Generative Design Patterns

S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling and K. Tan
Department of Computing Science, University of Alberta, Edmonton, AB T6G 2H1, Canada
{stevem, duane, jonathan, janvik, bromling, cavalier}@cs.ualberta.ca

Abstract

A design pattern encapsulates the knowledge of object-oriented designers into re-usable artifacts. A design pattern is a descriptive device that fosters software design re-use. There are several reasons why design patterns are not used as generative constructs that support code re-use. The first reason is that design patterns describe a set of solutions to a family of related design problems and it is difficult to generate a single body of code that adequately solves each problem in the family. A second reason is that it is difficult to construct and edit generative design patterns. A third major impediment is the lack of a tool-independent representation. A common representation could lead to a shared repository to make more patterns available. In this paper we describe a new approach to generative design patterns that solves these three difficult problems. We illustrate this approach using tools called CO_2P_2S and Meta-CO_2P_2S, but our approach is tool-independent.

Keywords: design patterns, frameworks, programming tools.

1. Introduction

The most common form of a design pattern is a *descriptive* one such as a pattern catalogue entry or a Web page. This form preserves the instructional nature of patterns, as a cache of known solutions to recurring design problems. Design patterns provide a common design lexicon, and communicate both the structure of a design and the reasoning behind it [2].

We use the well-known *Composite* design pattern as an example [10]. Figure 1 shows how this design pattern maintains part–whole object hierarchies, where individual parts and compositions of parts are treated uniformly. It supports hierarchy traversal operations. A leaf class instance implements each traversal as a local operation on itself. An composite instance implements the traversal as an optional local operation on itself, together with calls that apply that same operation to each of its child objects. Each child may be a leaf instance or another composite. There are also a set of child management operations like `add(Component)` and `remove(Component)`.

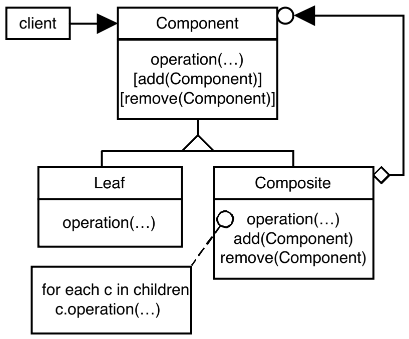


Figure 1. The Composite design pattern.

In addition to a diagram like Figure 1, the design pattern documentation usually contains a description of the pattern that consists of eleven parts: intent, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses and related patterns [10]. There are many variations of this pattern, but in this paper we are using the pattern as defined in [10] since that is the most common version. In Section 4.5 we show another version of this pattern as an example of pattern evolution.

During application design, each pattern must be *adapted* for use, since each pattern is a family of solutions. Each family member has the same basic structure but must be adapted to its specific context. For example, if a Composite pattern is used for drawing, then particular leaf and composite classes must be identified. Perhaps the leaf classes are `Line` and `Circle`, while the composite classes are `Group` (an arbitrary collection of other components) and `Quadrilateral` (a specific collection of four lines). Some specific methods must also be identified during adaptation. For example, there may be two traversal operations: `draw(GraphicsContext)` and `boundingBox()`.

During the implementation process, the adapted design pattern is a specification. Experienced programmers can quickly transform the specification into code since they have probably implemented the Composite pattern many times. For novice programmers, the process of coding the design pattern is more difficult and error-prone. This is partly due to the fact that design patterns are written documents that are subject to human interpretation. This makes them vulnerable to the ambiguities in natural language. An incorrect interpretation of a pattern can lead to an incorrect implementation. It would be beneficial to use *generative* design patterns that generate code. They reduce implementation time, are less prone to programmer error, promote rapid prototyping and code reuse, support performance tuning, and provide better overall software engineering benefits.

1.1 Generative design pattern problems

Unfortunately, there are several problems that must be solved to make design patterns generative. The first problem is related to design pattern adaptation and the interplay between adaptation and code generation. There are three choices for when to generate code: before adaptation, during adaptation, or after adaptation.

First, consider code generation before adaptation. Since a design pattern represents a broad solution family, the generated code will be quite complex and quite generic. There are several ways of dealing with the complexity. First, the generated code could be a framework so that adaptation is done by framework specialization [11]. However, most design patterns represent families of solutions whose structures cannot be adequately represented by a static framework.

As an example, consider the name and arguments of the traversal operations in the Composite pattern. One program may require `draw(GraphicsContext)`, while another program may require `containsPoint(Point)`. A static framework cannot meet these conflicting requirements unless the traversal code is not included in the framework. This defeats the purpose of generating the code from the design pattern. Un-adapted design pattern code is too complex to represent by a static framework. The complexity can sometimes be mitigated by a mechanism that supports cross-cutting. For example, the names and parameter lists of the traversal operations could be generated using aspect-oriented programming (AOP) [12]. In this case the names and signatures could be weaved with a generic framework to produce the final framework code. However, this is just an example of generating code during adaptation.

In fact, there are many situations when cross-cutting cannot compensate for generating code too early. For example, in the Composite pattern, there is a choice (safety or transparency) about where to declare the child management operations. In the safe implementation, they are declared in the composite class. Applying these operations to a leaf class instance results in a compile-time error. The disadvantage is that you must query a component to get its type or perform an unsafe type-cast before you apply a child management operation. In the transparent implementation, child management operations are declared in the component class. In this case, you can apply them on any components without type testing or casting. The disadvantage is that you must implement the operations in the leaf classes. You might implement these operations to do nothing for a leaf instance. However, in this case, trying to apply one of these operations to a leaf class instance might signal a logic error that you would miss. The important point is that it is impossible to generate a single static framework that will be adaptive to both choices: safety or transparency.

When code is generated before adaptation, there can be a genericity problem. Design patterns try to introduce flexibility to support application evolution that is often called "designing for change" [10]. This flexibility is usually achieved by adding indirection between objects. Unfortunately, this approach requires more code to be written and maintained and the indirection can reduce performance. In effect, the indiscriminant use of patterns can result in a slower application [17]. For example, we may not want to use the same kind of implementation for our `Quadrilateral` and `Group` classes. We can avoid indirection in our `Quadrilateral` class by using four `Line` instance variables.

In summary, it is very difficult to generate code before adaptation that is general for all of the problems the pattern is designed to solve. If this problem could be solved, the code would often be too complex and often too generic to achieve good performance.

