University of Alberta

Types and Code Generation for use in Generative Design Patterns

by

Patrick Earl ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2004

Your file Votre référence
ISBN: 0-612-95740-3
Our file Notre référence
ISBN: 0-612-95740-3

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.
Revised(4) Report on the Algorithmic Language Scheme

To Ema, Helle, and Lorne.
Without you, this would not have been possible.

Acknowledgements

Many thanks go to my supervisors, Duane Szafron and Jonathan Schaeffer, for their patience and guidance. I greatly appreciated the time I had to spend with each of them. Thanks to Jonathan for correcting a mistake in my writing that prompted me to learn to use semicolons; I appreciate knowing. Thanks to Duane for the interesting discussions about types and for the enjoyable movie and pizza parties.

Thanks to the staff and students at the University of Alberta. I enjoyed my time in your presence. Thanks also to Dominque Parker, with whom I had many interesting discussions and badminton games in the systems lab.

My family deserves many thanks for their support and encouragement on my journey. Thanks go to my grandmother, Ema, for her love and for the seed of money she planted so long ago. My parents, Helle and Lorne, provided not only nourishment, but continued love and encouragement. Thanks also go to my sister, Debra, who is wonderful in general.

Finally, I wish to thank God for the opportunities and gifts that have been given to me. I appreciate His patience and love for me.

Contents

List of Figures

Chapter 1

Introduction

Software developers are continually looking for better ways to create software. Reliability, reusability, and decreased development time are common goals in the search for new techniques and tools.

Design patterns have increased the level of abstraction at which a programmer can create software [18]. These patterns of design, though not new in themselves, have now been defined and recorded. These pattern descriptions allow developers to speak a common language and more easily define the building blocks of their software.

As useful as these recorded design patterns were, the element of code was still missing. Generative design patterns were created to fill the gap between the design pattern description and the source code required by the application. A generative design pattern system gathers a specification from the application developer and creates a customized pattern for use in the application. Generative systems allow for flexibility not present in static frameworks or libraries [25]. This additional flexibility is needed to create true multi-purpose design pattern implementations.

CO_2P_2S[1] [22] is one such generative design pattern system. In its original form, CO_2P_2S made use of hand-coded pattern specifications in its generation process. MetaCO_2P_2S was later created to provide tool support for building patterns. MetaCO_2P_2S provided GUI tools and a limited set of types that eased the pattern creation process. Yet, many patterns still required substantial amounts of custom Java code and an understanding of the internal MetaCO_2P_2S framework. There were also shortcomings and limitations in the code generation system, making the creation of pattern templates more difficult than neccesary.

The primary goal of this thesis research is to simplify the process of building a generative design pattern within CO_2P_2S. This involved the creation of a system of types and a new code generator. In a paper on CO_2P_2S and MetaCO_2P_2S [22], the authors discuss the use of numerous small types for pattern creation. The original system never fully realized that

[1]CO_2P_3S stands for Correct Object-Oriented Pattern-based Parallel Programming System. Since CO_2P_2S also handles sequential design patterns, Parallel has been dropped from the name.

vision, in part due to the limitations of the code generator. The system implemented in this thesis reaches the level where many small types can be combined in useful ways to produce complex design patterns. The result is a system that simplifies the task of creating generative design patterns.

1.1 Motivation and Goals

A generative design pattern system, such as CO_2P_2S, is more useful when it supports a large number of patterns. The easier it is to create those patterns, the more patterns there will be. The system should provide room for growth, making it easy to support new patterns. MetaCO_2P_2S provided some support for pattern creation, and this research aims to extend that support, making the building of patterns easier than before.

There are numerous smaller goals and tasks that are part of the primary goal of easier pattern creation. The amount of code that the pattern designer has to write should decrease. Additional reuse of existing code is beneficial. Reducing the potential for errors is another goal. In the old CO_2P_2S, creating GUIs for patterns was somewhat difficult; this should be improved.

Techniques for reducing the complexity and length of the generative design pattern's source code are desired. There are many ways this can be done, from simple macros, as in the C preprocessor, to advanced data structures and mathematical operations.

In [23], the author points out that some systems focus on code size and quality, but fail to address the ease of use of the system. When the system is difficult to learn and use, the resulting benefits are decreased. With this in mind, simplicity and ease of use are also goals for the new system.

1.2 Related Work

There has been extensive work on type systems, generative programming, and design patterns. Presented here is a brief summary of related work.

TXL [15] is a general purpose source-to-source transformation system. While TXL itself is not well suited to simple code generation, the μC language [16], implemented on top of TXL, is a better fit. Despite being easier to use, μC requires the user to specify the types of the source elements that are being manipulated. As an example, a loop that creates a list of variable declarations is given a type of `[declarations*]`. The need to learn the types of textual elements in the system detracts from the simplicity of the system.

The μC paper [16] also provides an example of an "ideal" metalanguage. The language demonstrated is simple, and though not complete, following its guidance helps maintain simplicity when creating a code generation language.

GenVoca [10] uses a layered approach to creating customized components. Each layer in the system is responsible for an individual aspect. The layers are composed to create components with different feature sets. Though simple in concept, this technique is not sufficient for all generative design pattern tasks.

Frame-based systems, such as ANGIE [1], are focused on combining parameterized code blocks. This approach seems well suited to general purpose generative programming and elements of it are used in this research.

Other generative programming systems [16, 29] use declarative logic to combine facts and direct code generation. Because of the graphical nature of the CO_2P_2S system, the format of the facts is of less importance. The generation systems presented already [1, 10, 15, 16, 29] use a textual interface to provide configuration information to the system. While this is sufficient for code generation purposes, the user of the CO_2P_2S system should not need to learn a language to compose a design pattern that meets his/her specifications.

Transformational systems [17] (for an example, see [14]) provide powerful techniques for manipulating source code. However, with these powerful techniques comes a steeper learning curve. Instead of just specifying what needs to be produced, mechanisms for selecting source elements and transforming them must be learned.

Czarnecki's book [17] provides a more comprehensive overview of the various generative techniques and languages.

The most applicable related work is described in "Automatic Code Generation from Design Patterns" [13]. The presented architecture uses three layers to generate design patterns from user specifications. The first of these layers is a web interface. The web interface displays information about a particular design pattern and allows the user to enter options that customize the pattern source code to the user's application. A layer of Perl [5] scripts maps the input from the user into output suitable for use in the code generator. The code generation layer is based around simple operations such as conditional inclusion, repetitive inclusion, textual replacement, and code segment reuse. Because the code generation language is simple, additional functionality must sometimes be provided by the mapping layer or by customized operators, implemented as external processes in a language of choice. Since this system requires the user to cut and paste code into their application, it is difficult to incorporate changes to a pattern configuration into the application. Without that ability, it is more difficult to experiment with different options. The CO_2P_2S system, in combination with the CO_2P_2S code generator, provides the ablity to reconfigure options and test different pattern configurations easily. The use of web pages as a medium for pattern configuration means that the user interfaces may be inflexible. The server-based architecture requires a server running to handle the requests. Since CO_2P_2S is a standalone application, it does not suffer from either of those difficulties.

1.3 Contributions

Through this thesis research, the following contributions were realized.

- A system of types was created for use in generative design patterns. These types are object-oriented and are manipulated through a GUI type editor, which was also created during the course of the research.
- As well as the type system itself, many types were created for use in constructing generative design patterns.
- To make use of the types, the CO_2P_2S Generation Language (CGL) was created. This meta-language, in combination with the type system, provides a flexible platform on which to create design patterns.
- The existing tools, CO_2P_2S and MetaCO_2P_2S, were rewritten to take advantage of the new system of types and code generator. The result is a complete system for working with generative design patterns. Through the new CO_2P_2S system, generative design patterns are easily created and utilized.

The contributions of this research have improved the CO_2P_2S system. Though not well tested, we expect that the pattern creation process will now be easier, faster, and more reliable.

1.4 Overview

Chapter two provides additional background, including an overview of the original CO_2P_2S and MetaCO_2P_2S. The second chapter also provides further motivation for the research presented in this dissertation. Chapter three is an overview of the new type system, using the Decorator pattern as an example. Chapter four provides more details on the type system, including some of the design decisions in its creation. Chapter five covers the new code generator and the meta-language it uses. Chapter six provides a comparison of the old and new systems and concludes the dissertation.

Chapter 2

Overview of CO_2P_2S

2.1 Design Patterns

A design pattern is an abstraction of a commonly used programming technique. Design patterns can be found across a wide variety of application domains. Patterns are often defined by high-level descriptions [18], separate from any source code. To use one of these design patterns in an application, the pattern must be translated from the high-level description into actual code that can be inserted into the application.

Some design patterns are simple, perhaps describing a way that objects in a GUI can relate to an underlying data model. Other patterns are more complex, such as a pattern describing a parallel computation on a mesh. By the very nature of patterns, they can be used under a variety of circumstances. Adapting the pattern to fit the circumstances is usually performed manually. In addition, design patterns may appear in code, but they may not be recognized due to poor documentation.

CO_2P_2S was created to automate the use of existing design patterns. This is particularly useful for complicated patterns such as the parallel mesh computation. In addition to generating code for a pattern, CO_2P_2S implicitly documents the patterns that are used in an application.

2.2 CO_2P_2S

The CO_2P_2S system is composed of two components, CO_2P_2S and MetaCO_2P_2S (Figure 2.1). MetaCO_2P_2S is used by a pattern designer to create a description of a pattern. CO_2P_2S is then used, by a programmer, to instantiate that pattern for use in an application. The pattern designer needs to create a single pattern template, yet that same template can be used in many applications.

The main screen of the CO_2P_2S user interface is shown in Figure 2.2. On the left side, there is one icon for each of the available patterns. To make a pattern available, the programmer must first import that pattern using one of the menu items.

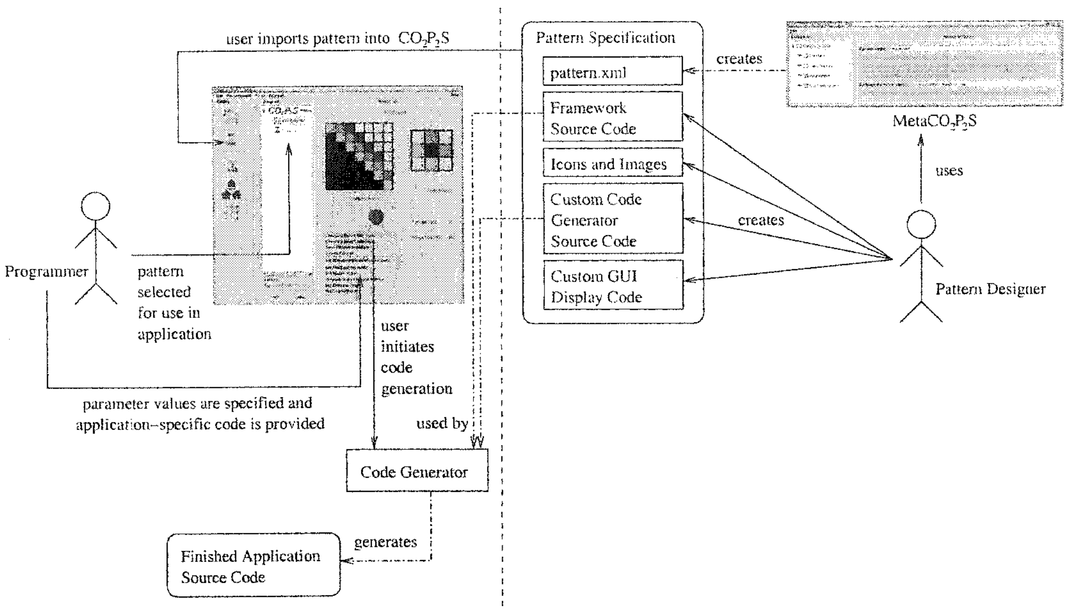

user imports pattern into CO_2P_2S
Pattern Specification
pattern.xml
Framework Source Code
Icons and Images
Custom Code Generator Source Code
Custom GUI Display Code
creates
$MetaCO_2P_2S$
uses
creates
Pattern Designer
Programmer
pattern selected for use in application
user initiates code generation
parameter values are specified and application–specific code is provided
used by
Code Generator
generates
Finished Application Source Code

Figure 2.1: CO_2P_2S User Interaction (Adapted from [12])

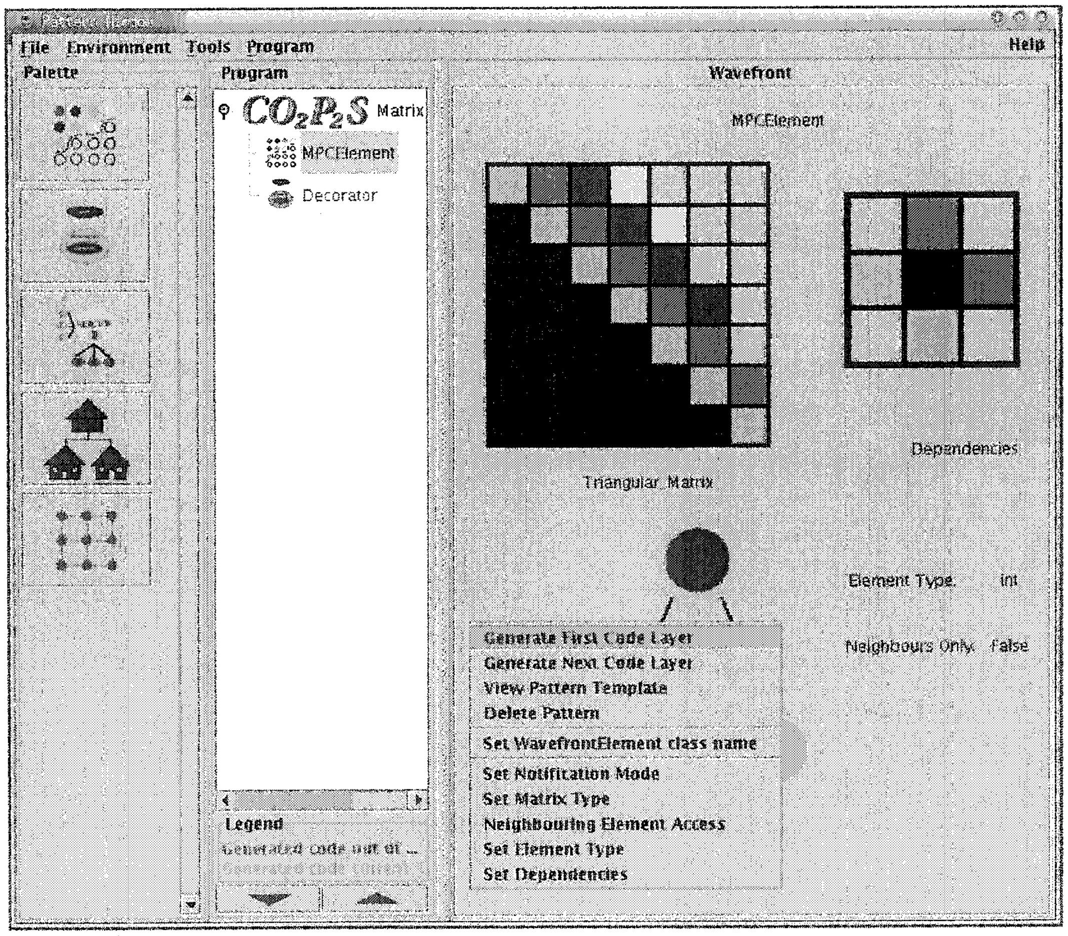


Figure 2.2: CO_2P_2S GUI

To create an application in CO_2P_2S, the programmer starts by creating a "New Program." Applications are composed of patterns and user classes. To add a pattern to an application, the programmer selects the pattern from the "Palette." The same pattern may be added multiple times if desired.

Once the pattern has been added to the program, the programmer must adapt the pattern to the application's requirements by setting the parameter values available via the menu shown. For example, an application may need the parallel wavefront pattern. The wavefront pattern performs a computation that flows across a matrix of cells. In the wavefront, the programmer must set the class name of the wavefront element, as well as five other parameters that control how the wavefront code will be generated. Changes to the parameters are reflected visually in the pane on the right.

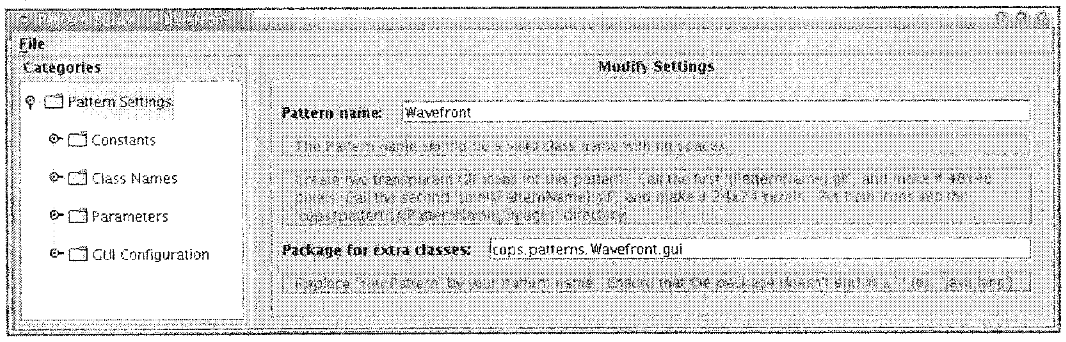
Once the parameters have been set to values appropriate for the application, the programmer will choose "Generate First Code Layer." The code generator will use the programmer's parameter choices to create an adapted pattern template. In some patterns, the programmer will specialize the pattern further by inserting application-specific code into methods or classes in the pattern.

After application-specific code is added or parameter values are changed, the code generator can process the code again. This allows the programmer to modify parameters even after the code has been generated for the first time. This feature is useful in performing tasks such as performance evaluation, and can even allow for changes in the application that the programmer did not anticipate.

2.3 MetaCO_2P_2S

To use a design pattern in CO_2P_2S, the pattern designer must first create the specification for that pattern. MetaCO_2P_2S was designed to assist in the creation of this specification. Though patterns are created once and used many times, a pattern system with only a few patterns is not very useful. It is therefore critical that adding new patterns to the system be as easy as possible. The goal of the research described in this dissertation was to replace MetaCO_2P_2S with a simpler, more efficient, and more powerful generative design pattern creation system. The rest of this section describes the earlier MetaCO_2P_2S system. A more complete description can be found in [12]. A view of the earlier version of MetaCO_2P_2S is helpful in understanding the problems that needed to be solved in this research project. In the earlier version of MetaCO_2P_2S, a pattern consisted of the following components:

- A system-independent description of the pattern, stored as XML. This description is the primary output of MetaCO_2P_2S.
- Annotated framework source code which will be processed by the code generator.



Pattern name: Wavefront

The Pattern name should be a valid class name with no spaces.

Create two transparent GIF icons for this pattern. Call the first '{PatternName}.gif', and make it 48x48 pixels. Call the second 'small{PatternName}.gif', and make it 24x24 pixels. Put both icons into the [illegible] directory.

Package for extra classes: cops.patterns.Wavefront.gui

Replace 'YourPattern' by your pattern name. [illegible]

Figure 2.3: MetaCO$_2$P$_2$S General Pattern Description

MetaCO$_2$P$_2$S is able to output skeleton framework files from the pattern description.

- Custom Java code to display special types of parameters and to generate the code associated with them.
- Pattern documentation, stored as HTML.
- The images used for displaying the pattern in CO$_2$P$_2$S.

2.3.1 MetaCO$_2$P$_2$S Pattern Description

The overall description of the pattern created by MetaCO$_2$P$_2$S contains the following elements (see Figure 2.3):

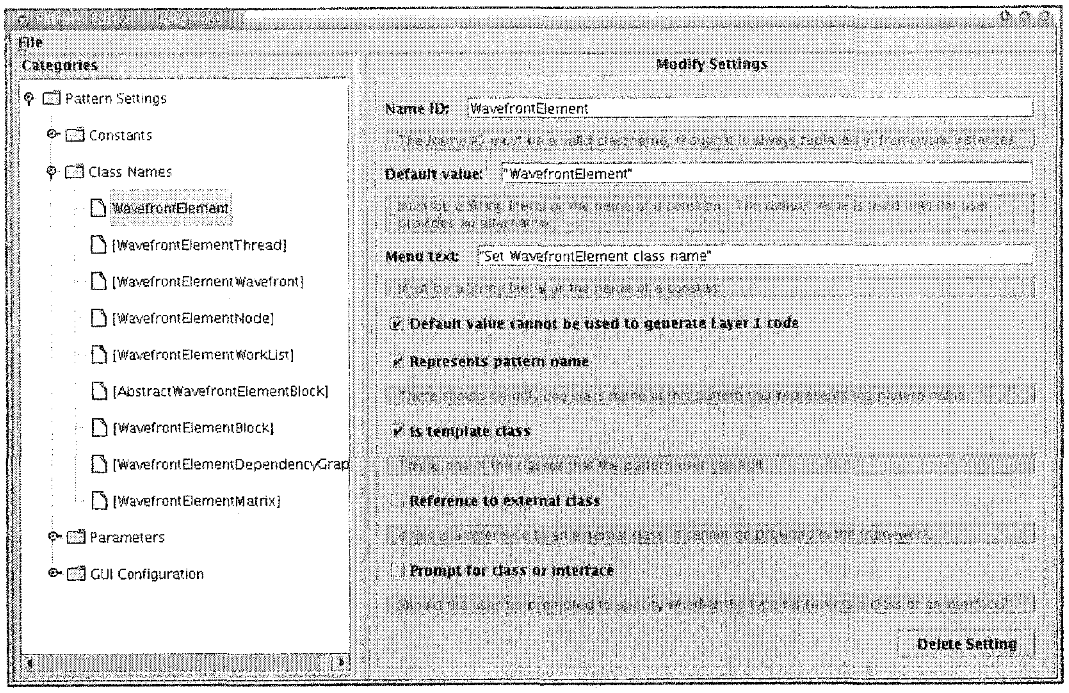
- The name of the pattern.
- The Java package where custom code is located.
- A list of string constants which my be used in place of string literals in other MetaCO$_2$P$_2$S fields.
- The list of all of the classes contained in the pattern.
- The list of parameters used in the pattern.
- A description of the pattern's GUI representation.

The last three items in the above list will be explained in more detail.

Class Names

MetaCO$_2$P$_2$S requires the pattern designer to specify all of the classes that the pattern contains. These classes can be either programmer-named classes (Figure 2.4) or framework classes (Figure 2.5).

Creating a programmer-named class allows the programmer to specify a name for that class. A default value for the class name may be provided. If the pattern designer specifies



File

Categories

- Pattern Settings
  - Constants
  - Class Names
    - WavefrontElement
    - [WavefrontElementThread]
    - [WavefrontElementWavefront]
    - [WavefrontElementNode]
    - [WavefrontElementWorkList]
    - [AbstractWavefrontElementBlock]
    - [WavefrontElementBlock]
    - [WavefrontElementDependencyGrap
    - [WavefrontElementMatrix]
  - Parameters
  - GUI Configuration

Modify Settings

Name ID: WavefrontElement

The Name ID must be a valid classname, though it is always replaced in framework instances

Default value: "WavefrontElement"

Must be a String literal or the name of a constant. The default value is used until the user provides an alternative.

