been fully exploited. DSVLs help insulate the developer from implementation concerns outside the scope of the problem in the same way third-generation languages hide the details of register allocation. Their domain-specific nature makes them powerful because they are specialized and easy to learn and use because they work with concepts that are familiar to experts within the domain. Patterns provide developers with a library of solutions to common problems. They are powerful because they are flexible and well-designed solutions that have been proven through multiple applications. They also comprise a language among developers that make designs easier to understand. Our pattern framework allows for patterns in and across diverse modeling domains. These patterns insulate developers from their implementations and behave according to the syntax and semantics of the domain in which they are applied. They also provide reuse through pattern composition, cross-domain aspect modeling, and support for powerful code generation.

As we delve further into the implementation of the pattern framework, we will need to address the issue of consistency checking. For example, the *be* arc can be used to connect a portal with any model entity or another portal. However, this flexibility can lead to incorrect and/or inconsistent models. For example, if the entity is a UML object in a class diagram, the syntax of the model does not allow the object to connect to anything via any arc type. Also, two portals connected via a *be* arc require arcs of the same type in opposite directions (into one portal and out of the other) on their other sides. This problem requires that a consistency check be performed before any flattening operation. Since flattening will proceed any analysis or code generation, inconsistent models will be avoided during these phases.

1 G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
2 E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
3 Honeywell International, "The Systems and Applications Genesis Environment (SAGE)," htttp://www.honeywell.com/SAGE.
4 A. Ledeczi, G. Nordstrom, G. Karsai, P. Volgyesi, and M. Maroti, “On Metamodel Composition”, *Proceedings of IEEE CCA 2001*, 2001.
5 MetaCASE Consulting, “Domain Specific Modeling: 10 Times Faster Than UML”, Technical Report White Paper, MetaCASE Consulting, 2001.
6 K. Schloegel, D. Oglesby, E. Engstrom, D. Bhatt, “A new approach to capture multi-model interactions in support of cross-domain analyses”, 2001.
7 DOME is an open source research project and is available from http://www.htc.honeywell.com/dome.