Second, consider code generation after adaptation, which can generate simple efficient code. Unfortunately, during adaptation, the pattern quickly gathers application-specific characteristics. For example, the identification of a `draw(GraphicsContext)` during pattern adaptation means that if code is generated after this adaptation, the generated code is only applicable to applications similar to our graphics application. This approach leads to an explosion of specialized versions of each pattern or the situation where we need a version that is slightly different than the available versions and we don't have it.

The third approach is to generate code during adaptation. In general, there can be an arbitrary number of adaptation and code generation cycles. However, we are interested in the simplest process that provides usable results. Therefore we propose (and have created) a simple three-phase process consisting of: initial adaptation, code generation and final adaptation. The initial adaptation phase adapts the design pattern to a single structural and control flow model suitable for the application. The code generation phase generates a framework for this architecture and the final adaptation phase performs standard specialization operations on the framework to adapt it to the final application domain. This approach solves the problem that frameworks cannot be easily adapted to support different structures or control flows, while maintaining the advantage that framework specialization is a good technique for specializing code for specific applications. A framework is the most popular adaptation technique that can be used after code generation. Other popular adaptation techniques (such as AOP) occur before code generation.

1.2 Parameterized design patterns

Published descriptive design patterns include lists of participants, implementation issues and sample code. These three parts of the pattern are used most when implementing a design pattern manually. However, they are not sufficient for generative design patterns. The key novel contributions of our research efforts is to quantify each adaptation option as a parameter with a fixed set of possible values that must be set during the initial adaptation process. There have been several efforts to produce generative design patterns. Some of these approaches are ad-hoc and some use more structured approaches [4][6][7][8][9][19]. However, our approach is new. Each descriptive design pattern is transformed into a generative design pattern using quantitative parameters with specific domains. We have discovered that although the specific parameters vary from design pattern to design pattern, the parameters can be classified into a few distinct types. After this initial adaptation, our approach has the flexibility of further adaptation by specializing a framework that still spans many application choices.

In this paper we show that our approach is viable by describing solutions to these three important problems:

1. Generative design patterns are conceptually complex. However, with good abstractions, a well-defined process, and proper tool support, the difficulties can be managed. Each design pattern must be analyzed to identify specific adaptation parameters. Then, the legal values of these adaptation parameters must be specified and code must be written that implements each combination of legal values. Since the number of combinations is exponential in the number of parameters, tool support is essential.

2. Generative design patterns must be adapted by the pattern user, and this is conceptually complex. Once again, the right abstraction, a well-defined process, and tool support are required. Tool support should help the user assign legal values for each design pattern parameter, generate code based on these parameter values, and support specialization of the generated framework code.

3. If only a few generative design patterns exist, few people will use them. A standard tool-independent representation is necessary to foster sharing and to encourage the construction of generative pattern repositories. However, due to the wide variability of application requirements, generative design patterns must be living entities.

In this paper, we present a solution to these three problems. Our approach to generative design patterns is independent of programming language and tools. However, to validate our process we have implemented two support tools, CO_2P_2S (Correct Object-Oriented Pattern-based Programming System - www.cs.ualberta.ca /~systems/cops) [1][13][14] and Meta-CO_2P_2S [5]. CO_2P_2S is used to adapt design patterns and generate framework code for use in specific applications. The complete process takes three steps. First, the user selects an appropriate generative design pattern from a set of supported patterns. Second, the user adapts this pattern for their application by providing parameter values. Finally, the adapted generative pattern is used to create object-oriented framework code for the chosen pattern structure. Meta-CO_2P_2S is used to create new generative design patterns, to generate tool-independent representations and to edit and evolve existing design patterns. Figure 2 shows the specialization flow, from generative design pattern to final application code.

Section 2 describes our process for generative design pattern adaptation using the Composite pattern as an example. Section 3 describes our solution to the difficult problem of constructing generative design patterns. Section 4 looks "under the covers" at the tool-independent representation that is generated during the generative design pattern construction process described in Section 3. Section 5 summarizes the paper. One of the important challenges of our approach is to manage the evolution of generative design patterns, especially the tension between genericity and simplicity.

2. Design pattern adaptation - parameters

In this section we describe a new view of generative design patterns that is parameter-based. We use the Composite pattern as an example pattern because it is well-known and has a variety of parameters. We also use the CO_2P_2S tool as an example of an adaptation and generation tool. The specific pattern and tool are not as important as the idea of parameterized patterns.

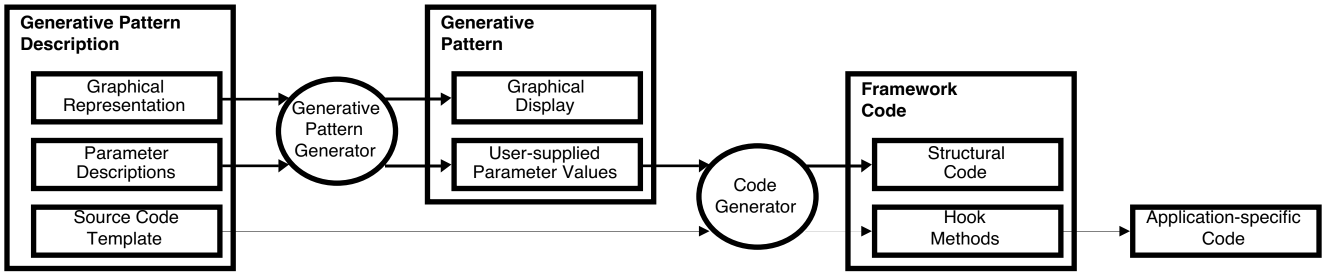


Figure 2. Patterns to frameworks to applications.

2.1 Parameters for the Composite pattern

The Composite design pattern has seven parameters. The parameters are divided into four categories, *lexical parameters*, *design parameters*, *performance parameters* and *verification parameters*. A *lexical parameter* is used to specify a class name, method name or some other syntactic structure in the generated framework. A *design parameter* affects the structure of the generated framework. A *performance parameter* only affects the performance of the generated code. When a *verification parameter* is turned on, the generated code performs run-time semantic checks on user-supplied code. When it is off, no verification code is generated. The Composite pattern has four lexical parameters, two design parameters and one performance parameter:

1. (lexical) The name for the abstract component class in the generated framework, called component name.
2. (lexical) The name for the abstract composite class in the generated framework, called composite name.
3. (lexical) The name for the abstract leaf class in the generated framework, called leaf name.
4. (lexical) A name for the pattern superclass, called superclass name.
5. (design) The location of the child management operations – safe versus transparent, called safe-transparent.
6. (design) A traversal list, called operation list.
7. (performance) The types of containers used, called containers.