Menu text: "Set WavefrontElement class name"

Must be a String literal or the name of a constant.

[✓] Default value cannot be used to generate Layer 1 code

[✓] Represents pattern name

There should be only one class name in this pattern that represents the pattern name.

[✓] Is template class

This is one of the classes that the pattern user can edit.

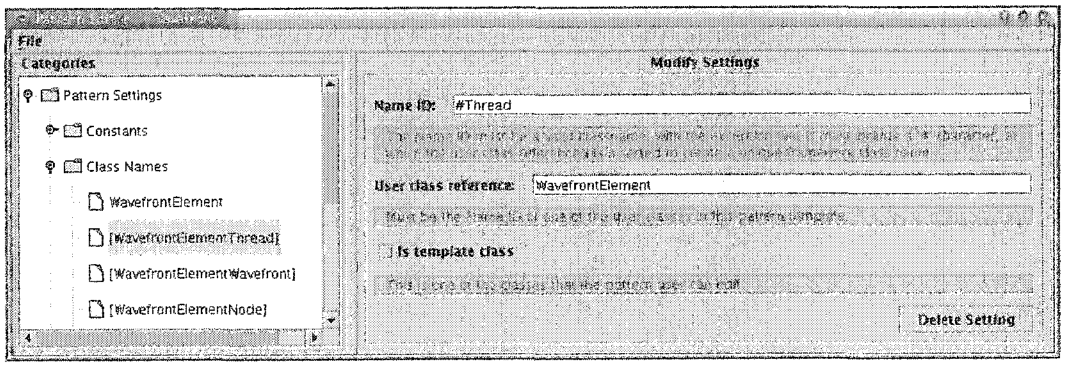
[ ] Reference to external class

If this is a reference to an external class, it cannot be provided by the framework.

[ ] Prompt for class or interface

Should the user be prompted to specify whether the type represents a class or an interface?

Figure 2.4: MetaCO$_2$P$_2$S Programmer-Named Class



File

Categories

- Pattern Settings
  - Constants
  - Class Names
    - WavefrontElement
    - [WavefrontElementThread]
    - [WavefrontElementWavefront]
    - [WavefrontElementNode]

Modify Settings

Name ID: #Thread

[illegible]

User class reference: WavefrontElement

Must be the Name ID of one of the user classes in this pattern template.

[ ] Is template class

This is one of the classes that the pattern user can edit.

Figure 2.5: MetaCO$_2$P$_2$S Framework Class

that this class "Represents pattern name," then the GUI will use this information to update the title of the pattern shown in CO_2P_2S. Some classes in the framework may need to be edited by the programmer. Other classes may define part of the interface to the generated framework that the programmer needs to know about. If the class should be visible to the programmer for editing or viewing, the pattern designer needs to select "Is template class."

Programmer-named classes can also be used to specify classes and interfaces that are not included in the framework source. An example of this would be allowing the programmer to specify a superclass that would be referred to by some class in the framework. To indicate that a given class refers to a class that is not part of the framework itself, the pattern designer would choose "Reference to external class." If the external reference may also be an interface, that alternative is accommodated by choosing "Prompt for class or interface."

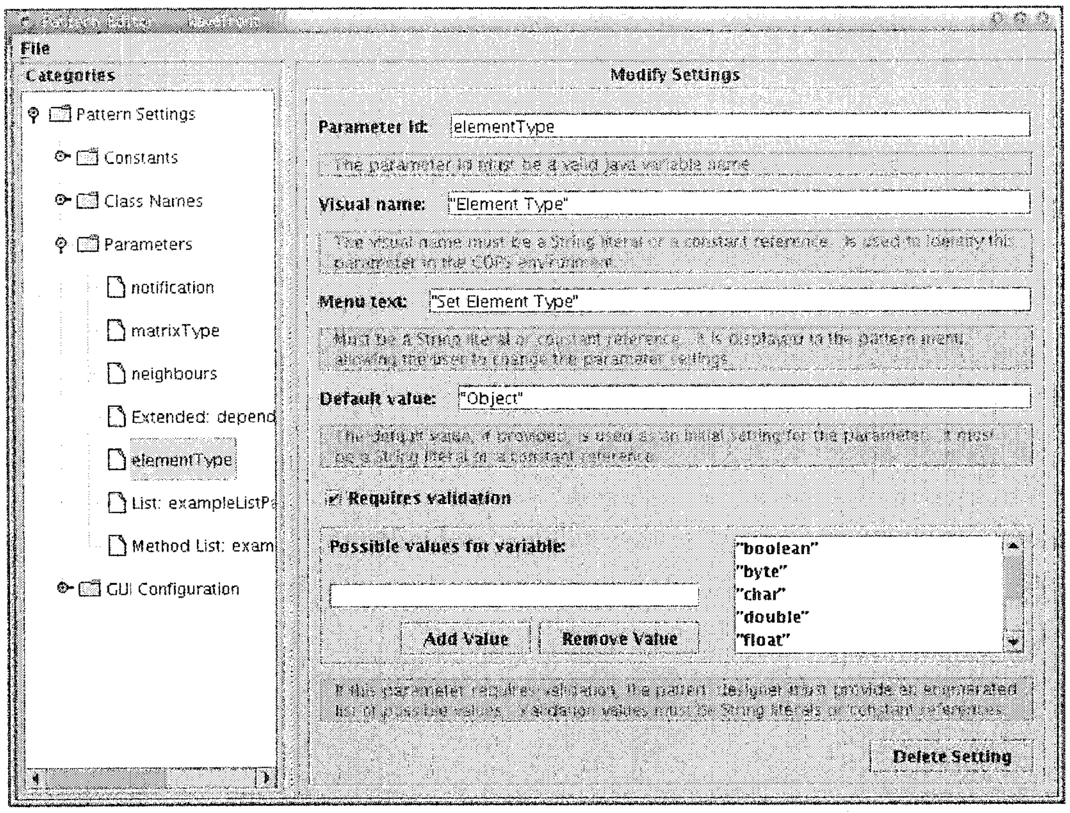
The other type of classes, framework classes, do not have their names specified by the programmer. Instead, the name is built by concatenating extra characters to the name of an existing programmer-named class. Just as with programmer-named classes, the pattern designer has the option of making the class visible to the programmer as a template class.

Parameters

MetaCO_2P_2S allows the pattern designer to specify one of four types for each of the parameters within the pattern. Along with the type-specific information, each parameter has an ID. The ID is used to identify this parameter within the context of the code generator. All parameters also have a "visual name" and "menu text" associated with them. The "menu text" provides the name of the CO_2P_2S menu item for the given parameter. The "visual text" is displayed, along with the value of the parameter, to indicate the current state of the pattern.

Basic (Figure 2.6): To the code generator, a basic parameter is simply a string. MetaCO_2P_2S allows the designer to specify a default value for the basic parameter. Basic parameters can also be "validated." When "requires validation" is selected, the pattern designer must provide a list of possible values for the parameter. Normally, a basic parameter provides a text field into which an arbitrary string can be entered. If validation is used, a set of buttons is provided, one for each possible value.

Extended (Figure 2.7): For complex parameter types, an extended parameter must be defined. Extended parameters are defined entirely by a Java class provided by the pattern designer. This class must contain methods that specify what code is generated (see Section 2.4.3 for details). The class must also provide a Java GUI that will be used to display the parameter in CO_2P_2S. Because the Extended parameter is contained within a single class, there is no formal definition of the configuration data used by the parameter. The class provided will save the parameter as an arbitrary string and



Modify Settings

Parameter Id: elementType

The parameter id must be a valid java variable name

Visual name: "Element Type"

The visual name must be a String literal or a constant reference. Is used to identify this parameter in the CO2P2S environment.

Menu text: "Set Element Type"

Must be a String literal or constant reference. It is displayed in the pattern menu, allowing the user to change the parameter settings.

Default value: "Object"

The default value, if provided, is used as an initial setting for the parameter. It must be a String literal or a constant reference.

[✓] Requires validation

Possible values for variable:

"boolean"
"byte"
"char"
"double"
"float"

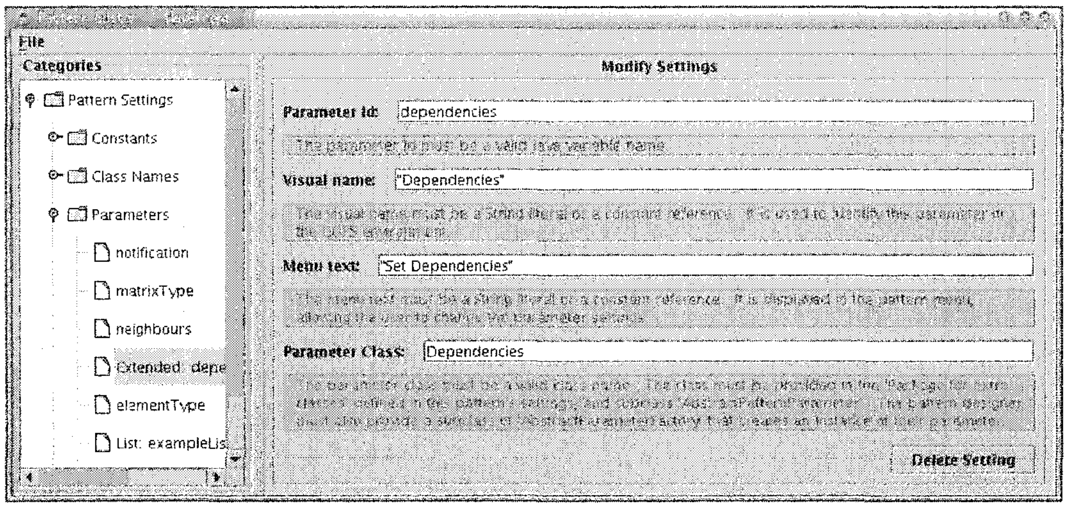
If this parameter requires validation, the pattern designer must provide an enumerated list of possible values. Validation values must be String literals or constant references.

Figure 2.6: MetaCO$_2$P$_2$S Basic Parameter Type

load the parameter by parsing that arbitrary string. In this manner, the CO$_2$P$_2$S GUI and code generator are tied together.

List (Figure 2.8): This type is a list of basic or extended parameters. The pattern designer must provide a Java class that contains the code generation methods (same methods as the extended type). The class should be a subclass of `PatternListParameter`. Inside the class, the designer must also implement stub methods that act as an interface to a provided list GUI. The provided list GUI shows a list of simple strings. Items in the list can be edited or reordered by clicking on various buttons. If the list contains extended parameters, the pattern designer must also provide a class representing the individual list elements.

Method List (Figure 2.9): The method list type provides a list of method signatures. These signatures can be obtained either by entering them through the CO$_2$P$_2$S GUI or by importing them from a class file. A Java class which represents an individual method element must be provided. This class controls the code generation and the GUI dialog used for each element in the method list. The pattern designer can also choose to make the method signatures that are imported from classes immutable.



File

Categories

- Pattern Settings
  - Constants
  - Class Names
  - Parameters
    - notification
    - matrixType
    - neighbours
    - Extended: depe
    - elementType
    - List: exampleLis

Modify Settings

Parameter Id: dependencies

The parameter Id must be a valid Java variable name.

Visual name: "Dependencies"

The visual name must be a String literal or a constant reference. It is used to identify this parameter in the [illegible]

Menu text: "Set Dependencies"

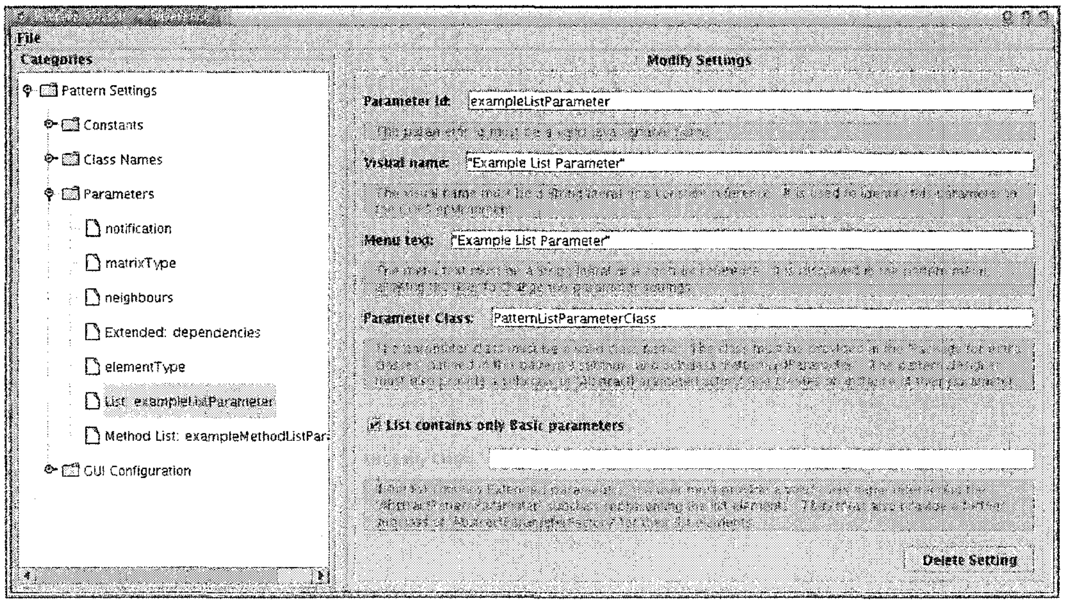
The menu text must be a String literal or a constant reference. It is displayed in the pattern menu, allowing the user to change the parameter settings.

Parameter Class: Dependencies

The parameter class must be a valid class name. The class must be provided in the Package for extra classes defined in the pattern settings, and subclass "AbstractPatternParameter". The pattern designer must also provide a subclass of [illegible] that creates an instance of their parameter.

Delete Setting

Figure 2.7: MetaCO$_2$P$_2$S Extended Parameter Type



File

Categories

- Pattern Settings
  - Constants
  - Class Names
  - Parameters
    - notification
    - matrixType
    - neighbours
    - Extended: dependencies
    - elementType
    - List: exampleListParameter
    - Method List: exampleMethodListPar
  - GUI Configuration

Modify Settings

Parameter Id: exampleListParameter

[illegible]

Visual name: "Example List Parameter"

[illegible]

Menu text: "Example List Parameter"

[illegible]

Parameter Class: PatternListParameterClass

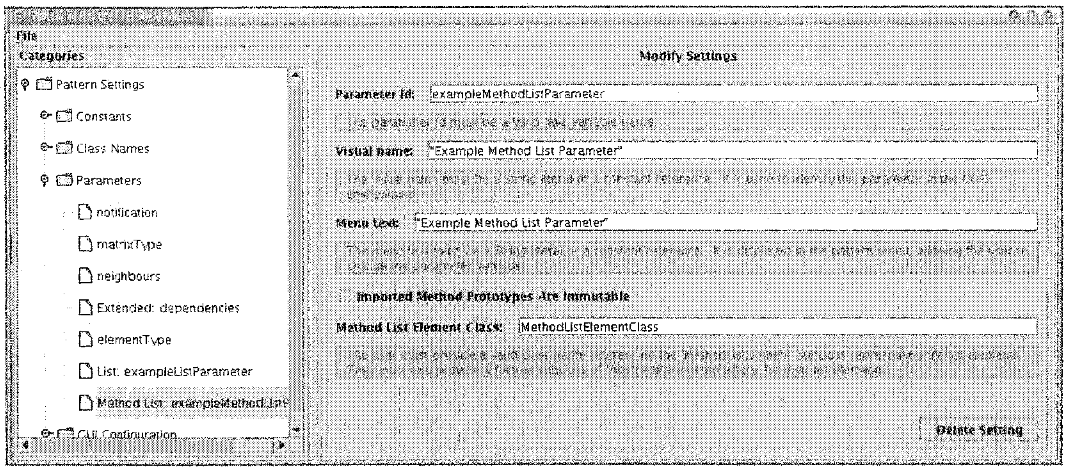
[illegible]

☑ **List contains only Basic parameters**

[illegible]

Delete Setting

Figure 2.8: MetaCO$_2$P$_2$S List Parameter Type



Parameter id: exampleMethodListParameter

Visual name: "Example Method List Parameter"

Menu text: "Example Method List Parameter"

Imported Method Prototypes Are Immutable

Method List Element Class: MethodListElementClass

Figure 2.9: MetaCO_2P_2S Method List Parameter Type

GUI Configuration

MetaCO_2P_2S also allows the pattern designer to specify a GUI representation of the pattern. The GUI representation can consist of text and graphic elements. Both text elements and image elements can be either static or dynamic. For each GUI element, the pattern designer must specify a valid Java variable name, and the absolute (x, y) location of the element.

For static text, a fixed string of text will be displayed. For dynamic text, the contents of the text to be displayed will be taken from either the name of a class or the value of a parameter. A default value is also provided by the pattern designer. The default value is used when the text element is static, or when the programmer has not specified the value associated with a dynamic text element.

For a graphical element, one or more images are provided by the pattern designer. To determine which image to display, a list of "Image Name Parts" is concatenated together to form the name of the image. Each "Image Name Part" can be a fixed string or a reference to a parameter.

2.4 Javadoc Code Generator

After a pattern is created with the help of MetaCO_2P_2S, CO_2P_2S can then be used to create a concrete instance of the pattern. The programmer used CO_2P_2S to provide values for each of the parameters that were specified in MetaCO_2P_2S. Based on those parameter values, the resulting source code changes. To specify what changes are needed, the pattern designer marks up the source code with special directives. These directives, combined with the parameter values from the programmer, provide the information the code generator needs to produce the concrete Java source.

Javadoc [4] was chosen to make the job of parsing easier. Javadoc allows you to define

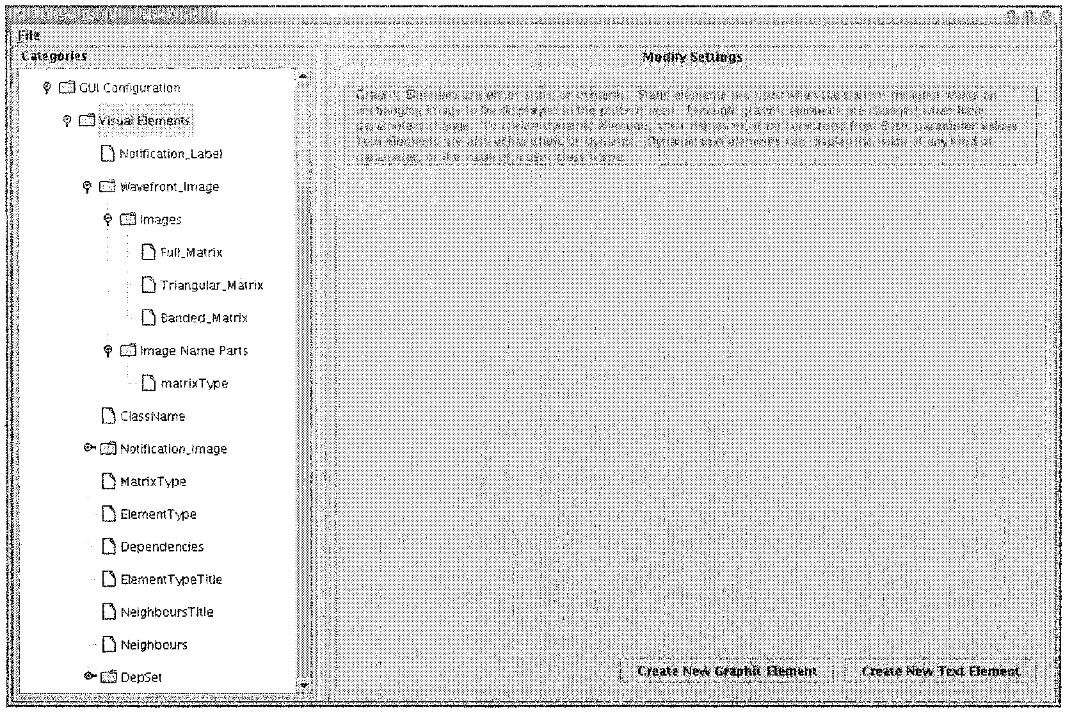


Figure 2.10: MetaCO_2P_2S Configuration of Visual Elements

arbitrary tags for Java language elements such as classes, methods, and fields (see Figure 2.11).

The Javadoc doclet produces output in two forms. One of these is an HTML form that is used within the CO_2P_2S GUI, allowing the programmer to provide custom code to be inserted at indicated locations in the code. This custom code may be in the form of bodies of hook methods. The other form of output is the finished source code that is ready to be compiled. Each Java template file will be transformed into a compilable Java source file. The HTML form is only output if the source is editable in some manner. In the HTML output, necessary HTML headers and footers are added and special characters such as $<$ and $>$ are escaped.

Instead of taking a source file and modifying it, the doclet reads the entire source file into internal data structures and then outputs a new version of the file based on those data structures. Javadoc does not parse method bodies or initial values for fields. Because it does not track those in its data structures, it is impossible to reproduce them purely through Javadoc. In Figure 2.11, you can see that `method()` has no body. With CO_2P_2S, the body for this method must be stored in a separate file, which would be loaded by the doclet and used as the method's body.

When filling in method bodies, the code generator checks to see if the programmer has supplied a custom method body for the method. If one has been provided, it is used in lieu of the default method body supplied by the pattern designer. If the method is not a

```
/**
 * This is a Javadoc comment with a single tag that directs the code generator
 * to allow the programmer to add custom imports to this class.
 *
 * @userImports
 */
class FrameworkCLASS_MyClass {
    /**
     * This method would only be included in the resulting source if the
     * needMoreMethods parameter was set to the "yes" value.
     *
     * @parameter needMoreMethods yes
     */
    void method() { }
}
```

Figure 2.11: Javadoc Tag Example

constructor, it may be an abstract method or a method declaration inside of an interface. In those cases, no method bodies will be present.

Having Javadoc tags alone is not sufficient for code generation, since method bodies cannot easily be modified. As a result, a small macro language was defined that supports conditional inclusion and limited macro expansion. Javadoc tags and the macro language will be described in the next two sections.

2.4.1 Javadoc Tags

What follows is a complete list of the Javadoc tags that are supported by the code generation doclet.

Class Tags

These tags apply to the entire class.

@userImports A single set of programmer-defined Java imports may be defined for the entire pattern. This tag causes the code generator to provide a way for the programmer to enter custom imports.

@importImports This tag is used to include the single set of imports, defined by `@userImports`, into other files.

@frameworkSuperclass <param> The current class will inherit from the super class or interface specified by `<param>`.