Each parameter has a name so that it can be referenced and a type that specifies its legal domain of values. Although the parameter names are pattern-specific, the types are re-used across patterns. We use the simple drawing application from Section 1 to illustrate parameters and parameter value selection. Assume we want to generate a framework that supports an operation `draw(GraphicsContext)`, that draws any component on a `GraphicsContext`, and an operation `boundingBox()` that returns a Rectangle that bounds an arbitrary component. To make the framework more realistic (and challenging), we assume that when a composite instance draws itself, it must first draw a background and then draw each of its components. We assume that we will need at least two application leaf classes: `Line` and `Circle`. We will also need at least two composite classes: `Group` and `Quadrilateral`. A `Group` can contain an arbitrary collection of components. A `Quadrilateral` is a specific collection of four `Lines`.

The user controls the names of the abstract classes in the composite pattern of Figure 1, by supplying values for the *component name*, *composite name* and *leaf name* parameters of type *Class Name*. For example, the user might use the names: `DrawingComponent`, `ListComposite` and `DrawingLeaf`. The user also enters an arbitrary *superclass name* for the component class. This facility is necessary to support general composition of generated framework code. If no superclass is required, the user can enter `Object`. In this paper, we will use Java as an example target language. If C++ was the target language, an empty superclass name could be used.

The design parameter, *safe-transparent*, has parameter type *Enumeration*. Each *Enumeration* parameter has a small set of legal values. In this case, there are two values: *safe* and *transparent*. If the *safe* value is selected, the child management operations are generated in the composite class. If the *transparent* value is selected, the child management operations are also generated in the component class.

The sixth pattern parameter is used to select all of the traversal operations that are generated for the pattern. The *operation list* parameter has type *List*. In general, a *List* parameter is a list of parameters of arbitrary type. In this case, each parameter in the list represents a traversal operation. For a *List* parameter, the user can add as many elements to the List as necessary. In our example, the list has size two since we must generate two traversal operations. Each list element in the *operation list* has type *Structure*. A *Structure* parameter has a fixed number of sub-parameters. In this case, each *operation list* element must represent all of the information necessary to generate code for one traversal operation.

To define the structure of each operation list element, we must look at what kind of information it must specify. In some composite applications, it is necessary to perform a computation in each composite instance, in addition to delegating the traversal operation to its child instances. In general, this composite action may occur before or after the child instances are traversed. For example, in our `draw(GraphicsContext)` method, a composite instance must draw a background before calling the `draw(GraphicsContext)` method on each of its children to draw the foreground. This is an example of a prefix method since it is called before traversing the child instances. As a second example, consider our `boundingBox()` method. Each composite instance must invoke this method on each of its children and then use the results to construct a bounding box from all of the Rectangle objects that are returned from the children. Such a method is called a suffix method and it takes as an argument an array of the results of the traversal operation that has been applied to its children.

Each element in the *operation list* must specify the signature of the operation method, whether the operation has a prefix method or not and whether the operation has a suffix method or not. If the operation has a prefix method, the name must be specified. The signature of the prefix method will be the same as the operation method that calls it. For example, the user will specify that the draw operation has method signature `void draw(GraphicsContext gc)`, that it has a prefix method and that it does not have a suffix method. The user will specify that the prefix method is called "`drawBackground`". The signature of the prefix method will be `void drawBackground(GraphicsContext gc)`.

If the operation has a suffix method, the name of the suffix method must be specified. The signature of the suffix method will include all of the parameters of the operation method that calls it, plus an array of the return values of the operation method applied to the children. So, in our example, the user will also specify that the boundingBox operation has method signature `Rectangle boundingBox()`, that it has no prefix method and that it has a suffix method. The user will specify that the suffix method is called "`computeBoundingBox`". The signature of the suffix method will be generated as `Rectangle computeBoundingBox(Rectangle[] anArray)`.

Each element of the *operation list* has type *Structure* and has three sub-parameters. The first sub-parameter, called *operation signature* has type *Method Signature*. The other two sub-parameters, *prefix* and *suffix*, each have type *Structure* and share a common structure.

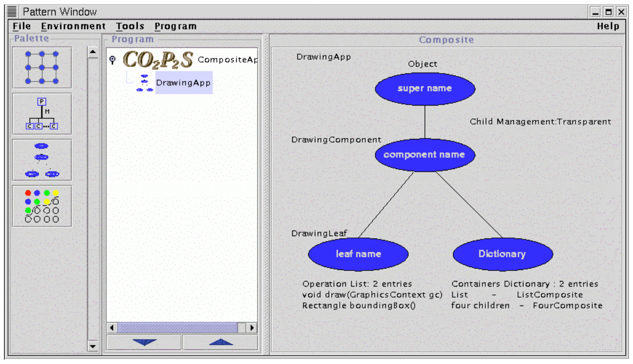
The *containers* parameter is the only performance parameter of the Composite pattern. Each composite object must store its child components in a container. For example, a vector, array, linked list, or hash table could be used. The overall structure of the generated code is independent of the kind of container used and the choice of container *mainly* affects the relative performance of the `add(Component)`, `remove(Component)` and traversal methods. However, the choice can have some semantic consequences as well, since if a hash table is chosen, the order of elements is arbitrary and a traversal may select children in any order. Similarly, if an array is chosen, the number of children has a fixed maximum.

Alternately, to increase performance by reducing indirections, a fixed number of named instance variables may be used. This container choice must be made before code generation, since the code for `add(Component)`, `remove(Component)` and the traversal operations is generated. In fact, the user may want to create application subclasses for more than one choice. For example, the `Group` of drawing objects should be a subclass of a composite that uses a variable sized container like vector, linked list or hash table. For efficiency, `Quadrilateral` could be a subclass of an abstract composite class that has four named instance variables.

The *containers* parameter has type *Dictionary.* It supports an arbitrary number of abstract composite classes, with different names and implementations. Each key in this *Dictionary* is an *Enumeration* value from: {vector, array, list, hash table, two children, three children, four children}. The last three choices generate a fixed number of instance variables. If a fixed number of children are required that is more than four, then an array should be used. This *Dictionary* starts with one entry whose default key is *list* and whose value is the value of the *composite name* parameter. New entries are added for each container that is selected from the *Enumeration*. For example, in our application the *composite name* is `ListComposite`. Notice that there are two abstract composite classes, `ListComposite` to support `Group` and `FourComposite` to support `Quadrilateral`.