@noConstructors If this tag is present, no constructors will be written. If a class has no constructors, Javadoc will insert an empty default constructor. This tag allows for more control over the generated constructors by inhibiting the empty default constructor.

@userCodeAllowed Allows the programmer to add custom methods and fields to the current class. There is one block of programmer code allowed per class.

@extParameter <param> The extended parameter `<param>` has Java code associated with it that allows it to insert additional fields and methods. The interface to this Java code is described in the Extended Parameters section. The methods that are called as a result of the `@extParameter` tag are `getCodegenMethods()` and `getCodegenFields()`.

Method Tags

These tags apply to individual methods or constructors.

@editable Makes the method editable by the programmer. A default body can be provided that will be used if the programmer does not specify a custom body.

@parameter <name> <value> Given the name of a parameter and a literal value, this field is used to determine whether the following method should be rendered. If the parameter called `<name>` is set to `<value>`, then the method will be rendered. If multiple parameters with the same name are given, the results of the comparisons are combined using OR semantics. If more than one parameter name is provided, the results for the individual parameters are combined using AND semantics.

@parameter <param> <value>, <param> <value>, ... To specify conditions in which parameter values depend on each other, this multi-parameter syntax can be used. In this case, the results of the comparisons are combined using AND semantics. If multiple occurences of this multi-parameter syntax are present, the results of those are combined using OR semantics.

Field Tags

These tags apply to individual fields.

@initialValue <value> Since Javadoc does not parse the values that fields are initialized to, this tag was introduced to provide the means to specify the initial value of a field. It is also worth noting that this workaround does not support the `static` { `field = computation()` } technique for initializing fields.

@parameter Same behaviour as for methods.

2.4.2 Macro Language

A small macro language is used to process method bodies and initial values of fields. The supported operations are listed below.

```
#FrameworkMACRO#(<param> <op> <value>)
<optional code>
#FrameworkMACROend#
```

`<op>` is either == or !=. If the condition is satisfied, the `<optional code>` will be output.

```
#FrameworkPARAM_<param>#
```

In this case, `<param>` must be an extended parameter. The `getCodegenBody()` method of that parameter is called to retrieve the code to be inserted at the location of the macro. `getCodegenBody()` takes two arguments. For methods, the arguments are the name of the class and the name of the method. For fields, the arguments are the type of the field and the name of the field. This macro is used for inserting an arbitrarily complex code fragment inside a method body or field initializer.

```
#FrameworkCLASS_<class>#
```

Inserts the programmer-defined name for the class.

2.4.3 Extended Parameters

MetaCO$_2$P$_2$S has a number of built-in types. When those types are not sufficient, an extended parameter must be created to perform the code generation. An extended parameter must follow the interface laid out in the AbstractPatternParameter class. The following methods from AbstractPatternParameter are relevant to code generation.

```
public abstract List getCodegenMethods( String className );
```

For a class with the `@extParameter` tag, this method will be called. This method can insert full methods by returning a list of `CopsMethod` instances. Each `CopsMethod` contains standard items such as the name of the method, the argument list, and the list of exceptions the method can throw. It also contains a data structure that allows the type creator to specify if the method should be rendered or not. The data structure allows the same conditional expressions as the `@parameter` tag.

```
public abstract List getCodegenFields( String className );
```

This method is also called when `@extParameter` is used. It inserts additional fields by accessing a list of `CopsField` instances. Aside from the differences between fields and methods, its use is very similar to that of the `CopsMethod` class. Note that both `CopsField` and `CopsMethod` inherit from `CopsConstruct` which provides the common functionality.

```
Javadoc Code Generator:

    sb.append("if("+test1+" && "+test2+") {\n");
    sb.append("}\n");

Ideal:

    if(@test1 && @test2) {
    }
```

Figure 2.12: Java Code Generation

`public abstract String getCodegenBody( String className, String method );`

If `#FrameworkPARAM_<param>#` appears within a field initializer or a method body, this method will be called to generate the code for that particular location. The location of the code is identified using the class name and the method signature. Without knowledge of the internals of the code generator, it is not possible to insert multiple code fragments in any particular method body.

```
public abstract String saveToString();
public abstract void loadFromString( String str );
```

Each parameter knows how to save itself to a string and load itself from a string. Unfortunately, since each parameter specifies its own saving and loading technique, there is no standard format for the output of the parameter values. This creates a strong coupling between the code generator and the CO_2P_2S code that stores the state of the parameters.

2.4.4 Limitations of Code Generator

Overall, the Javadoc code generation method lacks flexibility and adds unecessary complexity to the work of the pattern designer. Some tags, such as `@noConstructors` and `@initialValue`, exist solely to work around limitations in the Javadoc approach. This approach needed to be simplified.

With the Javadoc code generator, there is no way to exclude a file from being processed. Even if a class is not needed in a particular configuration of a pattern, it will still be produced. As well, there is no way to produce multiple files from a single template file. This is inefficient and restrictive.

To produce anything other than simple code, Java code must be written that assembles the code as a string. An example of how this obfuscates the code is shown in Figure 2.12. A better technique was required to simplify what the pattern designer needed to write.

Javadoc does not parse method bodies or field initializers. This created the need for the small macro language that was used in addition to the Javadoc tags. Also, because of this, the methods are stored in individual files which are later loaded by the code generator. Having the code spread out into many tiny files makes it more difficult to understand as a whole.

The files required for each pattern were previously distributed throughout the CO_2P_2S directory hierarchy. As part of this research, all of the files within a single pattern were placed within a single directory, making it easier to see which files belong to a particular pattern.

It is difficult to maintain consistency between files. Similar code must be duplicated even when there is little or no variation. A technique for combining similar blocks of code is desired.

2.5 Summary

Pattern-based tools will only succeed if they have a rich, extendable set of patterns. The easier it is to create those patterns, the more likely it is that a tool will succeed. MetaCO_2P_2S, as presented in this chapter, was a first attempt at a pattern builder. Though it did improve the process of designing patterns, it was awkward and difficult to use. The research presented in the following chapters has changed that.

Chapter 3

Type System Example

Within a design pattern there are options[1] that can be modified to change how the code for the pattern is generated. As a simple example, we could have a pattern whose purpose is to support the storage of a collection of numbers. These numbers could be either integers or floating point numbers. If we choose integers, the code for the container class would be different than if we chose floating point.

The type system within CO_2P_2S is the mechanism by which the pattern options are created and used. Each option in a pattern has a particular type. That type defines characteristics such as the GUI interfaces, the code generated for the type, and the structure of the data within the type.

The new CO_2P_2S type system is composed of four main layers (see Figure 3.1).

Type Editor Creation and modification of types occurs at this level. The types must be created before they may be used within MetaCO_2P_2S.

MetaCO_2P_2S Patterns are defined in MetaCO_2P_2S. The patterns have options which represent the variability within the pattern's code. Each of these options has a type.

CO_2P_2S Within CO_2P_2S, patterns are selected by the user and the values of the pattern's options are defined.

Code Generator The code generator combines the user-defined option values and the annotated framework code into application-specific source code.

This chapter will provide an example that examines the first three layers, starting from CO_2P_2S. The Code Generator layer will be discussed in Chapter 5. The example that is used is the Decorator design pattern.

[1]Note that in Chapter 2, these options were known as parameters. Parameters within the pattern are now known as options. Also note that pattern options may now be parameterized, bringing the parameters term back under a different meaning.

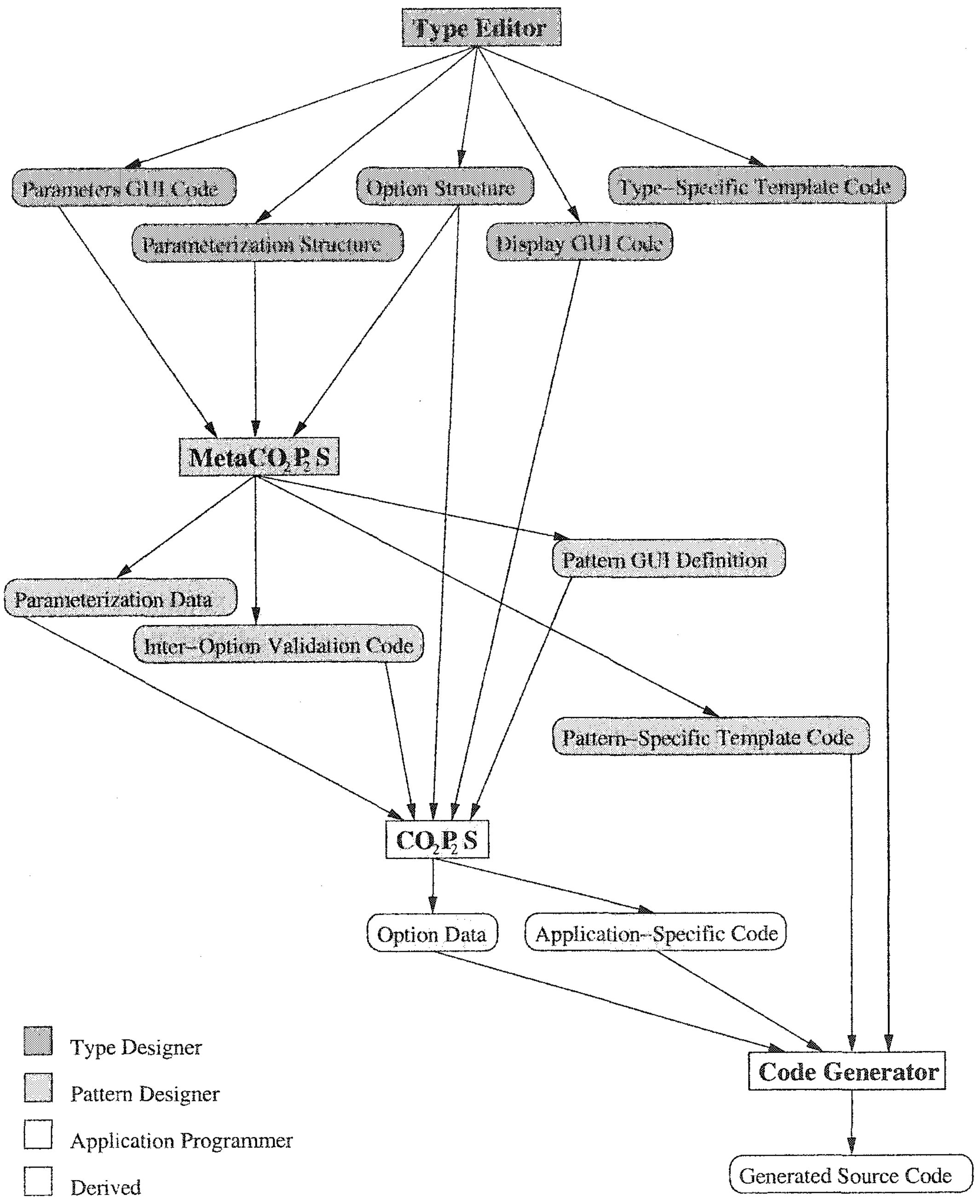


Figure 3.1: Type System within CO_2P_2S System

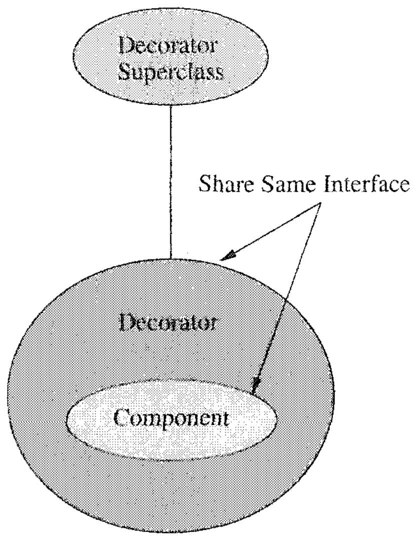


Figure 3.2: Decorator Pattern

3.1 Brief Overview of Decorator Pattern

The Decorator or Wrapper pattern is a simple structural design pattern [18]. It is used to modify the behaviour of an object at run-time by layering objects inside of each other. The object on the outside is known as the Decorator (see Figure 3.2). The object on the inside is known as the Component. The Decorator shares the same interface as the Component, so one acts as a drop-in replacement for the other. When the Decorator's method is called, it has a chance to perform operations and delegate to the Component's method as it chooses.

As an example, consider a system for drawing objects. We might have a circle object that has a `draw()` method. When called, it draws a black circle. Suppose we would like to draw the circle on a different type of background. We could have a `background` object that paints the entire canvas blue when its `draw()` method is called. Normally, we might call `background.draw()` and then `circle.draw()`, but what we actually want is to perform a single call to `draw()` and have the circle and background paint themselves. In this example, we want the background to act as a Decorator for the circle. The background would maintain a reference to the object that it decorates, in our case the circle. Inside background's `draw()` method, it will first paint the background and then call its components' `draw()` method. A yellow border around the canvas could be added in a similar manner. The border object would decorate the background object, which in turn decorates the circle. The programmer defines which components decorate which other components at run-time. We can see that although we are performing a number of different operations, we need only use the `draw()` call on the top level object. Note that the background, circle, and border all share the `draw()` interface, but with different code. Since the Decorator and Component share the same interface, they may be chained together in arbitrary configurations to perform complex tasks.

In Figure 3.2 there is a Decorator Superclass. This superclass is part of the Decorator pattern within CO_2P_2S, but is not an integral part of the Decorator pattern itself.

3.2 Example of Original System

For comparision, the code, that the pattern designer had to write, for the original Decorator pattern has been provided in Appendix A. The code will not be explained in detail since it is over 10 pages long and contains callbacks and references to more code that has not been listed. The point is that creating the decorator pattern for the original system was clearly a complicated endeavour.

3.3 Using the Decorator Pattern

We start by looking at the third layer of the type system, where the types are used by the application programmer. This section refers to the CO_2P_2S box from Figure 3.1.

When a CO_2P_2S user would like to use a pattern, they must first import it. This process is performed by selecting "Import Pattern" from the "Environment" menu (shown in Figure 3.3). The available patterns will be shown and the user selects the one they would like to use.

After the pattern is imported, it may be used in the creation of an application. To use the pattern, a programmer must perform the following steps.

1. Start a new program by selecting "New Program" from the "File" menu. The program must be given a name at that point.

2. Once the program has been created, the user selects the "Decorator" pattern from the palette on the left. An instance of the pattern is added for use within the program.

The decorator pattern is now ready to be configured (Figure 3.3). The right pane of the CO_2P_2S window shows a graphical representation of the Decorator pattern. In the graphical representation we can see the default values for four options that must be configured to use the Decorator pattern within an application:

- Decorator class name (`DecoratorPattern`),
- Decorator superclass name (`Object`),
- Component class name (`Object`), and
- Methods in the shared interface (none shown).

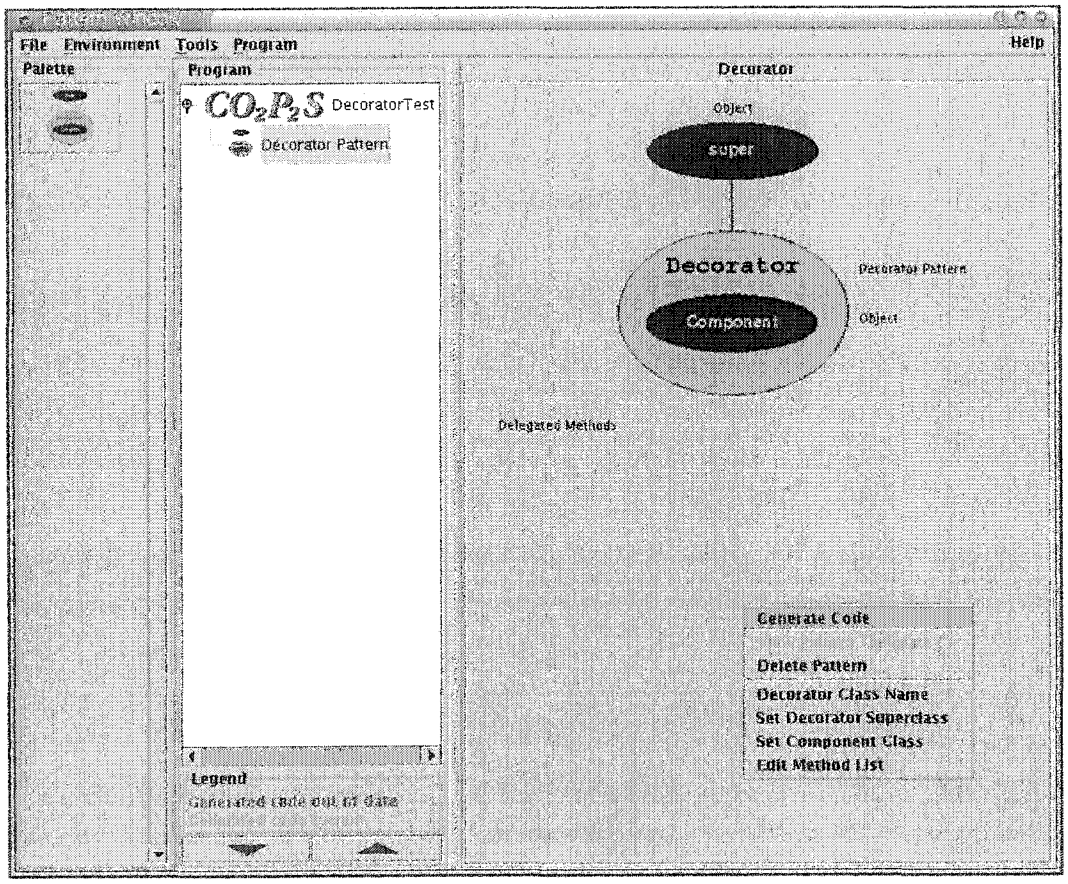


Figure 3.3: Decorator Pattern in CO_2P_2S

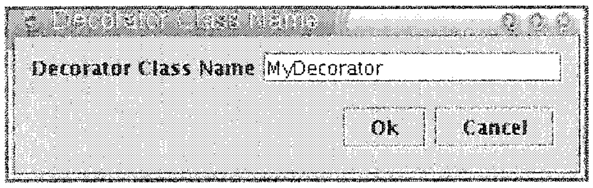


Figure 3.4: Decorator Class Name Dialog

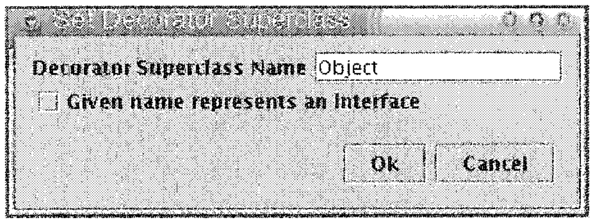


Figure 3.5: Decorator Superclass Dialog

The configuration process consists of setting the pattern options by using the menu shown in the bottom right of Figure 3.3.

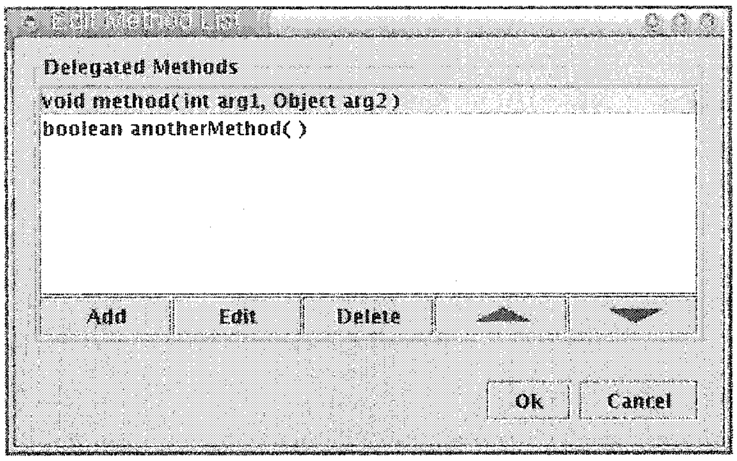
By selecting "Decorator Class Name" from the menu, we get the dialog box shown in Figure 3.4. A simple dialog box like this is also provided for the "Set Component Class" menu item. The dialog box allows the user to enter the desired class name. Validation is also performed to ensure that the entered string is a valid class name.

Setting the "Decorator Superclass" produces a similar dialog box (Figure 3.5). Since the superclass of the decorator may also be an interface, a checkbox is provided to indicate that.

The most complicated option in the Decorator pattern allows the programmer to define the shared interface by creating a list of methods (Figure 3.6). The list is manipulated using the buttons located near the bottom of the dialog. Since lists occur often, the ability to easily create lists of arbitrary items is desirable.

Individual methods in the list are edited with the dialog box shown in Figure 3.7. This dialog box contains a number of fields, including two sublists for specifying arguments used and exceptions thrown by the method. Due to the complexity of the dialog, it would be easier to compose the dialog from smaller well-defined building blocks.

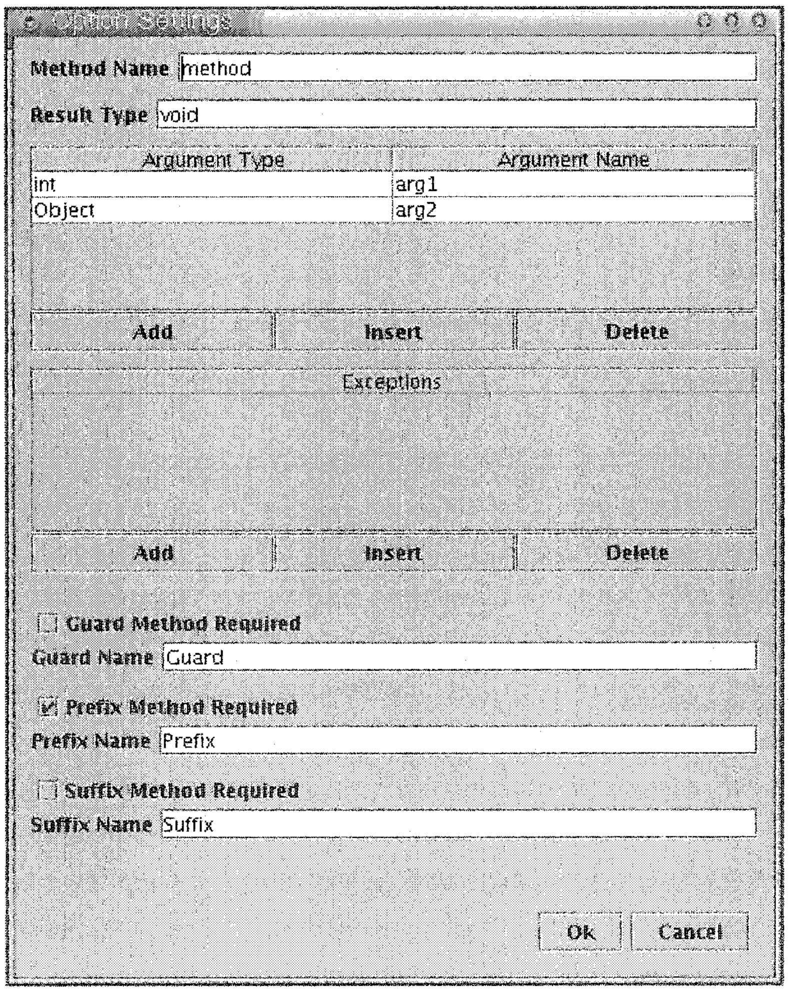
Once the four options are defined, the pattern is now ready to be generated. In Figure 3.8, we can see that the GUI has updated itself to display the current values of the pattern's options. All we need to do now is choose "Generate Code" to proceed. At that point, a last check is performed to ensure valid option values and the code is generated. For information on code generation, see Chapter 5.