2.2 Tool support for setting parameters

Figure 3 shows a sample CO_2P_2S screenshot with parameter values for this drawing application. Tool support can make the adaptation process straightforward. The left side of the tool provides a palette of supported generative design patterns, of which the Composite is one example. The right side of the tool shows the current state of the selected pattern. The set of generative patterns in an application is shown in the center pane. CO_2P_2S uses dialog boxes to guide the user through the parameter value selection process, by constraining user choices. For example, CO_2P_2S uses radio buttons for Enumeration parameters and checks for syntactic correctness of Identifier parameters after they are entered in a text field.



Operation List: 2 entries
void draw(GraphicsContext gc)
Rectangle boundingBox()

Containers Dictionary : 2 entries
List - ListComposite
four children - FourComposite

Figure 3. Composite pattern for the Drawing application.

The parameter types described in Section 3.1, *Class Name*, *Method Name*, *Method Signature*, *Enumeration*, *Boolean*, *Structure*, *List* and *Dictionary*, are most of the types required to implement our current set of supported design patterns. In fact, as described in Section 3.1, these parameter types can be expressed in terms of three simple parameter types: *String*, *Enumeration* and *List*. All parameter types can be constructed from these three basic types and a fourth type, called *Extended*.

After parameter value specification, the adapted design pattern is used to generate a code framework. This framework can be further adapted using framework specialization. For example, the user can create the classes `Line`, `Circle`, `Group` and `Quadrilateral` as subclasses of the abstract classes `DrawingLeaf`, `DrawingLeaf`, `DrawingComposite` and `FourComposite` respectively. The user inherits the code for the two traversal operations `draw(GraphicsContext)` and `boundingBox()` in each of the composite classes `Group` and `Quadrilateral`. The user must implement each of these methods in the leaf subclasses `Line` and `Circle`. In addition, the user must implement the `drawBackground(GraphicsContext)` prefix method in the `Group` class to draw whatever background is desired. The `Quadrilateral` class can just inherit the default implementation of `drawBackground(GraphicsContext)` which does nothing. The user must also implement the `computeBoundingBox(Rectangle[])` suffix method in the `Group` and `Quadrilateral` classes. Other leaf and composite classes can be added by framework specialization. However, if the user wants another abstract composite class with a different implementation strategy, the user will have to edit the adaptation parameters and re-generate the code. For example, if the user wants to add a `Triangle` class as a subclass of an abstract composite class that has three instance variables, another entry must be added to the *containers Dictionary* and the code must be re-generated. This regeneration will not affect the code already written in the user's concrete framework subclasses.

3. Constructing generative patterns

For a new generative design pattern, the designer begins by defining each parameter and its parameter type. This information should be stored in a tool independent representation. In Section 4, we will describe our proposed XML-based representation. Besides storing parameter names and types, the representation also supports information that can be used to generate a graphical user interface when the pattern is loaded into an end-user tool that supports pattern users. Patterns and parameter values can be attributed with labels, images and layout information.

For example, the CO_2P_2S tool uses such pattern attributes to generate a graphical user interface for each pattern, so that the pattern user can easily set all of the pattern parameters. The basic CO_2P_2S environment includes no patterns. Instead, any generative design pattern can be loaded into CO_2P_2S using a simple *load pattern* operation from a menu. At this point, the pattern representation is read and the entire GUI for that pattern is generated and added to CO_2P_2S. The new pattern appears in the palette and all of the dialogs for setting parameter values are also added. The user is free to add as few or many patterns to CO_2P_2S as needed. As new patterns become available, they can be loaded on demand.

Even though the pattern representation is tool independent, tool support for parameter definition serves two fundamental purposes. First, it reduces errors by providing menu-based choices for legal parameter types and by automatically generating the details of the storage format. Second, it supports rapid editing or generalization of existing generative design patterns. The pattern designer can load a pattern, browse its parameters and add new parameters. The CO_2P_2S tool was first constructed to support parallel programming. In fact it was called CO_2P_3S, where the third "P" stood for *Parallel*. It began with several "built-in" patterns. We quickly realized that adding patterns by hand was too slow and error-prone. We created the pattern representation and Meta-CO_2P_2S tool to support rapid pattern addition.

3.1 Generative pattern parameter types

Generative pattern parameters allow a pattern designer to customize a pattern and alter the framework code that it generates. These parameters are a key part of preserving the idea that our generative patterns are still a family of solutions that can be applied in a particular context. It must be possible to adapt the generative pattern to its intended use.

If we permit the pattern designer to create generative patterns with arbitrary parameter types, then describing these parameters will require considerable effort. Instead, to simplify the description of the generative patterns, we enumerate the different types of parameters that can be used. There are three basic parameter types (*String*, *Enumeration* and *List*), and all other types are derived from them. Figure 4 is a specialization diagram for these types.

Each *String* is a simple legal string value.

An *Enumeration* is a value from a fixed set of values.

A *List* handles the common situation where the pattern user supplies a list of other parameters, which may be of any type.

An *Identifier* has a value that is any legal identifier in the target programming language.

A *Class Name* is included in each generative pattern. Pattern users should be able to specify class names in the generated frameworks so that they are meaningful in the context of the application. In addition this allows multiple instances of a pattern in the same application, without name clashes. All of the generative patterns have at least one class name that the user provides. Normally, this is the name of the class that the user must specialize. The framework class names are derived from this user-supplied name by adding text to make them unique.

A *Method Name* represents the name of a method. It is especially useful when the rest of the signature is fixed by the pattern or by previous user choices.

A *Boolean* is a special case of *Enumeration*, where the legal values are *true* and *false*.

A *Structure* deals with the common case where a parameter consists of a fixed number of sub-parameters.

A *Method Signature* specifies the name, return type and argument types of a method. It is a special case of a *Structure* parameter since a signature consists of the return type (*Identifier*), a *Method Name*, and an *arguments List*. The elements in the *arguments List* are *Structure* parameters with two sub-parameters, an *argument type* (*Identifier*) and an *argument name* (*Identifier*). It is possible to define a separate argument parameter type for the Java target language. An argument type would be restricted to a *Class Name* or a primitive type. It is also possible to define a return type since it is either an argument type or *void*.

A *Dictionary* maps keys to values for the code generator. This is particularly useful for multiple selection parameters, where each selected value must be mapped to a class name for code in the generated framework.

An *Extended* parameter supports the case where the parameter values have an arbitrary form. The pattern designer must provide extra information for each of these parameters, such as a way for the pattern user to supply the parameter value and how different values affect the generated code.

These parameter types are sufficient to cover a broad range of generative pattern parameters. *Extended* parameters can be used to implement any other parameter types. In fact, if a new parameter type is needed for a pattern and it cannot easily be derived from one of the existing types, it usually starts out with type *Extended*. All parameter types except for *String*, and *Enumerated* started as *Extended* parameters before they were promoted to named types and derived from existing types.

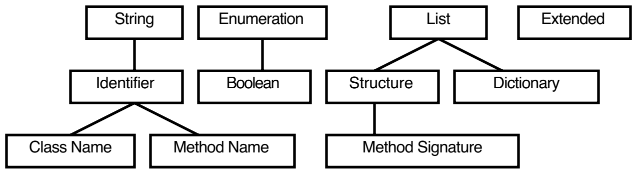


Figure 4. The specialization hierarchy for parameter types.

3.2 Framework generation

After a pattern user specifies values for all of the parameters, the set of values is used to generate object-oriented framework code. The code generation problem is a problem of conditional compilation of a code template, based on the set of parameter values. We need a way to represent each parameter in the code template and to transform each parameter to concrete code. The transformation is based on:

• the parameter definitions for a particular design pattern like the ones described in Section 2.1.

• the design pattern independent parameter type definitions listed in Section 3.1,

• and the values of the parameters for a particular design pattern as described in Section 2.1.

We can enumerate the set of representations and the potential transformations that must be supported by the code generator to derive its basic requirements:

• Each *String* parameter is represented by a *String Placeholder* in the source code template. It must be replaced with its parameter value during code generation. Sometimes, a transformation directly replaces the *String Placeholder* by a user-supplied parameter value. Sometimes the *String Placeholder* is constructed from the user supplied *String* value. For example, framework class names that are hidden from the user are derived from the user-supplied class names by adding additional text that describes the role of the class in the framework. *Identifier*, *Class Name* and *Method Name* parameters all map to *String Placeholders*.

• Each *Enumeration* parameter is represented by a *Guard Variable* that can be assigned one value from the domain of the *Enumeration* parameter. The key requirement for code generation is that methods, parts of method bodies and variables can be conditionally generated based on the value of a *Guard Variable*. To reduce the complexity of code generation, conditional generation based on a combination of values for different *Guard Variables* should also be supported. Boolean parameters map to simple *Guard Variables* with legal values *true* and *false*.

• Each *List* parameter is represented in the code by a *List Placeholder* that indicates its location. As the parameters in a list are expanded, the code for each parameter in the list is decorated and concatenated. Note that it is possible to nest lists. *Structure* parameters, *Method Signature* parameters and *Dictionary* parameters also map to lists.

• Each *Extended* parameter has no fixed transformation technique for code generation. An *Extended Placeholder* marks its location in the code template. However, it is an ad-hoc parameter type for which the pattern designer must provide custom code generation support. When the placeholder is encountered during code generation, a designer-supplied method must be called that returns the code to be inserted.

3.3 Using Javadoc for code generation

In this Section, we describe a code generator that we implemented using Javadoc. Although Javadoc does not directly support all of the types defined in the previous section, it has sufficient functionality to test our approach. All types that weren't implemented directly were implemented using the Extended parameter type.

Javadoc is a tool whose original purpose was to generate HTML formatted API documentation for Java classes. Javadoc parses Java source code files. In fact, it only parses declarations. However, Javadoc also parses specially formatted comment blocks that:

• start with /**,

• end with */,

• include lines that contain a pre-defined or user-defined tag name (`@identifier`) followed by a blank and text,

• are followed on the next line by a class declaration, field declaration, or method declaration.

For example, Figure 5 shows a Javadoc comment block that contains two tag names and precedes a method declaration. The first tag, `@param`, is a pre-defined tag name. It is a standard part of Javadoc that is used to provide a comment for each parameter in a method declaration. The other tags, `@parameter` and `@editable`, are user-defined tags that we have defined for code generation purposes (described below).

Javadoc has been extended to support pluggable Doclets. A *Doclet* is a Java program that can receive parsing information from Javadoc. This information includes declarations and comment blocks and is provided using the Doclet API, which provides access to the following information for each class: the imported classes and packages, the package of the class, the class declaration, and the declarations of the constructors, methods and fields.

```
/**
    * Iteration op for a top edge node in a 4 point mesh.
    *
    * @param east the node to the right
    * @param south the node below
    * @param west the node to the left
    * @parameter numNeighbours 4
    * @parameter boundary Non
    * @parameter boundary Horizontal
    * @editable
    */
   public void topEdge(SP_MeshElement east,
       SP_MeshElement south, SP_MeshElement west)
   {
   }
```

Figure 5. Using javadoc for parameters.

For each declaration, Javadoc also provides the text and tags from any associated comment block. By defining our own tags, we can allow the pattern designer to write code templates in standard Java with all pattern parameter information either in *String Placeholders* or encoded in Javadoc comments. This means that the code templates are much easier to read and that they can be compiled with no pre-processing during testing. Our code generator calls Javadoc and gains access to the parsed information using a Doclet. Javadoc has been used before for code generation [16], but in a simpler context.

String Placeholders are represented using the syntax of Java identifiers, so Javadoc can parse them. Since they all begin with a distinctive `SP_` prefix the Doclet can recognize them. Since they contain the parameter name, the Doclet can simply look up the parameter name in the parameter dictionary and replace them by the parameter value that is also a Java identifier. For example, Figure 5 shows the *String Placeholder*, `SP_MeshElement` that represents the user-defined class name of a mesh element. All other representations can be expressed in terms of *Guard Variables*, *List Placeholders* and *Extended Placeholders*.

Guard Variables that guard classes, methods or fields are expressed using the `@parameter` tag. The declaration that follows the comment block containing the tag is only generated if the parameter named in the tag has the value that follows the parameter name. If generation depends on one of several acceptable values, then the parameter value is repeated on separate lines, each with one acceptable value. If two different parameters must have specific values for code generation, then each different parameter should appear on a separate line with its required value. This allows arbitrary *ORing* of parameter values for the same parameter and arbitrary *ANDing* between different parameters. For example, Figure 5 shows the first line of a method that is generated only if the `boundary` parameter has the value `Non` or `Horizontal`, and the `numNeighbours` parameter has the value `4`.