Delegated Methods

void method(int arg1, Object arg2)
boolean anotherMethod( )

Figure 3.6: Decorator Method List Dialog



Method Name: method

Result Type: void

Argument Type	Argument Name
int	arg1
Object	arg2

[ ] Guard Method Required

Guard Name: Guard

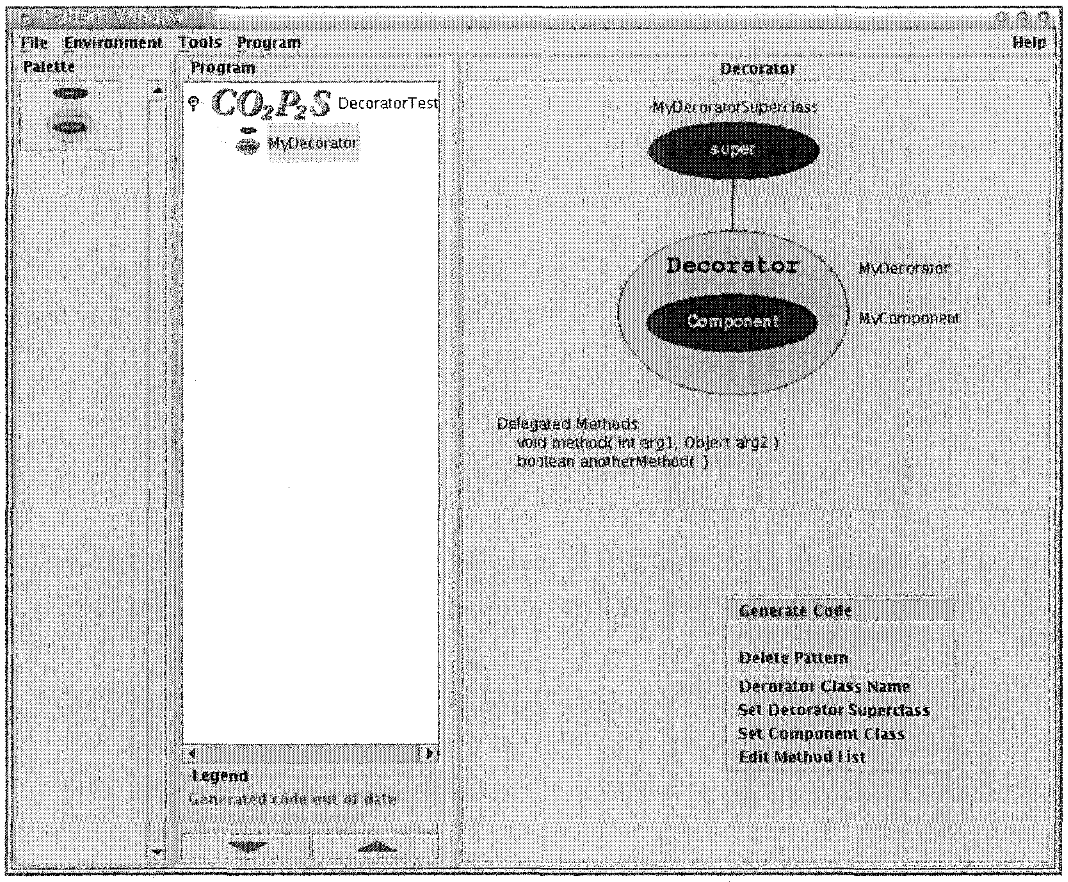
[✓] Prefix Method Required

Prefix Name: Prefix

[ ] Suffix Method Required

Suffix Name: Suffix

Figure 3.7: Decorator Method Dialog



```
Delegated Methods
    void method( int arg1, Object arg2 )
    boolean anotherMethod( )
```

Figure 3.8: Decorator Configuration Complete

3.3.1 Requirements CO_2P_2S places on the Type System

To make this process possible, there are a number of requirements that must be fulfilled by the type system:

- The structure of the data in the options must be defined, either implicitly or explicitly.
- The values manipulated by the GUI need to be initialized and validated.
- Each pattern option has a GUI associated with it that is used within CO_2P_2S. These GUIs may have a variety of widgets and may even spawn child GUIs.
- Support is required for the composition of larger GUIs from smaller GUI components.
- Support is required for lists of arbitrary items without having to create a complicated list GUI each time.
- In the graphical representation of the pattern, options need to be converted to a string form to be displayed. If the option is already in a form similar to a string, this conversion is trivial. For complicated structures like lists and records, more capabilities are needed.

3.4 Defining the Decorator Pattern

In the previous section we examined how the Decorator pattern was used from CO_2P_2S. The first step was importing the pattern's XML definition file into the CO_2P_2S environment. This pattern XML file was created by a pattern designer using MetaCO_2P_2S (see MetaCO_2P_2S in Figure 3.1). We will now examine how the Decorator pattern is created in MetaCO_2P_2S. In creating a pattern, the pattern designer must perform the following tasks:

1. Give the pattern a name.
2. Decide what options the pattern needs to support the variability present in specific applications.
3. Define a GUI representation for the pattern.
4. Create and annotate the framework source.
5. Create documentation for the pattern.

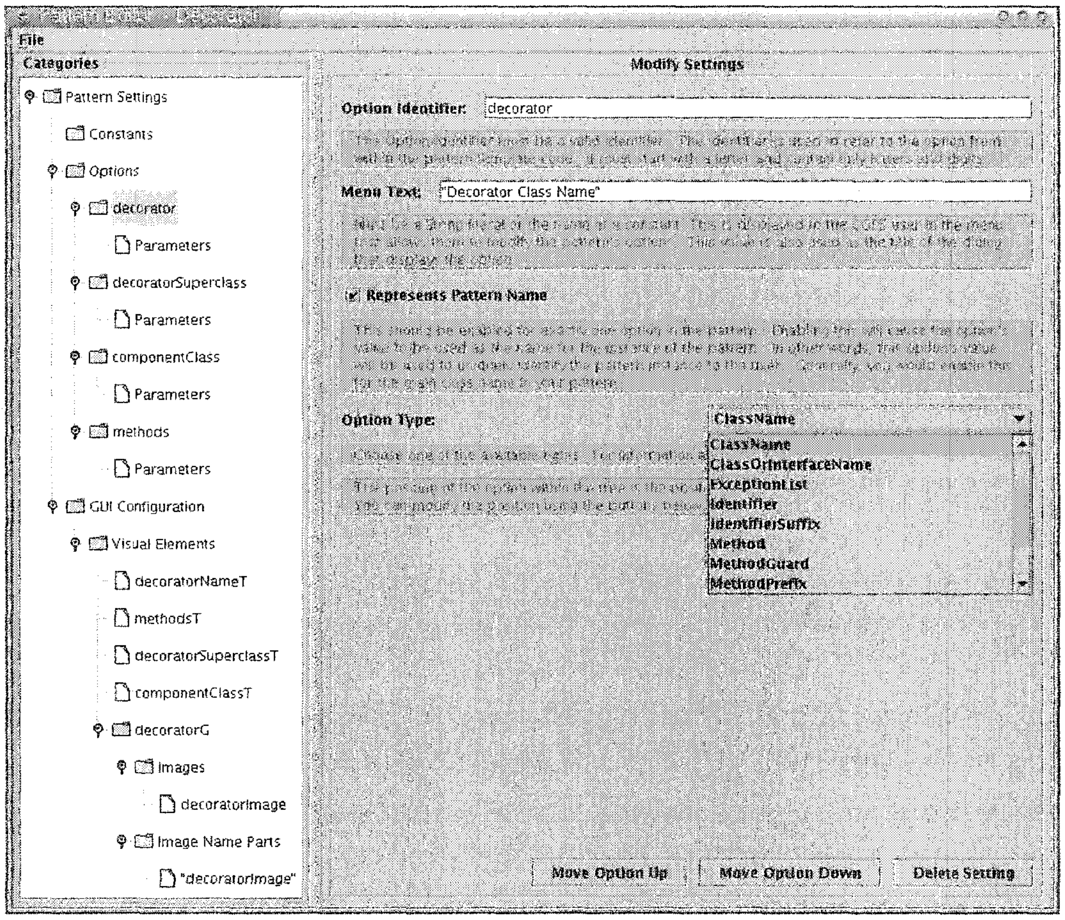
The code generation aspect will be discussed in Chapter 5. The format of pattern documentation will not be discussed since it is not relevant to this thesis.

The GUI representation for the pattern consists of graphic and text elements. Each of these elements can be either static or dynamic. A static element would be an image or string of text whose contents are fixed. Dynamic elements depend on the value of options within the pattern. Dynamic graphic elements can have their names constructed from the string values of one or more pattern options. Dynamic text elements represent the string value of a pattern option. Because the dynamic elements depend on a human readable string representation for the pattern options, it was necessary to provide a mechanism to generate custom strings from pattern options.

The most important part of the pattern definition in MetaCO_2P_2S is the options. Options come in all shapes and sizes. They may be as simple as a class name or as complex as a list of method signatures. In the original MetaCO_2P_2S, only a few option types were supported: string, enumeration, list, class name, and an arbitrarily complex extended type. The list and extended types involved writing substantial amounts of Java code to perform everything from the GUI display to the actual code generation. There was no real underlying structure to types like the extended type. The structure of the data was decided implicitly by the Java code that was written. Class names were not considered pattern options but were defined separately. This distinction was not needed and it also restricted the ability of the system to adapt which class files were generated. Though the use of extended options made the system flexible enough to handle most code generation, the creation of an extended type was a daunting task.

Instead of providing just a handful of types, the new type system provides a large variety of types that can be combined and used for most code generation tasks. Though somewhat pattern-specific types will need to be created in rare cases, most pattern options will now be handled through the larger variety of built-in types. Even when a pattern-specific type is needed, its creation is much simpler than the creation of an extended type.

Figure 3.9 shows the decorator pattern in MetaCO_2P_2S. On the left we can see the four options in the pattern. As well, a number of GUI elements are defined below. On the right side are the settings for the decorator class name pattern option. Every pattern option requires the pattern designer to provide these same settings. An identifier must be created which will be used later for code generation. As we saw in CO_2P_2S, each option is associated with a menu item. The text for that menu item is specified on that panel as well. There is a checkbox called "Represents Pattern Name." Since a program can have multiple patterns of the same type, there must be some way of distinguishing between them. When an option is set to represent the pattern name, its string value is used to identify the pattern instance to the CO_2P_2S programmer. The pattern designer must also choose a type for the option. All the types that were available when CO_2P_2S was started are listed in a combo box for easy selection. Note also that options can be moved up and down. Their order affects the order that the options appear in the menu. Generally a pattern designer would group similar options such as class names together.



Option Identifier: decorator

This Option Identifier must be a valid identifier. The identifier is used to refer to the option from within the pattern template code. It must start with a letter and contain only letters and digits.

Menu Text: "Decorator Class Name"

Must be a String literal or the name of a constant. This is displayed to the CO2P2S user in the menu that allows them to modify the pattern's options. This value is also used as the title of the dialog that displays the option.

[✓ Represents Pattern Name]

This should be enabled for exactly one option in the pattern. Enabling this will cause the option's value to be used as the name for the instance of the pattern. In other words, the option's value will be used to uniquely identify the pattern instance to the user. Generally, you would enable this for the main class name in your pattern.

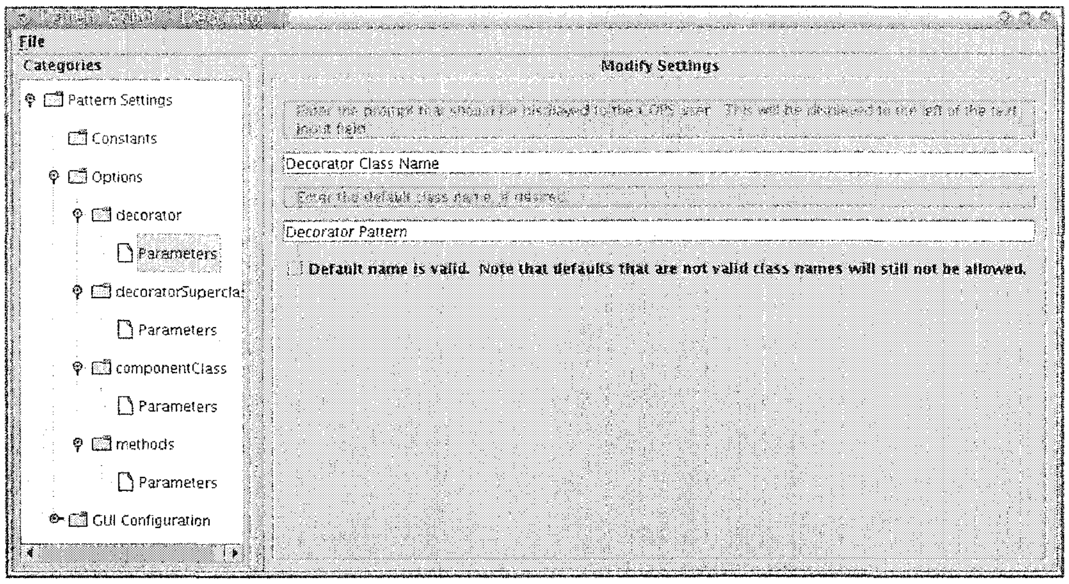
Option Type: ClassName

ClassName
ClassOrInterfaceName
ExceptionList
Identifier
IdentifierSuffix
Method
MethodGuard
MethodPrefix

Choose one of the available types. For information a

The position of the option within the tree is the order
You can modify the position using the buttons below

Figure 3.9: Decorator Pattern in MetaCO$_2$P$_2$S



Enter the prompt that should be displayed to the CO2P2S user. This will be displayed to the left of the text input field.

Decorator Class Name

Enter the default class name, if desired.

Decorator Pattern

☐ Default name is valid. Note that defaults that are not valid class names will still not be allowed.

Figure 3.10: Option Parameters for ClassName Type

In addition to selecting a type, the pattern designer must configure the type's parameters. Parameters are needed to add flexibility to the types without forcing the user to create numerous types with nearly the same function. An excellent example of this is an enumeration type. An enumeration can be thought of as a string with a constrained set of values. Instead of creating a new type for each set of values, we just have a single enumeration type and allow the pattern designer to set the possible values as parameters of the type. Parameters are also used to configure things like the labels in the GUI and what validation occurs. The parameterization provided is different from the parameterization provided by generics. With generics, you might be able to create a list of X, where X is a parameter. In the CO_2P_2S type system, you would need to create a list of only one type, but the parameterization might allow you to specify details of how that list would appear in CO_2P_2S.

In Figure 3.10, the parameters for the `ClassName` type are shown. In this case the parameters are used to configure the label in the GUI, provide a default value for the user, and change how the option is validated.

Option values sometimes depend on the values of other options. Though not implemented, there were plans to allow the pattern designer to enable or disable an option based on the values of other options. For example, if you turned on caching in the N-Server[19] pattern, CO_2P_2S would enable the option that allowed you to specify the cache replacement policy. This would be implemented by allowing the pattern designer to fill in a Java hook method. The method would have the pattern's options passed in as an argument. The method would evaluate if the option should be enabled based on the values of the other options, and return true or false. Since the type system standardizes the mechanisms for accessing the data within the types, this could be done without intricate knowledge of the type internals.

In addition to the extensive modifications and additions to the type system, there were several other simplifications made to the MetaCO_2P_2S system during this research. They are listed here without going into further detail.

- Previously, files and directories needed for a pattern were distributed in many locations. All of the files and subdirectories for a pattern now reside within a single directory. This makes pattern management and distribution easier.
- You no longer need to specify the pattern image directory. The images for the pattern are now stored in the `images` directory relative to where the pattern is stored.
- You no longer need to specify where the extra classes for the pattern go. The pattern itself needs no Java classes now. The functionality has been moved to the type system and to the code generator.
- "Parameters" were renamed to "Options" and that portion of MetaCO_2P_2S was completely rewritten.
- The "Class Names" section has been removed. Class names are now just regular options.
- For the visual text elements a default value is no longer required for dynamic elements.

3.4.1 Requirements MetaCO_2P_2S places on the Type System

Examination of the use of types within MetaCO_2P_2S provides us with this list of requirements on the type system:

- Types need parameters to make them more flexible. Without parameters, almost all options would require the creation of their own unique type.
- The type system should provide support for validating and enabling or disabling options based on the content of other options within the pattern. To do this, standard methods must be provided to check the data within other options.
- A large variety of types are needed for the many facets of code generation. It should not be difficult to create new types. The ability to base new types on existing types would also be helpful.

3.5 Type Editor

In this section, we leave the domain of the Decorator pattern and move on to see how the types themselves are constructed. We will examine some examples of types that are used by the Decorator.

We are now looking at the Type Editor portion of Figure 3.1. The Type Editor (Figure 3.11) is the system through which types are created and modified. In the previous incarnation of MetaCO$_2$P$_2$S, there was no type editor; type creation was done by manually writing substantial amounts of Java. On the left side of Figure 3.11, we can see the list of types. The top seven in the list are the fundamental types. These types are used to compose all the other types.

String A simple string value. Enumerated types would normally be built around Strings. This type is represented by the `String` class in Java and the `str` type in Python.

Boolean Used to represent true or false choices. This type is represented by the `boolean` type in Java and the `bool` type in Python.

Integer A 64-bit integer type. This type is represented by the `long` type in Java and Python.

Float A 64-bit floating point type. This type is represented by the `double` type in Java and the `float` type in Python.

Record Records have a fixed number of fields. Each field has a name and a value. The value can be of any type except the `None` type.

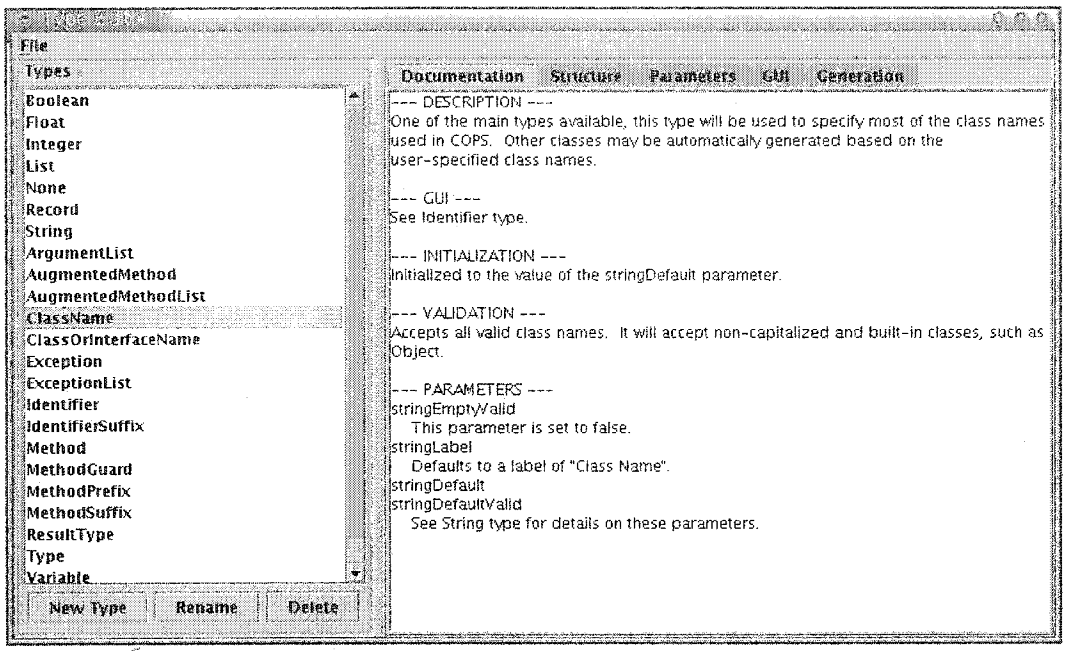
List Lists are composed of an arbitrary number of elements. Each element within a single list must be of the same type.

None The None type simply represents a type with no internal state. A type that contains only behaviour, or a type that inherits all of its state from another type, would be of type None.

Following the fundamental types in the type list are all the user-created types, organized alphabetically.

3.5.1 Type Components

On the top right side of Figure 3.11, the tabs show the five different components of each type. These are the type documentation, structure, parameters, GUI, and generation settings. Generation will be discussed in Chapter 5, but the other four components will be discussed here.



```
--- DESCRIPTION ---
One of the main types available, this type will be used to specify most of the class names
used in COPS.  Other classes may be automatically generated based on the
user-specified class names.

--- GUI ---
See Identifier type.

--- INITIALIZATION ---
Initialized to the value of the stringDefault parameter.

--- VALIDATION ---
Accepts all valid class names.  It will accept non-capitalized and built-in classes, such as
Object.

--- PARAMETERS ---
stringEmptyValid
   This parameter is set to false.
stringLabel
   Defaults to a label of "Class Name".
stringDefault
stringDefaultValid
   See String type for details on these parameters.
```

Figure 3.11: Type Editor showing ClassName Documentation

Documentation

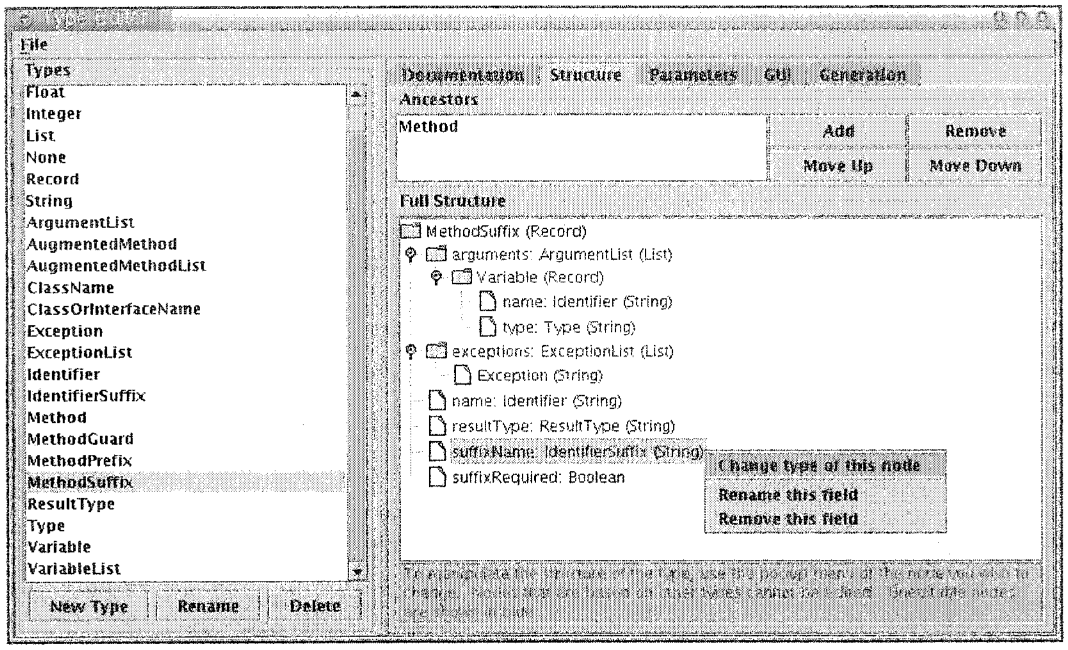
Previously there was only a handful of types available in the CO_2P_2S system. The documentation for each type was either implicitly provided by the GUI used to configure the type, or by reading a separate document. In the new system, the documentation for each type is written and viewed directly in the type editor. The documentation is formatted as simple text. The documentation describes the type, including the GUI, parameters, and validation that will occur. See Figure 3.11 for an example of what the documentation might look like.