List Placeholders and *Extended Placeholders* are both supported in Javadoc using the tag *`@extParameter`* followed by the parameter name. For example, consider the *operation list* parameter for the Composite pattern. Code for the *operation list* must be inserted in two different locations in the Composite code template. A list of traversal operation methods must be inserted into the component class. This list could either consist of abstract methods, or methods with empty bodies. The same list of traversal operation methods must be inserted into the composite class. However, this list must have functioning method bodies that delegate the operations to the child objects plus optional calls to prefix and suffix methods, depending on user-selected parameter values. The pattern designer will place the same Javadoc tag and parameter value, `@extParameter operationList`, into the class files for the component class and the composite class. Somehow, Javadoc must expand these two tags into different code in the two contexts.

The pattern designer must implement a Java class for each *List* or *Extended* parameter that is used in a pattern. If the class represents an *Extended* parameter it must be a subclass of a pre-defined CO_2P_2S class called `AbstractPatternParameter`. If the class represents a *List* parameter, it must be a subclass of a pre-defined CO_2P_2S class called `PatternListParameter`. In either case, it must implement the method `public List getCodeGenMethod(String classname)`.

When Javadoc encounters the `@extParameter` tag, it calls our Doclet and passes it the tag name and parameter name. The Doclet code maps the tag and parameter name to the corresponding class written by the pattern designer. For example, the parameter *operationList* may be mapped to a class called `OperationList`. The Doclet creates an instance of the class and invokes the `getCodeGenMethod(String classname)` method on this instance. The pattern designer must implement this method in each *List* or *Extended* parameter class that is defined for the pattern. This method checks the classname argument and generates the appropriate code that should appear in this class. For example, in the Composite pattern, the code would generate abstract methods for the component class and traversal methods for the composite class. To generate the correct code, the method checks the data structure that contains the user-selected parameter values for the pattern being adapted.

Unfortunately, our simple Javadoc implementation will not work for generating code inside a method body, since Javadoc does not parse method bodies. Therefore, we have written a small macro processor to support code generation inside method bodies. To simplify the macro processor and to support default method bodies for some generated methods, we have separated the bodies of all methods in the code template from the methods that contain them. Therefore, each class in a code template currently consists of a class skeleton file that contains empty method bodies together with a directory of files, one for each method body, defined in the class. At code generation time, the class skeleton file is processed by Javadoc and the method body files are processed by the CO_2P_2S macro-processor.

CO_2P_2S provides an HTML-based browser for viewing the generated framework. The pattern designer marks some of the framework methods as editable (using a user-defined Javadoc tag called *@editable*). The editable methods are called hook methods and although default implementations are supplied to the user, the browser allows the user to edit the bodies of these methods, but not their signatures.

The macro processor supports *Guard Variables* using a #IF# macro, which guards portions of method bodies. The macro-processor supports *List Placeholders* and *Extended Placeholders* using a single *#Extended#* macro that has the parameter name as an argument. When this macro is found by the macro-processor it calls the same Java code as was called by Javadoc to compute a *String* replacement for the macro.

3.4 Tool support: Meta-CO_2P_2S

Creating a generative design pattern is not a simple task. The parameters for a pattern must be specified and a framework must be created. To support graphical tools like CO_2P_2S, the pattern's graphical components must also be created and assembled so they can be displayed on a user interface.

To simplify this process, we provide tool support in the form of Meta-CO_2P_2S. One goal of this meta-programming environment is that its generative patterns should be *first-class*. That is, they should be indistinguishable in form and function from those patterns supplied with CO_2P_2S. We do not want to create a dichotomy of generative patterns: those supplied by the tool builders and those created by pattern designers using Meta-CO_2P_2S. The pattern designer's ability to create new patterns should not be hindered in any way.

Meta-CO_2P_2S provides dialogs for defining each parameter in a generative pattern, as well as other information that the pattern user never sees. For convenience, parameters are grouped into categories. The "Class Names" category is one example. Each class is assigned a role as structural or user-accessible. Structural classes are internal framework classes that a user does not normally need to access. User-accessible classes, which export hook methods that the user must implement, are flagged so that these files can be preprocessed before being presented to the user. For example, CO_2P_2S converts these files to HTML, as explained later in Section 4.4. The *@editable* tab in the source code template indicates which methods in these classes can be changed by the user.

User interface descriptions are in the "Constants" and "GUI Configuration" categories. Some of the constants are label strings that appear in the pattern description or in menu items. The GUI configuration describes the layout of the graphical pattern representation, including the image that shows the pattern structure. The appropriate image is selected by concatenating parameter values, and using the resulting text as the name of a GIF image. This provides a simple mechanism for updating graphics without needing to write display code.

Finally, the "Parameters" category describes the generative pattern parameters. Each parameter can be one of the parameter types listed earlier in the paper. Each item in this category fully describes the parameter, including its type, necessary extensions, and its set of legal values.

Meta-CO_2P_2S bundles the parameter information and GUI attributes together and stores the information in a file with standard XML format as described in Section 4.

4. Generative pattern representation

This section focuses on the representation of the different components of our generative design patterns (parameters and frameworks) and shows that this representation is not tied to a particular programming system. We also show how it is turned into a generative pattern form suitable for use in programming tools, of which CO_2P_2S is just one example.

The exact requirements of this representation depend on the particulars of the programming system. However, one of the goals of this representation is to be system-independent so that patterns for any number of tools can be generated from the same pattern description.

4.1 Pattern parameter representation

The parameter descriptions must be stored for use in programming systems. The parameter representation consists of two parts. The first part is a description of the parameter, including its type (Section 3.1). This description uses XML format that has the benefit of tool-independence. In addition, the format of an XML file can be verified using a DTD (Document Type Definition) file, so that a custom verifier is not required. Finally, XML has a number of system-independent support tools for parsing and manipulating files.

The second part of the parameter description is an XML description of the graphical elements needed to enter the parameter values. Text-based tools can ignore this part of the parameter description. This XML is not intended for humans. It should be generated by any pattern design meta-tool and read by any pattern adaptation tool.

4.2 Automating parameter value entry

The descriptions of the parameters for a generative design pattern must be stored so that they can be incorporated into a pattern adaptation tool. However, during adaptation, pattern users must have some mechanism for assigning parameter values. The mechanism for parameter value entry is actually a property of each parameter and that mechanism should be generated from the pattern description.

A *String* parameter can be entered in a standard dialog box with a single text field. A specialized *String* parameter, like an *Identifier* or *Class Name* can be verified before the typed text is accepted. An *Enumeration* parameter is entered by a set of labels or graphical images and associated radio buttons. A *List* parameter uses a list pane, and buttons for adding and removing elements. Of course when an add button is pressed, a dialog that is appropriate for the element type is launched.

Other parameter types may require more work because their structure is more ad-hoc. Although it is possible for an *Extended* parameter to be completely arbitrary, this is unusual. Rather, an extended parameter tends to be text that contains logically related information that cannot be easily captured as a set of choices. As a result, simply providing a text field for these parameters may not be the ideal choice. Instead, the pattern designer may find it better to create a customized dialog that allows users to easily enter the parameter value and that allows the data to be verified more easily.

As well as providing additional help for obtaining parameter values from the pattern user, it is necessary for a pattern designer to provide code generator extensions for interpreting the *Extended* parameter values and generating the appropriate code based on them. The dialog that gathers the parameter value can have an effect on how easy it is to perform this task. Using the method signature example, consider inserting a call to the method in the framework structure. With a table, we can easily generate a call statement by printing out the signature without the argument types. If the signature was a line of text, the parser would need to determine the function of each token in the text and print it out accordingly.

User interface and code generator extensions are possible because the CO_2P_2S interface is itself a framework. The reflective nature of Java allows these extensions to be instantiated and used when necessary. Part of the description of the parameters indicates the necessary extensions for each parameter, for obtaining the parameter values from the user and processing them for code generation.

4.3 Framework representation

We need a format for storing this framework code that facilitates the transformations described earlier. We refer to this family of frameworks as a *source code template*. In general, this template is a set of annotated source code files, where the annotations indicate how the generative pattern parameters alter the code. It includes all of the code for all possible variations of the pattern that can be specified using the pattern parameter values. A code generator performs a source code to source code transformation using the template and the parameter values. The result is a framework that implements the adapted pattern structure.

As described in Section 4.3, our current code generator is based on Javadoc and a Doclet. This means that some code is actually generated procedurally instead of residing in a declarative template. Each *Extended* parameter can be viewed as a behavioral component. It is possible to create a library of these reusable behavioral components to support parameter sharing between patterns.

4.4 From pattern descriptions to generative patterns to frameworks to programs

The relationships among all of the parts of the generative pattern description are shown in Figure 2. The pattern description, created using Meta-CO_2P_2S and stored using the format described earlier, holds all of the information needed by a programming tool to incorporate the generative pattern.

A pattern generator processes the parameter descriptions and (optionally) the graphical representation and outputs a generative pattern, in an analogous manner to the way frameworks are constructed from a source code template and pattern parameters. An adaptation tool incorporates the new generative pattern (discussed later) and makes it available to programmers. When a programmer selects the generative pattern, the tool obtains values for the parameter values based on their descriptions. The code generator described in the previous section takes these parameter values and the source code template and produces a customized framework implementing the selected pattern structure. The framework consists of structural code and hook methods. The user creates an application by supplying application-specific bodies for the hook methods exported by the framework.

Our pattern generator is implemented using XML and XSL (Extensible Stylesheet Language). An XSLT processor processes the parameter descriptions, written in XML, using the XSL stylesheet. The output is a tool configuration and associated graphical components for the user interface. The programming system incorporates the new generative pattern by including the configuration information, and can then provide the pattern to its users. If the tool has a graphical user interface, then the graphical components can also be used. These components consist of images and labels that are displayed in the interface so the user can visualize the state of the generative pattern during application development.

In CO_2P_2S, the tool configuration generated by the pattern generator is a set of Java classes that augment the user interface. The classes generated by the pattern generator use this framework to provide user interface and code generator support for the new generative pattern. These classes are compiled and are subsequently included in the tool using dynamic class loading in Java.

Other tools may require different configuration information from the pattern generator to include new generative patterns. For example, a tool may use a specification language to write programs, and not have a graphical user interface. Generating configuration information is a matter of providing a tool-specific XSL stylesheet for the pattern description. However, the description does not need to be changed. Once created, this description can be used in any tool that provides a stylesheet for the pattern generator (the XSLT processor).

Unfortunately, extensions to the user interface, specification language, and code generator will be system-specific until the parameter type system is completed. In addition, interface support for combining these types into parameters will be necessary for each system that wants to use our generative design pattern description.

4.5 Evolving generative design patterns

We have shown an example use of the generative Composite pattern. This pattern can support a variety of traversal operations for common cases. However, there are some problems that this pattern does not address.

For example, the traversal methods always visit the complete structure of the composite. For some problems, this may be inefficient. To support applications where partial traversals are necessary, the Composite pattern can be extended to include guard and continuation methods that control the traversal. These two methods are similar to the prefix and suffix methods except that rather than being part of the traversal, they dictate the flow of control through the composite structure.

The guard method determines if the children of a composite object should be traversed. It has the same arguments as the original method. If the guard evaluates to *true*, then the composite object executes its optional prefix method, traverses its children and then executes its optional suffix method. If the guard evaluates to *false*, the traversal method returns immediately. If the traversal operation has a return value, the default value is returned.

The continuation method is evaluated before each child object is visited. Like the guard method, it takes the same arguments as the traversal method. If this method returns *false* after traversing any child, the remaining children are not traversed. Regardless, the prefix and suffix methods for the composite object are always executed if they are present.

To provide this functionality, the pattern designer has a choice. The original generative Composite pattern can be extended, adding new parameters and augmenting the generated framework code, or a new pattern based on it can be created. Pattern flexibility must be balanced against ease-of-use. The more parameters that a generative pattern presents, the more flexible it will be. However, with too many parameters, the pattern becomes difficult to specify and use. Since we have no general solution to this problem, we have considered this balance on a pattern-by-pattern basis.