Structure

Ultimately, the types are used for code generation. The values of the type instances are set by the CO_2P_2S application programmer. To use those values, the code generator needs to know how the information is structured. The Structure panel is used to define that structure.

Figure 3.12 shows the Structure for the MethodSuffix type. In the decorator pattern, a suffix method is a method that is called after a delegated method is called. The suffix has the option of modifying the return value from the delegated method. The MethodSuffix shares the method signature of the delegated method. Figure 3.12 shows that MethodSuffix inherits from the Method type. This is the mechanism by which the method signature is shared. The choice of inheritance mechanism will be discussed in the next chapter.

At the top level, the MethodSuffix is a Record. It has six fields, four of which are inher-



```
Float
Integer
List
None
Record
String
ArgumentList
AugmentedMethod
AugmentedMethodList
ClassName
ClassOrInterfaceName
Exception
ExceptionList
Identifier
IdentifierSuffix
Method
MethodGuard
MethodPrefix
MethodSuffix
ResultType
Type
Variable
VariableList
```

```
Method
```

```
MethodSuffix (Record)
    arguments: ArgumentList (List)
        Variable (Record)
            name: Identifier (String)
            type: Type (String)
    exceptions: ExceptionList (List)
        Exception (String)
    name: Identifier (String)
    resultType: ResultType (String)
    suffixName: IdentifierSuffix (String)
    suffixRequired: Boolean
```

Figure 3.12: Type Editor showing MethodSuffix Structure

ited. These fields are inherited from Method: arguments, exceptions, name, and resultType. In the type editor, the inherited fields and their descendants are shown in a different colour, indicating that the nodes are inherited and cannot be edited.

The arguments field has a type of ArgumentList. When the tree node is expanded, we can see that the ArgumentList type is composed of a List of Variables. Each Variable is a Record containing name and type fields.

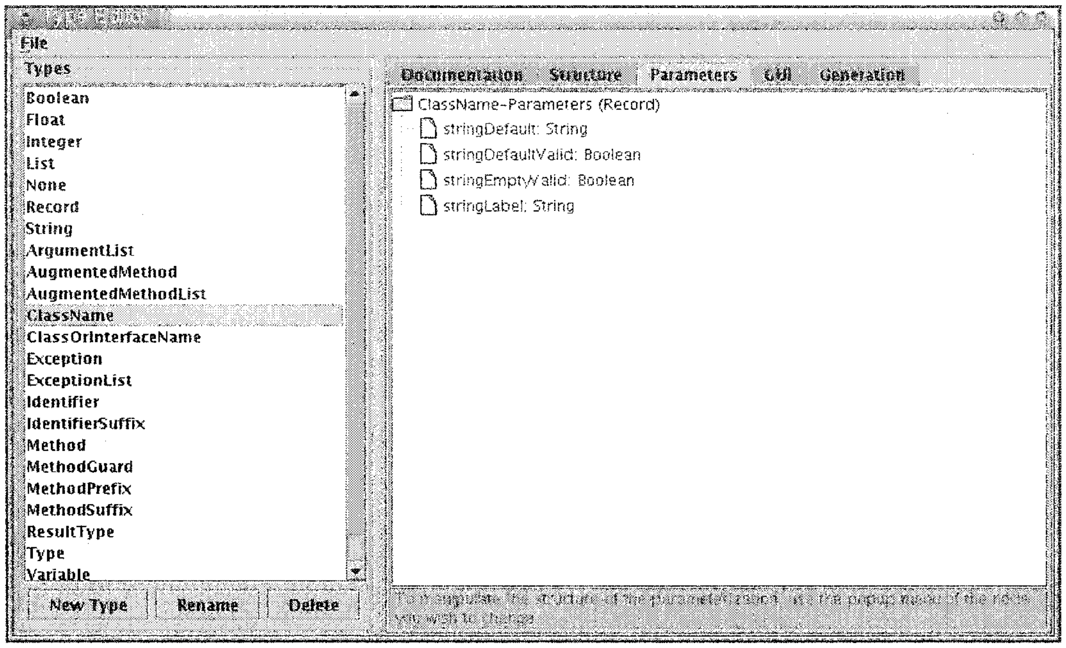
In addition to the fields inherited from Method, MethodSuffix defines the suffixName and suffixRequired fields. Just as with the inherited fields, the types of the fields are indicated to the right of the field names. If a type is not a fundamental type, the editor shows which fundamental type the type is associated with, in parentheses. This makes it easy to see and understand the full structure of a type without examining other types.

To modify the structure, the type designer right clicks on nodes within the tree. A menu similar to the one in Figure 3.12 is shown. If a node is a record, the popup menu will allow the user to add new fields.

Ancestors can be added or removed using the list and buttons shown near the top right of Figure 3.12. Ancestors can also be moved up or down in the ancestor list, the significance of which will be discussed in the next chapter.

Parameters

As discussed in Section 3.4, types require some sort of parameterization to make them more flexible. The parameterization of types comes in the form of a Record. Every user-created



```
ClassName-Parameters (Record)
    stringDefault: String
    stringDefaultValid: Boolean
    stringEmptyValid: Boolean
    stringLabel: String
```

To manipulate the structure of the parameterization, use the popup menu of the node you wish to change

Figure 3.13: Type Editor showing ClassName Parameters

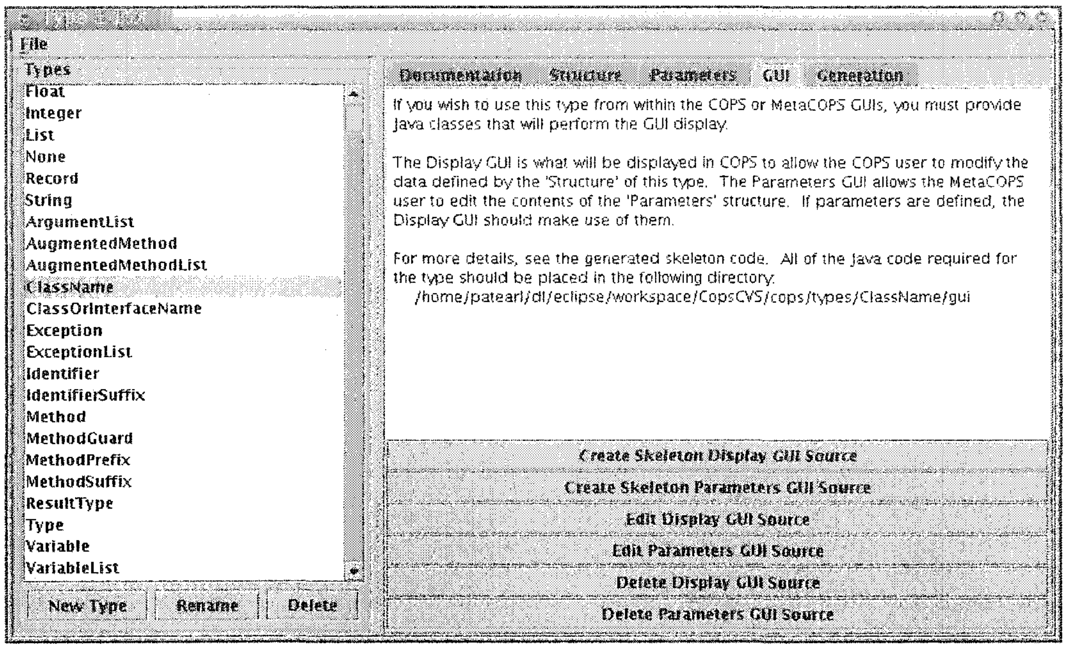
type, as well as the String, Boolean, Integer, and Float types, has a Record associated with it that defines the parameters for that type. Each parameter is simply added as a field within the parameters Record. Like the normal type structure, a parameter can be made up of records, lists, strings, booleans, integers, and floats. Unlike the normal type structure, no user-defined types may be used as part of each parameter. This means that any records and lists within paramaters will always be anonymous types.

When type B inherits from type A, the parameters from type A become parameters of type B. In Figure 3.13, we can see the four inherited parameters that make up ClassName's parameterization. ClassName inherits from Identifier, which in turn inherits from String. By convention, parameters have names that start with the full name of the type in which they are defined. This convention helps avoid conflicts and indicates where the parameters were originally defined.

Note that the structure of the parameterization is manipulated in the same way that the type's main structure is manipulated. As will be described in the next chapter, the underlying methods for manipulating the data defined by the structure are also the same.

GUI

CO_2P_2S and MetaCO_2P_2S use GUIs to display the types. Since CO_2P_2S is written in Java, the GUIs for the types are also written in Java. In addition to definining the GUIs, we also need mechanisms to initialize the type instances, validate their contents, and convert the contents to a string.



If you wish to use this type from within the COPS or MetaCOPS GUIs, you must provide Java classes that will perform the GUI display.

The Display GUI is what will be displayed in COPS to allow the COPS user to modify the data defined by the 'Structure' of this type. The Parameters GUI allows the MetaCOPS user to edit the contents of the 'Parameters' structure. If parameters are defined, the Display GUI should make use of them.

For more details, see the generated skeleton code. All of the Java code required for the type should be placed in the following directory:

```
    /home/patearl/dl/eclipse/workspace/CopsCVS/cops/types/ClassName/gui
```

Figure 3.14: Type Editor showing ClassName GUI Panel

By default, types are initialized to zero, empty string, false, or empty list values. At first glance, it may seem that it would be convenient to initialize values via a GUI mechanism. Digging deeper, we can see that initialization of values based on the contents of parameters, and the initialization of lists, would add considerable complexity to such a GUI. The approach chosen for intialization was to allow the setting of values through a standard data access API. This standard API will be described completely in the next chapter.

Validation can also be very complicated, involving anything from checking the ordering of list elements to the capitalization of the first character of a class name. Due to the complexity and variety of validation tasks, the validation is performed with Java code.

For similar reasons, the conversion to a string is also handled by Java code. The GUI code is created using a simple yet flexible framework which will be described in full detail in the next chapter. Examples of GUI code will be provided later in this chapter.

In the GUI panel (Figure 3.14), the six buttons on the lower right provide three operations that may be performed for either the "Display GUI" or the "Parameters GUI". These buttons provide convenient access to the creation, manipulation, and deletion of the two GUI source files.

The "Display GUI" is what will display the type to the CO_2P_2S application programmer. It also contains the code that will perform the initialization, validation, and /tt toString conversion needed by CO_2P_2S. This is the code responsible for the work shown in Section 3.3.

The "Parameters GUI" is what will display the type's parameters to the pattern designer in MetaCO_2P_2S. Initialization and validation of the parameters is also performed by the code in the "Parameters GUI". This is the code responsible for the work shown in Section 3.4.

There are three operations that can be performed on each of the Java GUI files. A skeleton template can be generated that provides framework and documentation for the type designer to work from. The Java code can be edited by clicking on the edit button. This will open the user's favorite editor as defined in the CO_2P_2S preferences. If no longer needed, the Java file may be deleted by clicking the delete button. See Appendix C for the skeleton code generated by the first two buttons.

Since the types involve the creation of Java code, the compilation of the Java code is also required. This compilation occurs when you save the types via the file menu. Any errors in the compilation can be detected and rectified immediately to prevent later problems.

3.5.2 Creating a New Type

To create a new type, the user clicks on the "New Type" button shown in the bottom left of Figure 3.14. After entering a name for the type, the user needs to choose an existing type to base the new type on.

If None is selected, the type will have no state of its own. It may add ancestors which provide state, but when the ancestors are removed, the type will have no state. Types with no state can be used to add packages of behaviour for use in code generation. When a type has no state, it may be inherited by any type, and it may inherit from any type.

If Record is selected, an empty Record will be created. The user may add local fields or other Record-based ancestors. Unlike the local fields, inherited fields cannot be modified or removed.

If List is selected, another dialog box will ask the user to choose the type for the elements in the list. Lists of lists can be created.

If any of the other types are selected, the new type has a type of None and the selected type is added as an ancestor. The added ancestor defines the state of the created type. For example, if String is selected, the structure of the type will be a simple string value. If the String ancestor is removed, the type will revert to None. When a type is based on a Record of some sort, it will allow the addition of local fields. If no local fields are added and the Record-based ancestor is removed, the state of the type will revert to None. If however, local fields are added while the type has a Record-based ancestor, the type will remain a Record when its ancestors have been removed.

If required, a type can be renamed or deleted at a later point. When renamed or deleted, any data that previously used those types needs to be recreated within MetaCO_2P_2S and CO_2P_2S. In addition, the GUI source files may need to be modified. Renaming and deletion of types requires changing or removing references to the type. In practice renaming and deletion is uncommon with mature types.

3.5.3 How the Type System Meets the Requirements

This section shows how the requirements of CO_2P_2S (Section 3.3.1) and MetaCO_2P_2S (Section 3.4.1) are met by the type system. It will do so in the context of types that are used directly or indirectly by the Decorator pattern. The requirements are shown in italics and the solutions in plain text.

Structure of Data

The structure of the data in the options must be defined, either implicitly or explicitly.

In the previous system, the structure of the data was defined implicitly by the Java source that was written for each option type. In the new system, the options are specified explicitly by combining types into a tree as shown in Section 3.5.1.

Since the structure is defined in a standard manner, the data within the system can also be manipulated in a standard manner. The invention of file formats and internal data storage techniques is not necessary. The system handles all of that automatically using the pre-defined structure.

In addition to simplifying the code within CO_2P_2S and MetaCO_2P_2S, the pre-defined structure specifies the interface to the data that will be used by the code generator. Record fields are always accessed as fields within a class and lists are always accessed by standard list operators. Accessing the data within a type is a well-defined process at any level of the system.

For example, because the structure of data is well-defined, it was possible to add a "Cancel" button to the dialogs in CO_2P_2S. The `Data` models know how to backup and restore their own data. This functionality is used by the automatically created dialog boxes to provide the "Cancel" button.

Initialization and Validation

The values manipulated by the GUI need to be initialized and validated.

As described earlier, initialization and validation can be complex tasks. Even though they may be complex, they can usually be expressed succinctly in the form of Java code. To initialize or validate types, the type designer must provide initialization and validation methods. Figure 3.15 shows the methods for initialization and validation of the String type.

The `model` that is passed to `intializeModel()` and `validateModel()` represents the structure of the String type. Since the structure of the String type is just a simple string, we use it as such. Since the String type has parameters, we can get the Record that contains those parameters by calling `getParameters()` on the String `model`.

```
public void initializeModel(Data model) {
    DataRecord params = model.getParameters();
    model.asString().set( params.getString("stringDefault").get() );
}

public String validateModel(Data model) {
    DataRecord params = model.getParameters();

    String label = params.getString("stringLabel").get();
    String current = model.asString().get();

    boolean emptyOk = params.getBoolean("stringEmptyValid").get();
    if(!emptyOk && current.equals("")) {
        if(label.equals("")) {
            return "Please enter a non-empty string.";
        } else {
            return "Please enter a non-empty string for the field" +
                   " labeled:\n" + label;
        }
    }

    boolean defOk = params.getBoolean("stringDefaultValid").get();
    String def = params.getString("stringDefault").get();
    if(!defOk && current.equals(def)) {
        if(label.equals("")) {
            return "The default show here is invalid:\n" + def;
        } else {
            return "The default show here is invalid:\n" + def +
                   "\nIt cannot be used in the field labeled:\n" + label;
        }
    }

    return null;
}
```

Figure 3.15: String Initialization and Validation

In initializeModel(), the first thing we do is get the DataRecord that contains the key/value pairs representing the String's parameters. Second, the "stringDefault" parameter is retrieved. Since the elements in the parameters Record could be of any type, we use getString(key) to cast the value to its proper DataString type. The DataString is a container for a plain Java String. It supports the get() method which returns a String, and the set(value) method which takes a String as an argument. Note that the model is passed into the method as a Data instance. Since we need to use it as a string, we call the convenience method asString() on it to cast it to a DataString. Putting that all together, initializeModel() initializes the value of the string to the contents of the "stringDefault" parameter.

Looking at validateModel(), we can see that similar operations are occuring. Again, the parameters are retrieved for the String type represented by the model. The "stringLabel" parameter and the model's current String representation are stored in convenience variables. Next, we fetch the "stringEmptyValid" parameter from the parameters Record. Since this time the parameter is not a string, but a boolean value, we use getBoolean() to perform the cast to a DataBoolean for us. As with DataString, we use the get() method to access the boolean inside. After performing some validation logic, we return a string indicating what has gone wrong. A similar process is repeated for the default value parameters. If nothing is wrong, we simply return null.

The type system will examine the return value of the validateModel() type. If it is non-null, it will display the provided error message and will ensure that the user makes the appropriate changes before proceeding.

From these examples, we can see that the initialization and validation of types is performed by retrieving and storing values using the methods that are part of the Data family of classes. Aside from learning how to access the data model, the type designer uses simple Java tools to perform the tasks of initialization and validation.

Though the code shown for initialization and validation is for CO_2P_2S, the MetaCO_2P_2S method is very similar. The only differences are that the type designer would only be accessing the parameters, and the parameters would be accessed in a slightly different manner.

GUIs and Widgets

Each pattern option has a GUI associated with it that is used within CO_2P_2S. These GUIs may contain a variety of widgets.

To simplify the creation of GUIs, the type system assumes that the type designer will be laying out a panel by specifying "rows" of widgets that flow from top to bottom inside a container panel. If the type designer wishes to lay out the widgets in a more complicated manner, it can be done using the standard mechanisms provided by the Java Swing Library.

```
public void initializePanel(Data model) {
    // Create a new panel into which we will stuff a label and a text field
    Box panel = Box.createHorizontalBox();

    // Put the parameter stuff into convenience variables.
    DataRecord parameters = model.getParameters();
    String label = parameters.getString("stringLabel").get();

    // Add the label.
    panel.add(new JLabel(label));

    // Add a bit of space.
    panel.add(Box.createHorizontalStrut(5));

    // Add the field.
    panel.add(new JTextField(model.asString(), null, 0));

    // Add our dual-widget panel to the main panel.
    add(panel);
}
```

Figure 3.16: String GUI Source

Instead of providing a few fixed widgets, the type system provides models upon which widgets act. In Figure 3.16, the `JTextField` is created with `model.asString()` as its data model. The `DataString` class follows the Java `Document` interface. The other members of the Data family also provide similar models that can be easily attached to widgets. The models will be discussed further in the next chapter.

If a model cannot be easily attached to a widget, the widget can just be created manually based on the primitive values stored in the data model. The type system provides a `dumpToModel()` method that will be called when the information from the GUI is required by the model. The type designer need not worry about using events to update the model when the GUI contents change. Like the `validateModel()` method, `dumpToModel()` may return a String indicating that an error has occured.

The GUIs for CO_2P_2S and MetaCO_2P_2S are created using the same techniques. There is no longer any need to learn many different methods for creating GUIs.

As seen in Figure 3.16, the creation of a GUI involves combining standard Swing widgets and `Data` models into a panel. The final `add()` method adds the constructed label/field box to the main panel. If the panel needs to be displayed as a dialog, the type system will handle that automatically.

```
public void initializePanel(Data model) {
    add(getSuperPanel("Method"));
    addSpacer();
    add(getSuperPanel("MethodGuard"));
    addSpacer();
    add(getSuperPanel("MethodPrefix"));
    addSpacer();
    add(getSuperPanel("MethodSuffix"));
}
```

Figure 3.17: AugmentedMethod GUI Source

Composition of GUI Components

Support the composition of larger GUIs from smaller GUI components. There is also a need for the ability to spawn child dialog boxes to manipulate smaller pieces of each option.

While the old system provided some support for creating GUIs from smaller components, the combination of GUI components was limited to the small set of built-in types. The new system provides a unified approach for combining all types of GUI components. There are three mechanisms for combining existing types into larger GUIs. The GUI panel for a given ancestor can be retrieved. A GUI panel for fields or list elements can be retrieved. As well, dialogs can be automatically created for fields or list elements. An example of the construction of the AugmentedMethod GUI is shown in Figure 3.17. This example demonstrates the simplicity of combining GUIs from ancestor types into a larger GUI.

GUIs for fields and list elements are created in a similar manner. Suppose that the passed in `model` represents a record with a String field named "foo". We could create a GUI for the String by performing one of these operations:

- `model.getData("foo").getPanel()` This method returns a panel which might then be `add()`ed to the parent panel.
- `model.getData("foo").getDialog()` This method spawns a dialog that can be used to modify the contents of the "foo" field.

Using these simple techniques, complicated GUIs can be created using the many available smaller components.

Lists of Arbitrary Items

Support lists of arbitrary items without having to create a complicated list GUI each time.

Though the previous type system had a rudimentary parameterized list type, it rquired the creation of substantial amounts of Java code to function. The new system requires that

```
public void initializePanel(Data model) {
   DataRecord params = model.getParameters();
   String title = params.getString("augmentedMethodListTitle").get();

   DataJList jlist = new DataJList(model.asList(), title);

   add(jlist);
}
```

Figure 3.18: AugmentedMethodList GUI Source

the user specifies the exact contents of their lists, but still allows for the easy creation of lists.

We just saw an example of how the AugmentedMethod GUI was created. Figure 3.18 shows how a list of these would be created. Note that the underlying data structure is specified as a List of AugmentedMethod elements. The `DataJList` and `DataJTable` types were created to make it easy to display and edit lists. The `DataJList` and `DataJTable` types will be described in detail in the next chapter. In this case, we can see that creating a fully functional list is as easy as passing the model and an optional title to the `DataJList` constructor. The items in the list are displayed according to their visual string representation, described next.

String Representation of Arbitrary Data

In the graphical representation of the pattern, options need to be converted to a string form to be displayed. In the case the option is already in a form similar to a string this conversion is trivial. For complicated structures like lists and records, more capabilities are needed.

In the old system, this conversion to strings was still performed, but the new system does so in a more consistent and simpler manner. Figure 3.19 demonstrates how an entire method signature string can be constructed from its component types. It is basically a matter of performing a few Java operations to combine the string representations of the various fields and list elements.

Like the GUI panels, the string representations of ancestors can be accessed using the `getSuperString("ancestorType")` method.

Parameters

Types need parameters to make them more flexible. Without parameters, almost all options would require the creation of their own unique type.

We have seen how parameters can be used to make types more flexible. The structure of the parameters is defined within the type editor, making it apparent which parameters

```
public String toString(Data model) {
    String res = "";
    DataRecord rec = model.asRecord();
    res += rec.getData("resultType").toString();
    res += " ";
    res += rec.getData("name").toString();
    res += "( ";
    Iterator it = rec.getList("arguments").iterator();
    while(it.hasNext()) {
        DataRecord arg = (DataRecord)it.next();
        res += arg.getData("type").toString();
        res += " ";
        res += arg.getData("name").toString();
        if(it.hasNext()) res += ", ";
    }
    res += " )";
    return res;
}
```

Figure 3.19: Method toString Source

each type has. As with the normal data structure, what was implicit in the old system has been made explicit in the new system.

Inter-Option Checking

The type system should provide support for validating and enabling or disabling options based on the content of other options within the pattern. To do this, standard methods must be provided to check the data within other options.

Though inter-option validation and enabling was not implemented in time for the thesis, well-defined mechanisms for accessing the option data are provided. The standardization of data access makes the implementation of inter-option checking straightforward.

Easy Type Creation

A large variety of types are needed for the many facets of code generation. It should not be difficult to create new types. The ability to base new types on existing types would also be helpful.

All through this chapter there have been examples of how types can be created using other types as a starting point. Most individual types require only a basic structure and a handful of lines of Java code. This makes it easy to understand existing types and create new types.

Once these types are created, they are seen by the pattern designer as tidy bundles that can simply be used from the familiar GUI environment of CO_2P_2S and MetaCO_2P_2S. The pattern designer is no longer required to write substantial amounts of Java to display or perform code generation for complex pattern options.

When creating types, the type editor performs checking to ensure that the types never fall into an inconsistent state. Whenever a change is attempted in the type editor, it will be checked for validity before it is allowed to occur. The early detection of mistakes assists the type designer in the type creation process.

Modularity

Well-defined interfaces that allow for modularity within the CO_2P_2S system are desirable. Though not mentioned before, modularity was one of the goals of the type system. The ability to separate pieces of the system makes it easier to develop extensions that replace or modify existing modules in the $\mathrm{CO_2P_2S}$ system.

Since the structure and parameterization are well-defined, it is much easier to develop separate tools that works with the data. Before, the structure was defined implicitly by writing Java code. Without examining the Java code, one could never be sure what format the data would be in. In fact, the code generator could not be separated from the definition of the types, because the data format was proprietary to each type.

3.6 Summary

This chapter has examined the Decorator pattern and a number of examples of types. It has described how the type system shown in Figure 3.1 fits together. The precise APIs and underlying design decisions behind the type system will be examined in the next chapter.

Chapter 4

Type System Details

The previous chapter discussed a few examples that demonstrated the use of the type system. This chapter will examine some of the design decisions in the type system. It will also provide details of the API used by the type designer in creating new types.

4.1 Type Combination

Sometimes the type designer wishes to combine the functionality of two or more individual types into an aggregate type. If the types being combined do not depend on data contained within the other, a simple record can be used to combine them. In the record, there would be one field for each of the required types. If a type depends on data contained within another type, a more complex technique for combining types is required. As an example, many patterns use methods, naturally leading to a type that represents a method signature. Sometimes these methods need to be prefixed and suffixed by additional methods that cause some change in behaviour. The techniques for performing type combination will be demonstrated by combining the three types shown in Figure 4.1. MethodPrefix and MethodSuffix both require the information provided in a MethodSignature. Three ways to create aggregate types were identified: References, Delegation, and Inheritance.

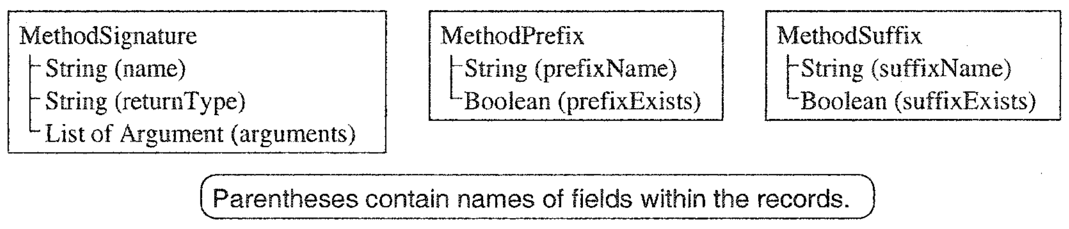


Figure 4.1: Example Types to be Combined

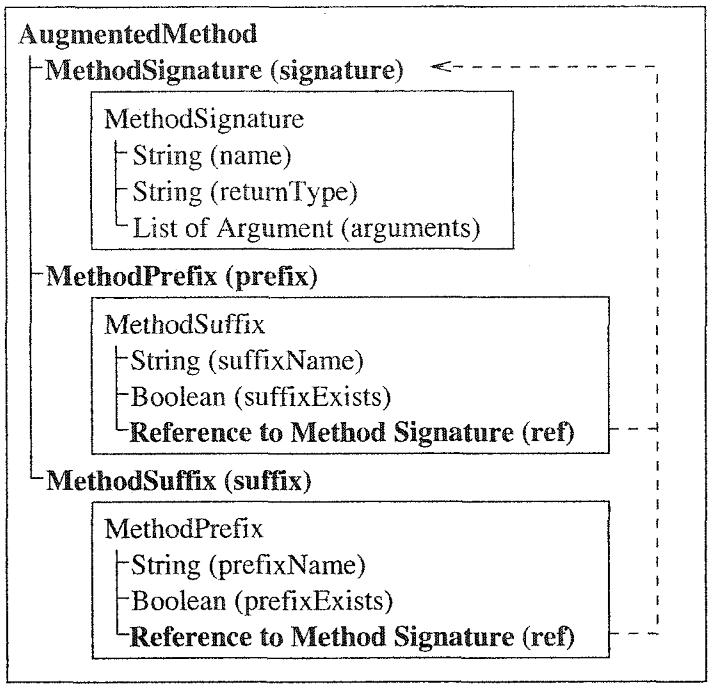


Figure 4.2: Combining Types Using References

4.1.1 References

One way to combine types is to allow types to have references to other types. Figure 4.2 demonstrates this technique. In the figure you can see that a "Reference to Method Signature" has been added to the MethodPrefix type and the MethodSuffix type. The MethodPrefix and MethodSuffix types can follow the references to find the data they need from the MethodSignature. The type designer would access the signature, prefix, and suffix as fields of the new AugmentedMethod. The references, as shown by the arrows, are specified when AugmentedMethod is created. The type designer would need to specify where the references pointed. In this case, the `self.signature` identifier would be used to resolve the two references. The main disadvantage of this technique is the added complexity of dealing with the references. Instead of somehow resolving themselves, the type designer must create explicit links for each reference. As an advantage, name conflicts are avoided due to the separation of the types into individual fields.

4.1.2 Delegation

In the case of the method signature example, there are three pieces of code that will ultimately be produced by the code generator. One of these is the delegated call to the method that matches the method signature. Another piece is the call to the prefix method that may modify the arguments which will be passed to the delegated method call. The call to

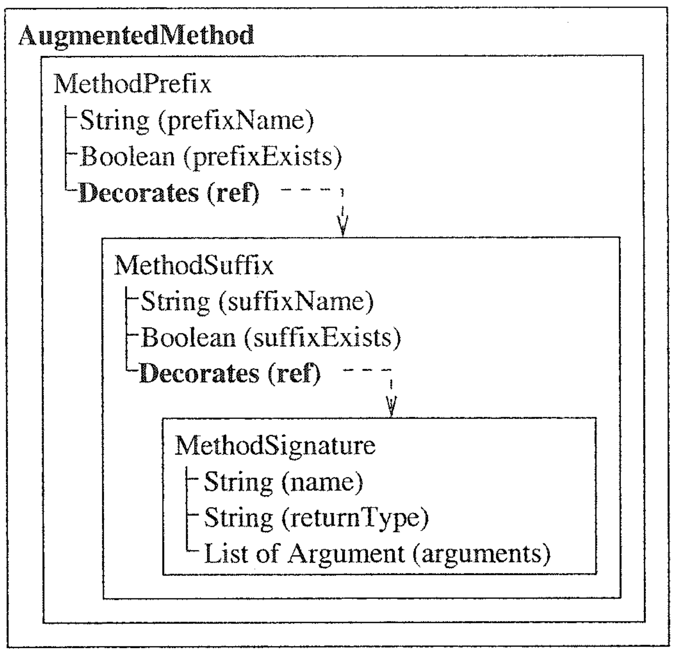


Figure 4.3: Combining Types Using Delegation

the suffix provides an opportunity to do post-processing and potentially modify the return value provided by the delegated method. In this example, it is possible to think of the MethodSuffix as decorating the MethodSignature by supplying additional code after the delegated call. The MethodPrefix further decorates by providing additional code before the delegated call. Figure 4.3 shows how the AugmentedMethod would be constructed. The outer MethodPrefix decorates a MethodSuffix, which in turn decorates a MethodSignature. When MethodPrefix tries to access the `ref.name` field, MethodSuffix will determine that it does not own that field and will pass the field access on to MethodSignature.

Note that the technique described here is not decoration in the sense of the design pattern. There is no common interface specified for the different types. The types just provide mechanisms for accessing the wrapped types, more like a system of delegation.

Delegation and inheritance are able to solve similar problems [27, 28]. One of the advantages of delegation is that the ordering of member resolution can be specified at runtime. However, the runtime specification of the ordering would shift some of the burden from the type designer to the pattern designer. To avoid this, the order of delegation would need to be specified at type creation time. This is no easier than specifying the resolution order of classes in some multiple inheritance systems. Using delegation to share data leads to additional issues also found in multiple inheritance. If two types share the same field name, there would be no way to access the inner field without providing additional access mechanisms. Using delegation under these conditions is no less difficult than using simple

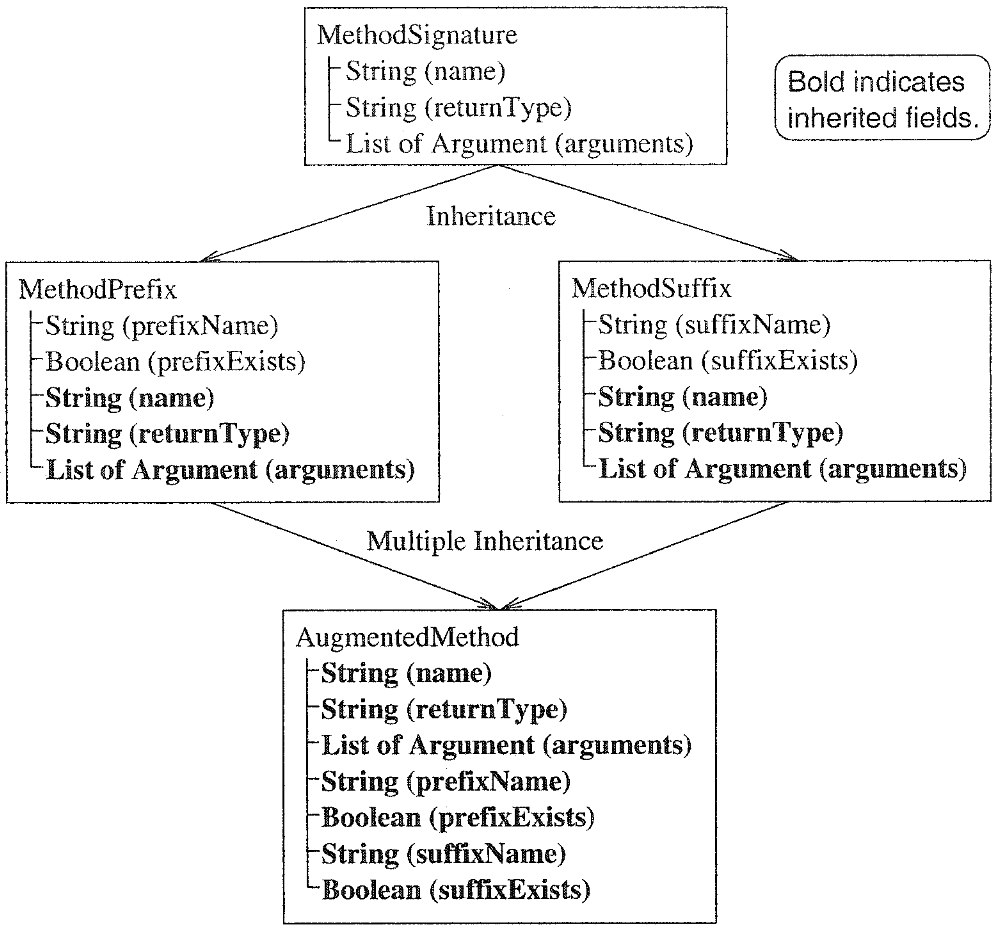


Figure 4.4: Combining Types Using Inheritance

multiple inhertance.

In the investigation of delegation and inheritance, the option of using a prototype-based system [11, 21] was also considered. Prototypes would be useful in a code generation language in that you could simply clone an existing type, make modifications, and then use those modifications to produce additional code. However, using prototypes in that situation is more complicated than it needs to be. Since the modified prototype is not likely to be reused, the programmer might as well write the code outside of the object. As well, the prototype model does not extend well to the idea of having a self-contained type, which may be accessed repeatedly by pattern designers. Having a set of classes would be more intuitive to the pattern designer, who would be familiar with Java.

4.1.3 Inheritance

Instead of using references, inheritance shares data by combining inherited and local fields directly within the same type. For an example, see Figure 4.4. Inheritance was used in the type system for the following reasons:

- Inheritance removes the need for the type designer to explicitly specify references to other types.
- Inheritance would be familiar to the type designer, who would have used Java.
- When inheritance occurs, the resulting data structure is no more complicated than the simple combination of the ancestor types. Understanding the structure of the data is simple.
- As will be explained in the next chapter, the Python[6] language is used as part of the code generation system. Using inheritance makes the types fit naturally with Python.

In an inheritance based system, there are a number of issues that must be considered.

Single Inheritance versus Multiple Inheritance

Single inheritance is often easier to use and implement when compared with multiple inheritance. However, single inheritance leads to some issues that are not easily resolved without a more flexible mechanism. When seeking to combined multiple types, single inheritance dictates that a chain of types must be created. If the types are loosely related, this chain of types may create intermediate types that make little sense on their own. With multiple inheritance, it is easier to add a new feature to an existing type anywhere in the type hierarchy. Multiple inheritance was chosen because the additional simplicity of representation was more important than the negative aspects of the issues discussed mext.

Data Copying and Name Conflicts

Knudsen [20] demonstrates the different relationships possible in a multiple inheritance system. The basic choice Knudsen describes is the ability to specify whether the base class is copied. Figure 4.5 shows the essence of this choice. Because of the variety of circumstances under which general purpose inheritance might be used, Knudsen advocates flexibility in allowing the programmer to choose how ancestor types are combined. To maintain the simplicity of the type system, the base class, A, is not copied. D will only be able to see one copy of the members from A. Because one of the primary purposes of inheritance was to allow for data sharing, always copying the base class would not work. Providing the choice would have made the system more complicated than it needed to be. If the type designer needed another copy of the base class, the missing behaviour could be emulated by creating fields for B and C in D.

With any form of inheritance there is the potential for name conflicts. With single inheritance, we can resolve these conflicts in a straightforward manner by using the member closest to us in the inheritance hierarchy.

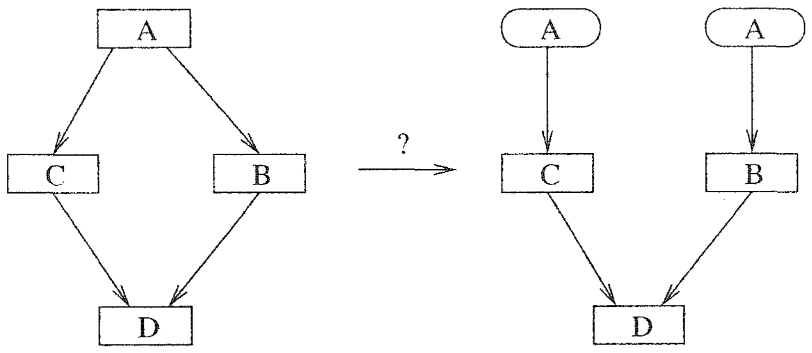


Figure 4.5: Multiple Inheritance Example

With multiple inheritance, dealing with conflicts is more complicated. Some languages provide mechanisms to rename conflicting members to avoid conflicts. To promote the simplicity of the type system, name conflicts are not allowed. Because of this, no mechanism is needed to explicitly resolve name conflicts in the structure of the data. Note that parameters use a naming convention that places the name of the type before the rest of the parameter name. Though conflicts may be tricky to resolve when they do occur, they seem to occur rarely in practice.

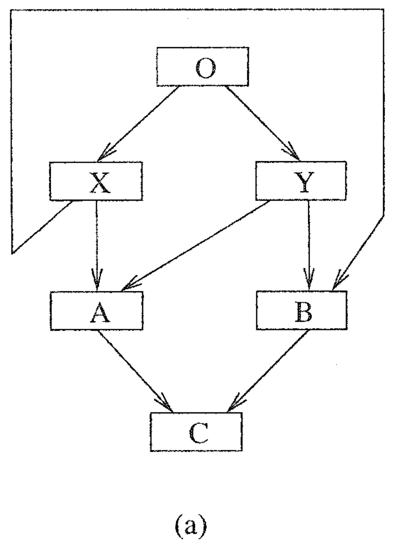
With multiple inheritance, it is not always clear where a particular method comes from. Since the type system does not clone the base class, it is possible to create a linearization of the type hierarchy which may be traversed when searching for a particular method. This is the technique used by Python as described in the next section.

Method Resolution Order

Due to the use of Python as the underlying language for the code generator, the type system adopted the use of the Python method resolution ordering[8]. When determining which method to use, the type system must make sense of the type hierarchy. If the hierarchy is a simple tree, we need only search up the tree until we find a matching method. However, with multiple inheritance the situation becomes more complex.

Each type can have multiple ancestors. These ancestors ultimately need to be traversed in some order when seaching for methods. The type editor allows the type designer to change the ordering using the "Move Up" and "Move Down" buttons shown in Figure 3.12. The higher the ancestor is in the list, the sooner its methods will be resolved. In other words, the higher a type is in the list, the more specific it is considered to be.

Figure 4.6 (taken from [8]) shows how the method resolution order can be ambiguous. In Figure 4.6, the types inherited from the left take precedent over the types inherited from the right. We can see from Figure 4.6a that class A implies an ordering of (X, Y), meaning that a method of the same name in X and Y will come from X. Class B implies an ordering



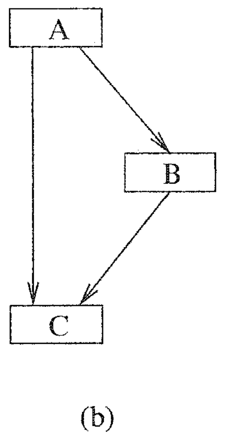


Figure 4.6: Method Resolution Order

of (Y, X). When class C combines A and B, the order of resolution of X and Y becomes ambiguous. In Figure 4.6b, class C implies an ordering of (A, B), but class B itself implies an ordering of (B, A). That situation is also ambiguous. The type editor will not allow the type designer to create ambiguous type hierarchies. It performs checks to ensure that valid resolution orders can be created for all of the types on the system. The method resolution order is constructed according to the algorithms from [8].

Forms of State

In conventional class-based systems, everything is like a record with fields. When inherting from other classes, we would simply check for conflicts between the fields. In the CO_2P_2S type system, types can take the form of numbers, strings, or lists. In these cases, inheritance must be handled slightly differently. For example, it does not make sense to combine a string and a record directly. To handle this, when using a non-record type there is only one type within an entire inheritance hierarchy that may provide its own state. This base type would be either String, Boolean, Integer, Float, or a user-defined List type.

For simplicity, inheritance of fields is not allowed in anonymous records within the type. Fields can only be inherited in the top level records of user-defined types. As a clarifying example, suppose we have a type A and a type B. We need to create a list of A plus B. Allowing the inheritance of fields inside anonymous records is akin to combining a list of A and a list of B. That behaviour is not allowed, forcing the user to create a list of C, where C is a combination of A and B. The later model makes more sense since each C element would know how to handle itself.

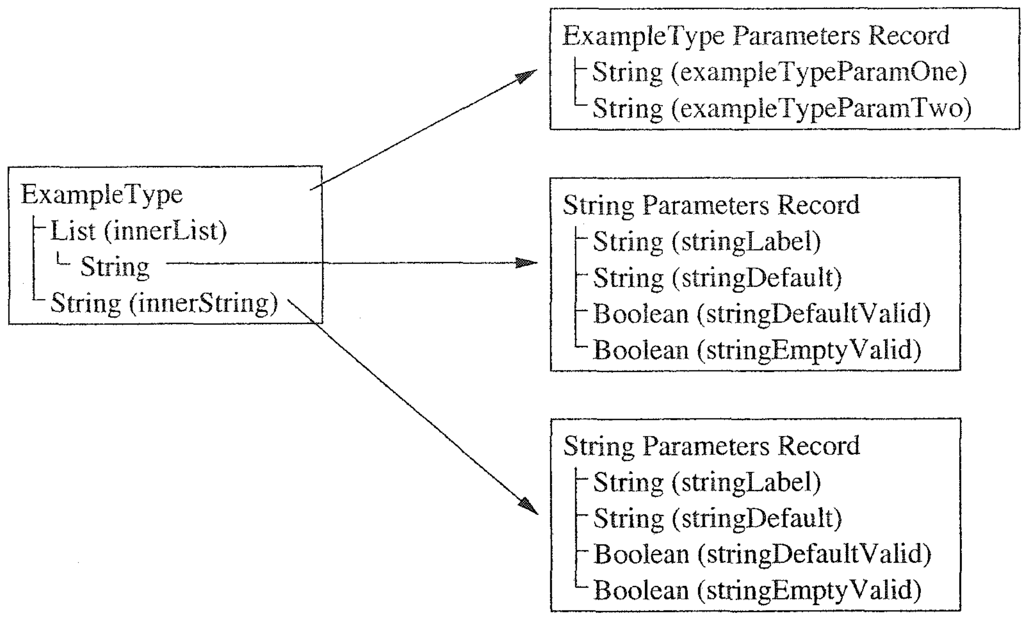

ExampleType
List (innerList)
String
String (innerString)
ExampleType Parameters Record
String (exampleTypeParamOne)
String (exampleTypeParamTwo)
String Parameters Record
String (stringLabel)
String (stringDefault)
Boolean (stringDefaultValid)
Boolean (stringEmptyValid)
String Parameters Record
String (stringLabel)
String (stringDefault)
Boolean (stringDefaultValid)
Boolean (stringEmptyValid)

Figure 4.7: Accessing Parameters

4.2 Accessing Parameters

As explained earlier, each type is associated with a record containing zero or more parameters. In Figure 4.7 there is an ExampleType Record with two fields. One of the fields, `innerList`, is an anonymous List of Strings. The other field, `innerString`, is a String. On the right side of the figure are the parameters that are associated with each component of the type. Though there is only a single type shown on the left, the composition of smaller types has created a situation where many parameters exist. It may be necessary for the higher level type to modify the GUI label for one of the inner strings. To do this, there must be a way to access the parameters of contained types. In the Display GUI which is used in CO_2P_2S, a `Data` model is passed in. In the Parameters GUI, a `DataTree` tree is passed in. To access the parameters in Figure 4.7 via the model and tree, we would do the following:

```
DataRecord params;

// Display GUI
params = model.getParameters();
params = model.getData("innerList").getData(0).getParameters();
params = model.getData("innerString").getParameters();

// Parameters GUI
params = tree.getParameters();
```

```
    params = tree.getField("innerList").getElement().getParameters();
    params = tree.getField("innerString").getParameters();
```

When accessing the normal model's list, we get the individual list element (0 in this example) and use its parameters. Normally one would not need to access parameters as part of a list, so this is just shown for completeness. The normal case would be that the String GUI would simply use the parameters that were set, even if they were specialized for ExampleType.