To provide guard and continuation methods, we created a new generative pattern, the GuardedComposite. We copied the original Composite description and then used Meta-CO_2P_2S to make the changes. We added two new *Boolean* sub-parameters to each traversal method, indicating the presence of the guard and continuation methods. We also added two more *Method Name* sub-parameters to record the names of the guard and continuation methods, if they exist. This approach is similar to using prefix and suffix methods. Recall that the description of a generative pattern consists of an XML file for parameter descriptions, graphical images, and possibly some tool extensions. Once copied, the parameter descriptions were edited using Meta-CO_2P_2S to add the new parameters. The copied source code template for the framework was modified as well. Finally, the interface and code generator extensions were augmented. The user interface needed to obtain values for the new parameters for the generative GuardedComposite pattern, and the code generator needed to be altered to generate correct code for the guard and continuation methods in the traversal methods of the composite classes.

A more sophisticated approach would be to derive a new generative pattern from an existing pattern, using an inheritance mechanism. This would support reuse of parts of a pattern description and source code template.

5. Summary

We have presented an approach to generative design patterns that incorporates three new ideas:

1. Generative design patterns are defined by a set of typed parameters with specific legal values and a code template that generates frameworks whose structure depends on combinations of values for these parameters.

2. Design pattern adaptation is a three-phase process where the first phase involves parameter value selection, the second phase involves framework code generation and the third phase involves framework specialization.

3. A two part tool-independent representation of generative design patterns consists of an XML-based representation of pattern parameter values and a code template with simple parameter-based annotations.

We have described two tools, CO_2P_2S and Meta-CO_2P_2S that support our process and that have be used to adapt and generate design patterns. In addition, our generated code does not suffer from the performance penalties that plague many "automatic programming" approaches. In fact, our generated code has been successful in the performance-conscious domain of parallel programming [1][14]. For example, CO_2P_2S has been used to generate efficient solution code [2] for all problems in the Cowichan problem set. This problem set is used to evaluate problem coverage and performance in parallel programming systems [18].

CO_2P_2S currently supports six design patterns for parallel computing: mesh, wavefront, pipeline, search-tree, distributor and phases. It also supports sequential patterns: composite, decorator, abstract factory, chain (tree) of responsibility and observer. However, Meta-CO_2P_2S can be used to quickly add new design patterns.

Any programmer who wants to use a generative design pattern to write an application, follows four steps:

1. Pick an appropriate set of design patterns. One approach is to use a pattern language [15].

2. Use a tool like CO_2P_2S to adapt design patterns to an application by selecting values for the pattern parameters.

3. Press a button to generate frameworks for each design pattern that has been customized for your application.

4. Use framework specialization to finish the application.

We have also described a new approach to generative design pattern construction. To create a new generative design pattern, a pattern designer follows four steps:

1. Find an existing descriptive design pattern that applies or create a new descriptive design pattern.

2. Analyze the descriptive design pattern with emphasis on the: participants, implementation issues and sample code, to identify pattern parameters that could affect the structure of the framework code you will need. Study existing generative design patterns for similar issues and parameterization requirements. If possible, find an existing generative design pattern that is similar and edit it, instead of creating a new one.

3. For each parameter that was identified in 2), determine the legal parameter values by considering the necessary parameter values for your application (you may need multiple adapted copies with different values) and the known uses section of the pattern documentation. Also consider legal parameter values of similar parameters in other generative design patterns and use existing parameter types as a guide.

4. Construct a code template that spans the legal parameter values defined in 3). Use other generative design patterns to guide you, or if possible, edit an existing generative design pattern to take advantage of the code template that has already been written. Use the parameter types to guide the construction of your code template. If possible, use a tool like Meta-CO_2P_2S to simplify the construction of behavioral components.

Writing parameterized design pattern code can be hard. We need strong tool support and in this paper we have described some support that we do provide. However, we need more tools. For example, we need tools to check for coverage. In other words, did the pattern designer provide code for all legal parameter value combinations? We need support for testing. For example, we need a tool to generate test application code that exercises all combinations of legal parameter settings. Creating new programming languages that support higher levels of abstraction is a useful goal. However, tools have become the new programming languages of modern software development. We believe that the future of mastering generative design patterns lies in open representations and tools that support them.

6. References

[1] J. Anvik, et.al. Generating parallel programs from the wavefront design pattern. In High-Level Parallel Programming Models and Supportive Environments, Ft. Lauderdale FL., CD-ROM April 2002.

[2] J. Anvik. Asserting the utility of CO_2P_3S using the Cowichan problem set, Master's thesis, Department of Computing Science, University of Alberta, 2002. In preparation.

[3] K. Beck and R. Johnson. Patterns generate architecture. ECOOP, Vol. 821 of LNCS, pp. 139-149. Springer, 1994.

[4] J. Bosch. Design patterns as language constructs. JOOP, 11(2), pp. 18-32, 1998.

[5] S. Bromling. Meta-programming with parallel design patterns. Master's thesis, Department of Computing Science, University of Alberta, 2002.

[6] F. Budinsky, M. Finnie, J. Vlissides, and P. Yu. Automatic code generation from design patterns. IBM Systems Journal, 35(2), pp. 151-171, 1996.

[7] TogetherSoft Corporation. TogetherSoft ControlCenter tutorials: Using design patterns. www.togethersoft.com/services/tutorials/index.jsp.

[8] A. Eden, Y. Hirshfeld, and A. Yehudai. Towards a mathematical foundation for design patterns. Technical Reprort 1999-04, Department of Information Technology, University of Uppsala, 1999.

[9] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In ECOOP, Vol. 1241 of LNCS, pp. 472-495. Springer, 1997.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[11] R. Johnson and B. Foote. Designing reusable classes. JOOP, 1(2), pp. 22-35, 1988.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In ECOOP, volume 2072 of LNCS, pp. 327-353. Springer, 2001.

[13] S. MacDonald. From Patterns to Frameworks to Parallel Programs. Ph.D. thesis, Department of Computing Science, University of Alberta, 2002.

[14] S. MacDonald, D. Szafron, J. Schaeffer, and S. Bromling. Generating parallel program frameworks from parallel design patterns. In Euro-Par, Vol. 1900 of LNCS, pp. 95-104. Springer, 2000.

[15] M. Massingill, T. Mattson, and B. Sanders. A pattern language for parallel application programs. Technical Report CISE TR 99-022, University of Florida, 1999.

[16] M. Pollack. Code generation using javadoc. www.javaworld.com/javaworld/jw-08-2000/jw-818-javadoc_p.html, August 2000.

[17] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Vol 2. Wiley, 2000.

[18] G. Wilson. Using the Cowichan problems to assess the usability of Orca. IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systemsy, 183-193, 1994.

[19] ModelMaker Tools. Design patterns in ModelMaker. www.modelmakertools.com/mm_design_patterns.htm.