When accessing the tree from the code that runs in MetaCO_2P_2S, the calls are slightly different. In particular, when accessing the parameters for the list element, we would call `getElement()` to modify the parameters used for all instances contained within the list.

Conceptually, the parameters are designed to be set in MetaCO_2P_2S (by the parameters GUI) and then read in CO_2P_2S (by the display GUI).

4.3 API Overview

This section provides an overview of the parts of the API that the type designer might need to use. The rest of the API is documented in Appendix E. The classes are divided into three categories. Data access classes are used to read and write the data contained within the type instances. Framework classes make up the inherited framework that the type designer works within. Finally, the GUI classes provide some useful widgets for the type designer to work with.

4.3.1 Data Access Classes

DataTree

The `DataTree` class is used within the Parameters GUI to access the parameters that are part of the type. This was explained in Section 4.2.

Data

This is the superclass for the remainder of the data access classes. It provides common functionality. See Appendix E for the full list of methods.

```
class Data {
    // Access Methods
    public DataRecord getParameters();
    public JPanel getPanel();
    public boolean createDialog();
    public String toString();
    // Casting Methods
    public DataString asString();
    public DataBoolean asBoolean();
    public DataBoolean asInteger();
    public DataInteger asFloat();
    public DataList asList();
    public DataRecord asRecord();
}
```

getParameters() Returns the parameters associated with this data item. If none are available, `null` is returned.

getPanel() Returns the `JPanel` for this data item. This method is used to access the GUIs of child elements. If no panel is available, `null` will be returned.

createDialog() Creates a dialog whose contents are the same as the panel returned by `getPanel()`. The dialog will have "OK" and "Cancel" buttons. If "Cancel" is selected, any changes to the model will be rolled back. Cancelling a higher level dialog will also cancel the changes made by dialogs created from inside the higher level dialog. `createDialog()` returns `true` if the dialog is "OKed", `false` otherwise.

toString() Returns the string representation of this data item.

Casting Methods These methods simply perform a cast to one of the indicated Data types.

DataString

`DataString` is a container for a Java `String`. This class implements the `Document` interface, providing a convenient model for use in text widgets.

DataBoolean

`DataBoolean` acts as a wrapper for a `boolean` Java primitive. It also provides the `ButtonModel` interface for easy manipulation using various button-like widgets. `JCheckBox` is able to take advantage of the ButtonModel interface.

DataInteger

This class acts as a wrapper for a `long` Java primitive.

DataFloat

This class acts as a wrapper for a `double` Java primitive.

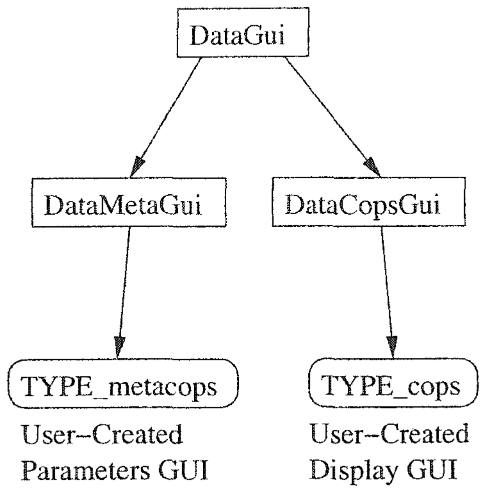


Figure 4.8: Type Framework

DataList

This class acts as a full featured list data type. It implements a number of Java interfaces for easy access to the underlying type. Operations such as sorting can be accomplished through the `List` interface. GUIs can be created based on `ListModel` or `ComboBoxModel`. Note that the DataList performs checks to ensure that only elements of the right type can be added. Elements of the right type can be created using the `create` factory methods. Creating an element does not add it to the list; this must be performed as usual through the `List` interface methods.

DataRecord

This class provides the `Map` interface for manipulating records. Note that the optional `clear()`, `put()`, `putAll()`, and `remove()` methods are not supported.

4.3.2 Framework Classes

The types in this section are used in the creation of the display and parameters GUIs. Remember that the GUI source includes not only GUI creation, but also initialization and validation. The class hierarchy is shown in Figure 4.8 with the user created types at the bottom.

DataGui

At the highest level, the `DataGui` class provides functionality that is common to both the parameter and display GUIs. This class is itself a `JPanel`. In the constructor, the JPanel is set to use a vertical box layout.

DataCopsGui

This class is used as the superclass for all of the "Display GUI" classes. It provides default implementations of the five methods that need to be overridden.

```
class DataCopsGui extends DataGui {
    public void initializeModel(Data model);
    public void initializePanel(Data model);
    public String dumpToModel(Data model);
    public String validateModel(Data model);
    public String toString(Data model);

    protected JPanel getSuperPanel(String ancestor);
    protected String getSuperString(String ancestor);
}
```

initializeModel() This method is responsible for initializing the contents of the given model. If it is not overridden, the model will be initialized with empty lists, zero numeric values, empty strings, and false booleans.

initializePanel() Responsible for creating the GUI panel. If this method is not overridden, the type will not be available for selection in MetaCO_2P_2S. The contents of the panel must reflect the information in the provided model.

dumpToModel() If the GUI does not automatically update the underlying models, this method can be used to store the values from the GUI into the model. In CO_2P_2S, this method will be called when a dialog is accepted or even cancelled. If all is well, `null` should be returned. If an error occured, a string indicating the problem should be returned.

validateModel() This method is used to confirm that the contents of the model are correct. If they are, `null` should be returned. If they are not, an error string should be returned.

toString() Returns the string representation of the type. If `toString()` is not provided and the type is based on a String, Boolean, Integer, or Float value, the simple string representation of the value will be used. For Records and Lists, default strings will be automatically generated but they are not particularly helpful. If the type designer is creating a Record or List based type that might be displayed in a list or as part of a pattern GUI, the designer should override this method.

getSuperPanel() Given the name of an ancestor type, this method will retrieve the GUI panel used to display and manipulate the ancestor's structure.

getSuperString() Given the name of an ancestor type, this method will retieve the string generated by the ancestor type.

DataMetaGui

This class is used as the superclass for all of the parameter GUI classes. It provides default implementations of the four methods that need to be overridden. This class provides similar methods to `DataCopsGui`, only it does not include `toString()` or `getSuperString()`.

4.3.3 GUI Classes

DataJTable

In combination with `DataJTableModel`, the `DataJTable` type provides a flexible way to display lists of editable items. It is based on the Java `JTable` class. Figure 3.7 includes two of these tables.

DataJList

The `DataJList` class provides an editable list of arbitrary elements. The only requirement is that the list elements provide a sane `toString()` implementation. Figure 3.6 provides an example of the list.

4.4 Inheritance Issues

As explained in Section 4.1.3, the type hierarchy is linearized to provide a method resolution order. This resolution order is used in the initialization and validation of types.

First, the type system creates the linearization of the type hierarchy. For initialization, the type system traverses this hierarchy and initializes the types starting with the least specific. The fields of each type are initialized before the type itself is initialized. This allows more specific types to override the initialization of less specific types.

For validation, a similar process occurs, except the `validateModel()` method is called first for more specific types. This allows more specific types to produce more informative error messages. Note that there is no way to override the validation provided by the less specific types. As an example, the ResultType type allows all valid Java type names as well as the `void` type. In the Type type, which specifies types of variables, we do not wish to allow the `void` type. In this case, Type must inherit from ResultType, since Type's validation is more limiting.

No automatic inheritance occurs for the `initializePanel()` or `dumpToModel()` methods. When a type provides the `initializePanel()` method, it also signifies that it has a GUI available for use. Not inheriting the panel initialization leaves the type designer with the choice of providing a GUI. As well, since no GUI is provided by default, the type designer must explicitly choose an ancestor GUI in the case of multiple inheritance. Since `dumpToModel()` is functionally tied to `initializePanel()`, it makes little sense to allow it to be inherited independently of the panel initialization.

4.5 Type Storage

Types are stored in a single directory. This directory contains the `type.xml` file, any Java GUI files, and the code generator source file. Previously, types were scatted throughout the CO_2P_2S directory hierarchy.

Types are stored as XML as described in Appendix D. As well, when types are in use they are stored using the XML described in the `CopsData` namespace. The DTDs related to type storage and use are all provided in Appendix D.

4.6 Existing Types

This section provides a brief list of the existing types. The fundamental types are not included in this list as they have been described elsewhere.

ArgumentList A list of type/name pairs. This represents a list of arguments that would be passed into a method.

AugmentedMethod A complex type that combines the functionality of Method, MethodSuffix, MethodPrefix, and MethodGuard.

AugmentedMethodList A list of AugmentedMethods.

ClassName Represents a valid class name. Does not enforce the convention that the first letter of the class is upper-cased.

ClassOrInterfaceName Represents the name of a valid class or interface. A checkbox is provided to differentiate between classes and interfaces.

Exception An exception that may be thrown by a method. Represented simply as the name of the exception.

ExceptionList A list of elements of the Exception type.

Identifier A valid identifier.

IdentifierSuffix A valid suffix to an existing identifier. An IdentifierPrefix type is not necessary as it would follow the same rules as Identifier.

Method A method with a name, a result type, and lists of arguments and exceptions.

MethodGuard Represents a method that acts as a check to see if a delegated method should be called.

MethodPrefix Represents a method that can modify the parameters that are passed to a delegated method. It may perform additional operations before the delegated method is called.

MethodSuffix Represents a method that can modify the return value from a delegated method. It may also perform additional operations after the delegated method is called.

ResultType Represents any valid Java type including the `void` type.

ResultVariable Used for code generation, assisting in generation related to ResultTypes. Represents a named variable with a certain ResultType type.

Type Represents any valid Java type.

TypeWrapper Used with code generator to create and manipulate object wrappers for the Java primitive types.

Variable Stores a name and type that make up the identify of a variable.

VariableList A list of Variables.

4.7 Summary

This chapter provided further details about the type system. Design decisions and issues were examined. The API of the type system was described briefly, and the implemented types were listed. The next chapter discusses the code generation subsystem, also showing how types are used in the creation of application source code.

Chapter 5

Code Generation

To be of use, a generative design pattern must produce functional source code. The user of the pattern specifies the application requirements via options defined in the CO_2P_2S GUI. The pattern designer must build the pattern to take those options and produce a customized implementation. Previous chapters demonstrated the use of types for GUI display and input validation. This chapter will cover the use of types during code generation. The CO_2P_2S Generation Language (CGL) will also be covered.

5.1 Overview of Generation Subsystem

Figure 5.1 shows the flow of data within the code generation subsystem. To generate the pattern source used in the application, three groups of input are required. First, the pattern template itself is required. This would be composed of the Java source, annotated with CGL. The CGL annotations affect what Java is ultimately produced by the generation system. In addition to the pattern template provided by the pattern designer, the option values used to configure the pattern are needed. The application developer provides configuration information via the CO_2P_2S GUI. The configuration information is then passed to the code generator as an XML file containing the option values. As well, code fragments (such as the body of a method) may be provided by the application developer. The third group of input to the code generator is the type information. As discussed previously, each option value has a type which specifies the code generated for that option. The generation functions needed for each type are specified using Python, possibly with embedded CGL.

Once all of the inputs have been collected, the code generator produces compilable, yet specially annotated Java source. Jalopy [3] is used to do whitespace formatting. Finally, a post-processing step uses the special annotations and produces Java source and HTML templates. These HTML templates are used by the CO_2P_2S GUI to allow the application developer to enter code fragments and view the important parts of the output source.

Figure 5.1 shows five inputs to the code generator. The option value XML is generated

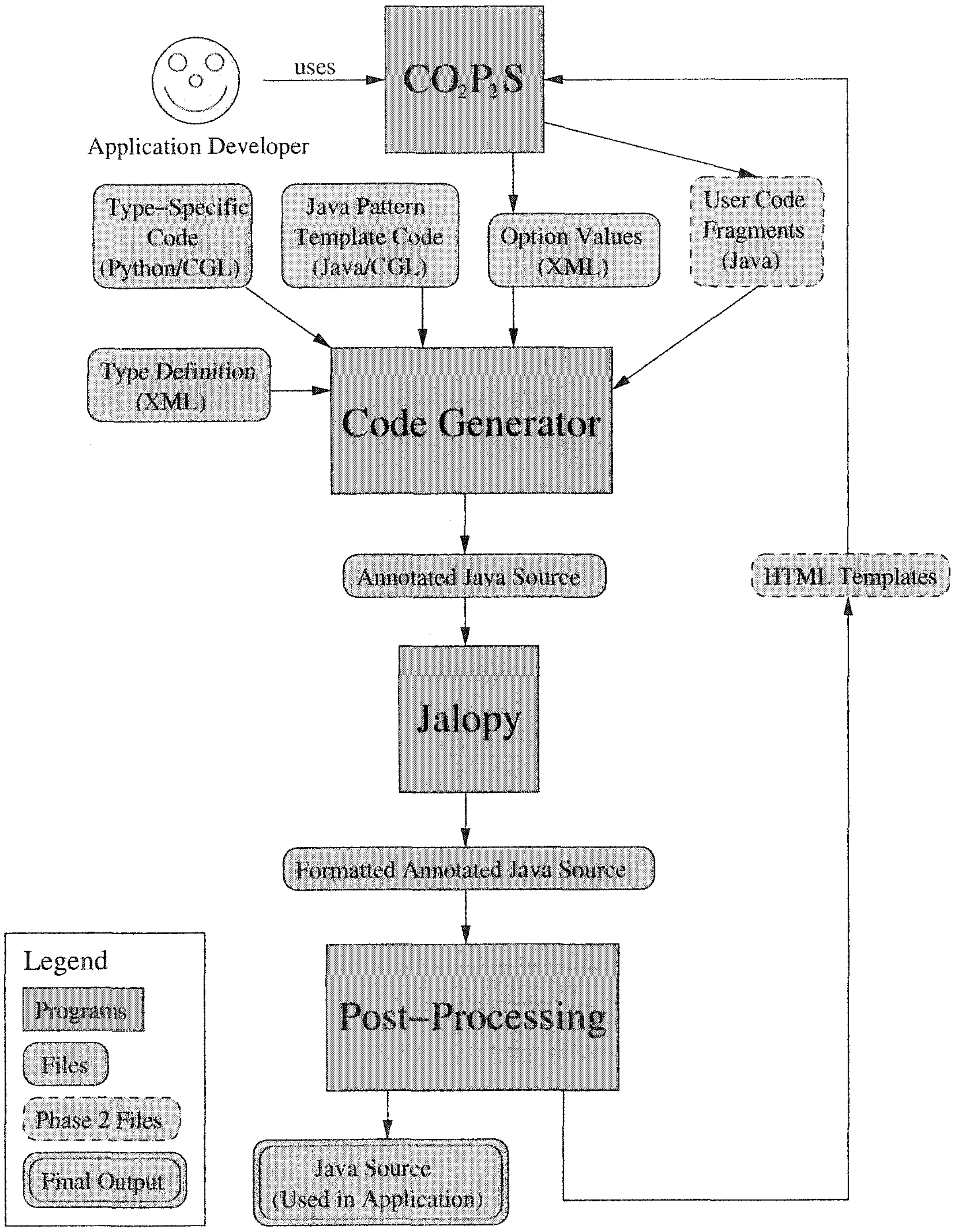
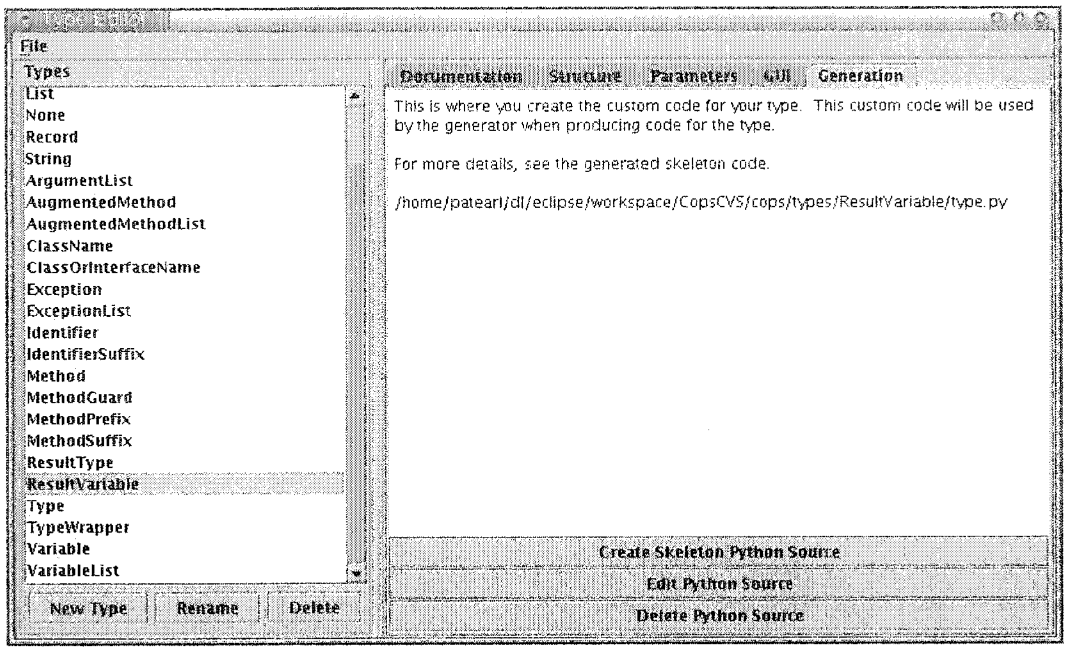


Figure 5.1: Code Generation Subsystem



This is where you create the custom code for your type. This custom code will be used by the generator when producing code for the type.

For more details, see the generated skeleton code.

/home/patearl/dl/eclipse/workspace/CopsCVS/cops/types/ResultVariable/type.py

Figure 5.2: Type Editor Code Generation

from the configuration information provided by the application developer. Aside from the option values, there are four other inputs to the code generator. One of these is the XML type definition, created by the type editor. The type definition contains information such as the ancestors and fields of the type. The other three inputs are the type-specific code, the pattern template code, and the user code fragments.

5.1.1 Type-Specific Code

Each of the option types within CO_2P_2S has the ability to generate code in a specific manner. Some abilities are provided by the CO_2P_2S system, while other abilities are provided by the type designer. Internal abilities allow the String, Boolean, Integer, and Float types to generate default representations of themselves. Designer-provided abilities come in the form of methods, which are attached to each type.

Figure 5.2 shows the panel where the code generation for the ResultValue type is specified. Just like the GUI panel discussed previously, there are three actions that may be performed. A skeleton file, containing informative comments and empty methods, can be created. The file can then be edited or removed. To see the generated skeleton file, consult Appenix F.

Inside the file are the methods provided by the type designer. There are two special methods, `init()` and `string()`. The `init()` method is used to prepare the type and might be used to create or modify fields within the type. The `string()` method is used to specify the default representation of the type. Other methods may be created, adding custom

```
--- Python ---

def arguments(self):
    """Return the names of the variables separated by commas."""
    return ", ".join(map(lambda x: x.name, self))


--- CGL ---

cgl(r'''

#macro arguments(self)
  #for arg in self
    #emit arg.name
    #if not last
      #emit ", "
    #end
  #end
#end

''')
```

Figure 5.3: Example Method from VariableList Type

functionality to the type.

Figure 5.3 shows an example of a custom method that is part of the VariableList type. The first example shows the method written using some of the features of Python. The second example shows how the method can be written in CGL.

Note that the CGL method is enclosed in `cgl(r''' ... ''')`. To avoid preprocessing Python code, embedded CGL is handled via the `cgl()` call in combination with a multi-line string.

The type files themselves are created by the type designer, the most advanced user of the CO_2P_2S system. Pattern designers would not need to concern themselves with the internal implementation of the types.

Unlike the Java GUI for the type, no compilation is needed for the Python file used in the code generator. The types are loaded at generation time, leading to a fast development cycle for the types themselves.

5.1.2 Pattern Template Code

The pattern template code is the annotated Java source created by the pattern designer. This template code generally contains the bulk of the information in a pattern; it is used to generate the source code for use in applications.

The majority of the template code will be in one or more `.java` files that reside in the

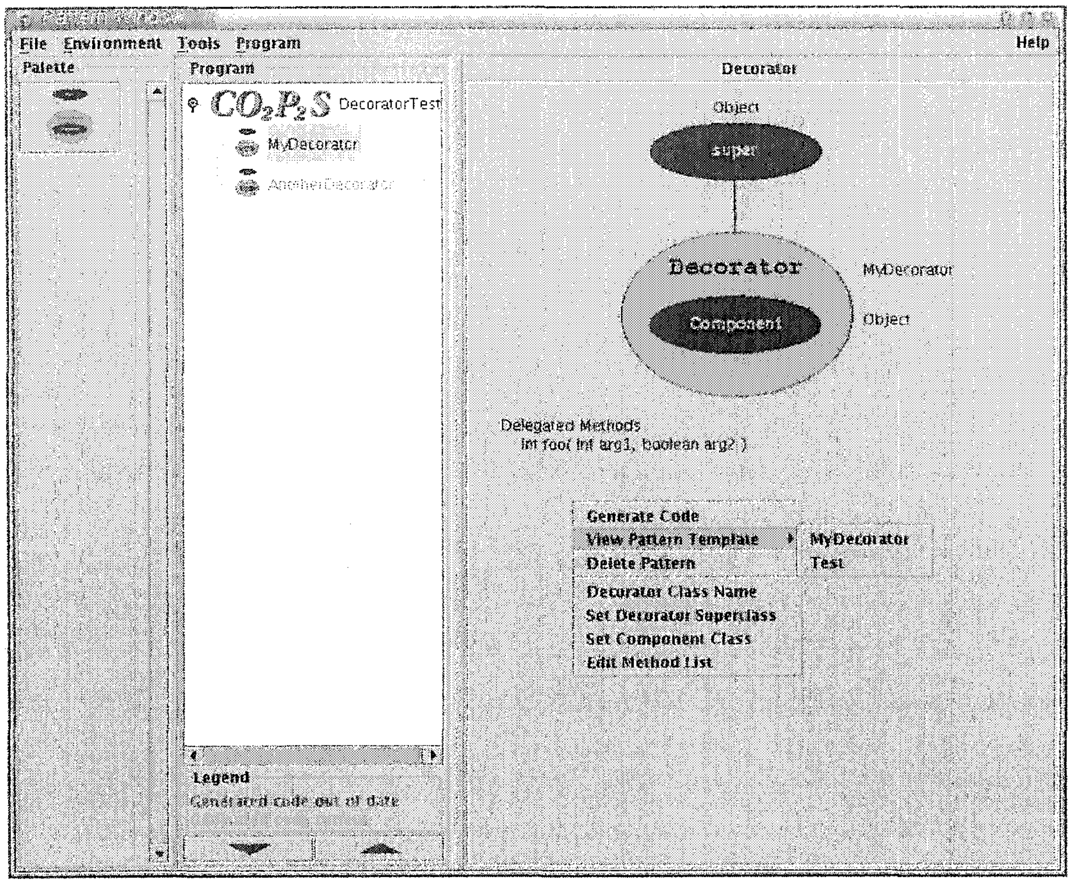


Figure 5.4: Decorator Template Example

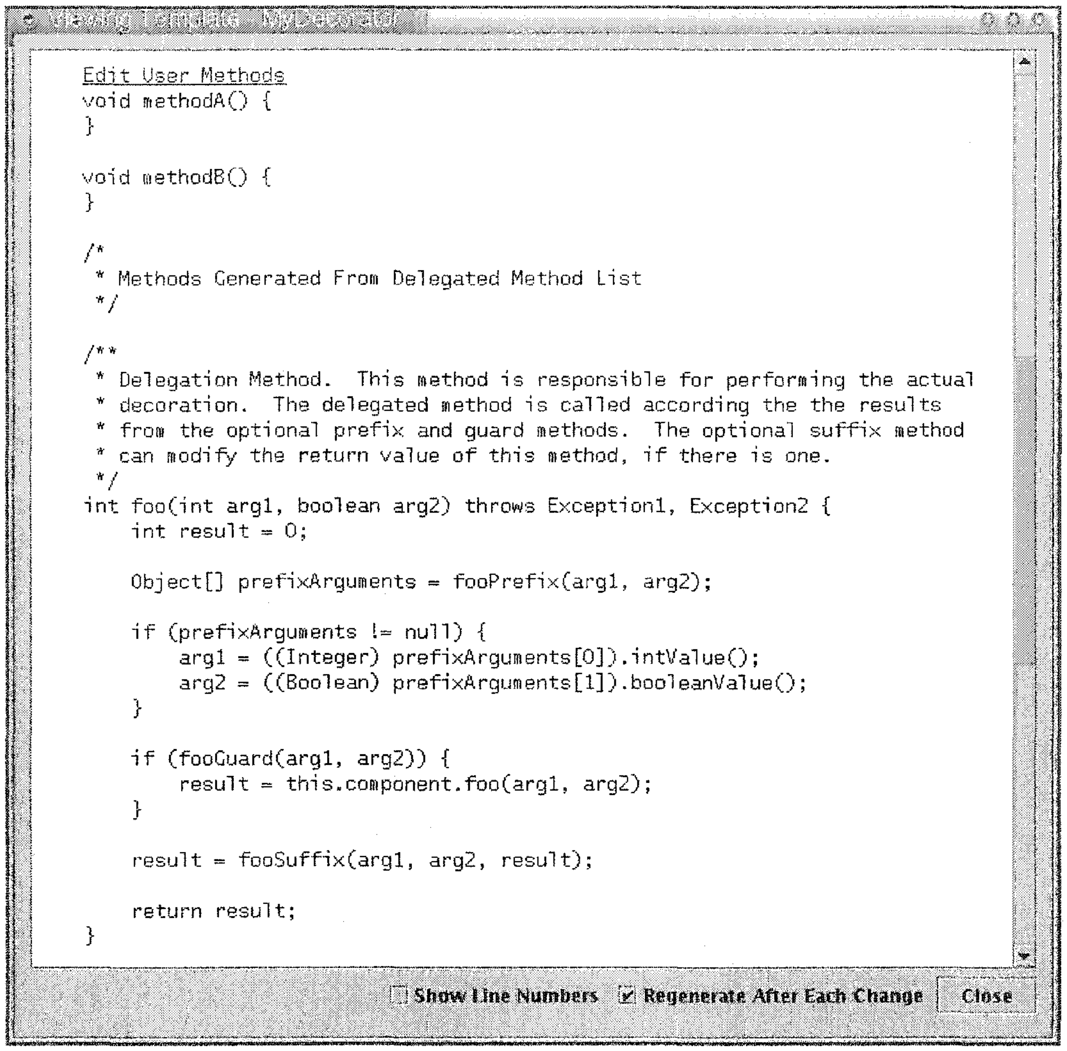
framework directory. When the generator is processing the code, it will examine all of the files in the framework directory. Note that there need not be a one-to-one relationship between input files and output files.

In addition to the regular Java files, there is also a pattern.cgl file which is used to store pattern-wide macros. If a macro is needed in multiple template files, it would be placed in pattern.cgl in the framework directory. Aside from the types themselves, this is the mechanism for sharing code and handling changes which crosscut multiple template files. If needed, pattern-wide convenience variables can also be created in pattern.cgl.

The annotations used in the Java code will be described starting in Section 5.3.

5.1.3 User Code Fragments

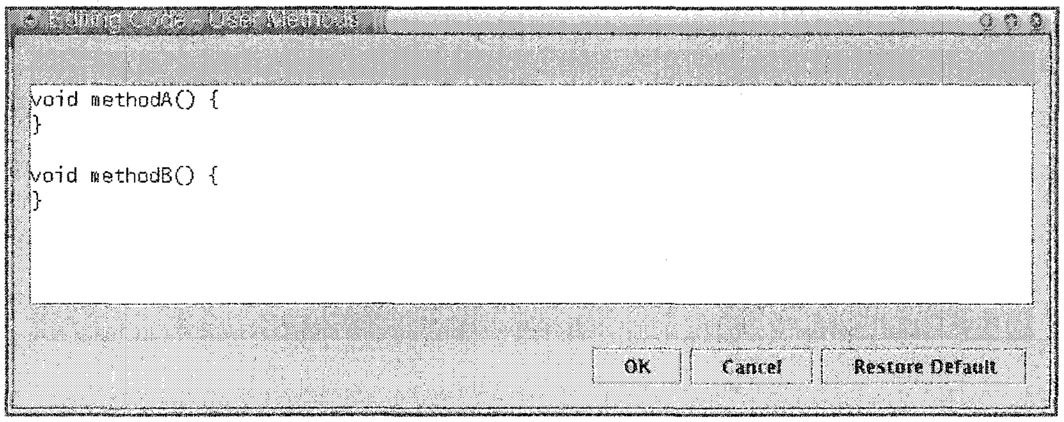
After the code has been generated, some classes will be available for viewing or editing. Figure 5.4 shows how those classes are accessed in the Decorator pattern. Note that a "Test" class has been added for demonstration purposes; it is not part of the Decorator itself. Once the CO_2P_2S user selects the class to view, a viewer will display the contents of the source. A viewer showing a portion of the Decorator code is shown in Figure 5.5.



```
Edit User Methods
void methodA() {
}

void methodB() {
}

/*
 * Methods Generated From Delegated Method List
 */

/**
 * Delegation Method.  This method is responsible for performing the actual
 * decoration.  The delegated method is called according the the results
 * from the optional prefix and guard methods.  The optional suffix method
 * can modify the return value of this method, if there is one.
 */
int foo(int arg1, boolean arg2) throws Exception1, Exception2 {
    int result = 0;

    Object[] prefixArguments = fooPrefix(arg1, arg2);

    if (prefixArguments != null) {
        arg1 = ((Integer) prefixArguments[0]).intValue();
        arg2 = ((Boolean) prefixArguments[1]).booleanValue();
    }

    if (fooGuard(arg1, arg2)) {
        result = this.component.foo(arg1, arg2);
    }

    result = fooSuffix(arg1, arg2, result);

    return result;
}
```

☐ Show Line Numbers ☑ Regenerate After Each Change

Figure 5.5: Viewing Decorator Template



```
void methodA() {
}

void methodB() {
}
```

Figure 5.6: User Methods

From the code viewer, the user can examine how the generated code functions. There is an underlined link at the top, indicating that a portion of the file is editable. When the user clicks on the link, the editor shown in Figure 5.6 is displayed. If the appropriate annotations are present in the source, the editor may display context information, allowing the programmer to orient themselves. For example, if editing a method body, the context information would likely include the method signature.

In order to display an editable template in the viewer, HTML is used. This means that the code generator system will need to generate two copies of the source: one in HTML form and one as plain Java. These two outputs can be seen as output of the post-processing step in Figure 5.1. After coming out of the code generator, the source code goes through a source code formatter (Jalopy). Since the source code formatter does not understand HTML, annotated Java is used instead. Once the code has been formatted, it is post-processed and converted to actual HTML. This HTML is then used directly in the CO_2P_2S code viewer. If the users performs any edits, files are created containing their code. These files are subsequently used by the code generator. Figure 5.1 has dotted lines around User Code Fragments and HTML Templates; this indicates that the items are not used in the first phase of code generation. In the first phase, the Java source is generated along with the first version of the HTML template. Using the HTML template in the code viewer, the CO_2P_2S user may add their own code fragments. These code fragments are then used in the second and subsequent cycles.

5.2 Output Formatting

The generated code should be as readable as possible. This means that whitespace should be used appropriately. There are a number of issues that require special whitespace handling.

- User code fragments may required indenting. Though the user may have properly indented their own code, the fragment may require additional indenting if it is within an indented block of the source.
- When a macro returns code, we would like that code to be indented according to the surrounding indentation level.
- Sometimes indentation is optional. For example, we may conditionally include an `if()` statement in the code. If the statement is included, the code it affects should be indented.

Some of these issues can be handled in the code generator itself, but it leads to special cases and fragile whitespace handling. The solution used here is to run the Jalopy source code formatter on the output from the generator. The Jalopy source code formatter parses the source and reconstructs it from its individual tokens. This alleviates whitespace issues and increases consistency in the generated code. If the user wishes to change the output style, they may edit the `generator/jalopyConventions.xml` file through a GUI tool provided with Jalopy.

Because most source code formatters do not understand HTML, an extra step was required to handle the links allowing the user to add their own code fragments. Instead of generating the HTML link right away, a special one line Java comment is added to the source. The source code formatter then properly indents the one line comment. When the source code has been formatted, the special comments are converted to HTML links. The rest of the source is converted to HTML, escaping characters and adding tags as needed.

Sometimes the user may notice that the whitespace formatting is not correct. This would be due to an error in the generated Java source that breaks the Jalopy parser. That effect should be seen only by the pattern designer during the creation and debugging of the pattern.

5.3 Language Syntax

One of the problems with the original generation system was that it spread the code generation for a single pattern file throughout many different input files. Method bodies were contained in one file, the rest of the class source in another, and there was Java source code that generated the non-trivial code fragments. The new code generation system allows for the integration of all of these pieces within a single file. This makes the pattern template much easier to read and understand.

There are two ways that CGL annotations can be added to Java source. Lines that start with # are treated as CGL statements. As well, the @ character can appear anywhere and the value of the following expression will be inserted into the output source. The syntax of CGL is simple and though it was designed with Java in mind, is not tied to any particular language.

5.3.1 # Command Syntax

When a # is encountered at the start of a line, ignoring whitespace, it indicates that everything up to the end of the line should be parsed as a CGL command. If needed, ## can be used to generate a single #. This escape feature would be useful in generating C code when lines like `#include <stdio.h>` are needed.

5.3.2 @ Expressions

@ is used to insert the value of an expression into the Java source. In order to handle syntax ambiguity, the expression can be wrapped in braces. Having braces also allows the

```
--- Informal Grammar ---

    '@' PRIMARY

    '@{' python-expresssion '}'

    PRIMARY =   python-identifier
              | PRIMARY '.' python-identifier
              | PRIMARY '.(' python-arguments ')'
              | PRIMARY '.[' python-slice-expression ']'


--- Examples ---

    @var
    @var[3]
    @var.method('argument')
    @var.field
    @function()[4].field2
    @{var + 1}
    @{var}
```

Figure 5.7: Syntax of @ Expressions

use of an arbitrary expression. Figure 5.7 shows the informal grammar and some examples of @ expressions. For the complete grammar of the Python non-terminals, see [7]. Note that the contents of @ expressions are not CGL, but are plain Python expressions. Any expression used in a form such as `@{EXPRESSION}`, can also be used in locations such as `#if EXPRESSION` or `#for VAR in EXPRESSION`. No @ need be attached to those expressions; @ is only used to insert expression values into the Java source.

5.3.3 Choice of Characters

@ and # were chosen due to their limited use in Java and Python. In Java, @ is only officially used in Javadoc comments. # is not used in the Java language. If @ occurs in Java strings, it requires escaping. The parser will ignore # in Java strings, since it is not at the start of a line. In Python, @ is also unused, where # is used only for comments.

The { } characters were chosen to delimit @ expressions when needed. Braces are only used in Python to create dictionaries. Braces are used in a similar manner in languages such as TCL, Perl, and various shell script dialects. `[ ]` and `( )` were rejected due to their frequent use in expressions. < > was rejected, since its use in comparison conditions makes parsing difficult. A matching pair of characters was used to increase the clarity of the code. Starting and ending an expression with an @ character can make lines with multiple expressions difficult to read.

```
#rem This is a comment.
#rem
#rem Hopefully your comments are more useful than this.
```

Figure 5.8: Example of `#rem`

```
There were @numberOfPigs pigs.
The wolf ate one.
@{numberOfPigs - 1} pigs remained.
```

Figure 5.9: Example of `@expression` Replacement

5.4 Language Features

This section will detail the requirements on a generation language and how CGL fulfills those requirements. A simple example of each operation will also be shown.

5.4.1 Documentation

The ability to provide comments in the meta-language is helpful for those viewing or editing the pattern template. It can be used to understand design decisions. Relying on the comment features of the generated language is not sufficient, since the intended audience of the comments is different.

Language Feature

The `#rem` command, short for remark, allows the insertion of random comments within CGL code. See Figure 5.8 for an example.

5.4.2 Value Replacement

The ability to replace a special string of text with another string is essential in a generation language. For example, the name of a method may be based on input provided by the user.

Language Feature

To implement the replacement, CGL provides the @ technique for inserting expressions directly into the generated source. @ takes the string value of the expression and replaces the @ expression with the string value. Figure 5.9 shows an example of string replacement in CGL.

```
#if sky.colour = "blue"
  System.out.println("It's a nice day outside.");
#elif sky.colour = "red" and hour > 12
  System.out.println("There is a sunset occuring.");
#else
  System.out.println("The sky is @sky.colour.");
#end
```

Figure 5.10: Example of `#if`

5.4.3 Conditional Code Inclusion

When generating code, it is often necessary to select between two or more code generation alternatives. Sometimes, we may want to entirely omit a section of code.

Language Feature

CGL solves the conditional inclusion problem in a simple manner, similar to `#ifdefs` in C preprocessing. In the `if` expression, standard operators such as `not`, `or`, `and`, =, <, >, `!=`, <= and >= may be used. See Figure 5.10 for an example.

```
#if <conditional-expression>
#elif <conditional-expression>
#else
#end
```

5.4.4 Repetitive Code Inclusion

The ability to repeat a section of code during code generation is very useful. However, instead of repeating the segments verbatim, we usually wish to modify the code fragments in some manner.

Language Feature

In CGL, the iteration is performed explicitly through the use of a `for` statement. The loop variable will sequentially be set to each element in the list expression during each iteration of the loop. Figure 5.11 provides an example of this command.

```
#for <loop-var> in <list-expression>
#end
```

Inside the loop, there are three additional variables available. If you wish to use these variables within nested loops, you will need to `#assign` them to temporary variables.

`first` Boolean variable indicating if this is the first iteration.

`last` Boolean variable indicating if this is the last iteration.

```
--- Iterating Over the Contents of a List Variable ---

    #for item in listvar
      @item is at index @index in the list.
    #end

--- Iterating Over Constant Lists ---

    #for suit in ['Hearts', 'Clubs', 'Diamonds', 'Spades']
      #for value in range(1, 13)
        There is a card in @suit with a numeric value of @value.
      #end
    #end

--- Result of Second Example ---

    There is a card in Hearts with a numeric value of 1.
    There is a card in Hearts with a numeric value of 2.
    ...
    There is a card in Spades with a numeric value of 13.
```

Figure 5.11: Example of `#for`

index Integer variable indicating the iteration count, starting from zero. This might be used in operations such as splitting a long list of items based on a modulus.

5.4.5 Keeping Similar Code Together

Generated code often consists of similar or identical pieces of code. It is desirable to have a single generalized instance of the code rather than maintaining multiple copies.

Language Feature

Aside from the types accessible from CGL, macros are provided to support common code. The provided argument list may be empty, but the parentheses must be present. The argument list is in the same form as those for Python functions. See Figure 5.12 for an example. In the example, the i and j variables appear within single quotes. The quotes are used to indicate to CGL that a literal value of "j" is desired, rather than the contents of the j variable.

```
#macro name ( <argument-list> )
#end
```

5.4.6 Aliases and Modification

Sometimes names of variables within the code generator may become verbose. It is helpful to have a way to access the data with a shorter name.

```
--- Macro Definition ---

    #macro loop(var, from, to)
      #if from < to
        for(int @var = @from; @var <= @to; @var++)
      #else
        for(int @var = @from; @var >= @to; @var--)
      #end
    #end

--- Macro Usage ---

    @loop('i',1,5) {
      @loop('j',3,1) {
        System.out.println("Numbers: " + i + " " + j);
      }
    }

--- Result ---

    for(int i = 1; i <= 5; i++) {
      for(int j = 3; j >= 1; j--) {
        System.out.println("Numbers: " + i + " " + j);
      }
    }
```

Figure 5.12: Example of #macro

```
--- Example ---

    #assign methodCopy method
    #assign methodCopy.name "newMethodName"

    #rem  Create a new instance of ResultType and make it the "void" type.
    #rem  Change the result type of the method copy.
    #assign methodCopy.resultType ResultType("void")

    @method

    @methodCopy

--- Result ---

    int myMethod(int arg1, boolean arg2)

    void newMethodName(int arg1, boolean arg2)
```

Figure 5.13: Example of `#assign`

Also, it may be easier to modify an existing object than to create an entirely new object. As an example, a method signature may be close to what we want, however we need to change the name. If we do not want to destroy the original version, we need some way to copy it and modify the copy.

Language Feature

The `#assign` statement is used to copy the value of an expression and store it into a given identifier. The identifier may be part of an existing object. Note that the fields of types are all documented within the type editor, allowing for easy editing. Figure 5.13 provides an example. Note that the results of the example depend on the content of the method variable. Arbitrary values have been provided for demonstration purposes.

```
    #assign <identifier> <expression>
```

5.4.7 Output Files

In the previous system, a single input file always created a single output file. It is sometimes desirable to produce zero or more than one output files.

Language Feature

The `#output` command allows you to specify the base name (without the `.java` or `.html` extensions) for the output files. When the generator is finished, files with `.java` and `.html` extensions will have been generated with the given name as a base name. If no output file is specified, or if the output file is an empty string, no output will be produced. See Figure 5.14 for an example.

```
#output <basename>
```

5.4.8 Displaying Generated Source

Certain files in the generated framework may be important to the application developer, where other files may not be.

Language Feature

The `#template` command allows the pattern designer to specify that this file is important and should be visible from the pattern template menu in CO_2P_2S. By convention, the `#template` command should come right after the `#output` command, as demonstrated in Figure 5.14.

5.4.9 Editing Generated Source

In some frameworks, we want certain parts of the source to be editable. In the Wavefront pattern [9], the bodies of certain methods are editable. The editable methods in the Wavefront pattern contain sequential code that performs a calculation on an individual cell. The pattern framework then takes those sequential methods and runs them in a parallel manner.

Language Feature

CGL provides six commands for the purpose of providing edit capabilities to the application developer. `#user_context` is used, in combination with one of the other commands, to provide context information to help the user with their editing. The context information will be displayed above and below the source being edited. `#user` and the other four commands are used to create a user-editable code segment. The given name will be generated as an HTML hyperlink, displaying: Edit `<name>`. The given name must uniquely identify the code segment within its scope. Normally, the scope of the user code segment is per-file. In that case, the names need only be unique within the file itself. However, if you want the editable segment to be reachable from all the source files, use one of the `_global` varieties instead. Any place that uses a `#user_global` with the same name will use the same code. If you wish to provide default code for the editable section, use one of the `_default` varieties. The default code should be placed between the `#user_default` and `#end` statements. The `#user_default_global` and `#user_global_default` commands combine the functionality of the `#user_default` and `#user_global` commands; there is no difference between them. Figure 5.15 provides two examples of these commands. In the example, square brackets

```
--- Example ---

    #for class in ["ImportantClass", "ClassA", "ClassB"]
    #output class

    #if class == "ImportantClass"
      #template
    #end

    class @class {
        void print() {
            System.out.println("This is in file @class.");
        }
    }

    #end

--- Result ---

    ImportantClass.java:  (Viewable in GUI by Application Developer)

        class ImportantClass {
            void print() {
                System.out.println("This is in file ImportantClass.");
            }
        }

    ClassA.java:  (Not Viewable in GUI)

        class ClassA {
            void print() {
                System.out.println("This is in file ClassA.");
            }
        }

    ClassB.java:  (Not Viewable in GUI)

        class ClassB {
            void print() {
                System.out.println("This is in file ClassB.");
            }
        }
```

Figure 5.14: Example of #output and #template

```
--- Example ---

    #user_global "Import List"

    #user_context
      int addNumber(int arg) {
    #user_default "Body of addNumber Method"
          return arg + 100;
    #end
      }
    #end

--- Result ---

    [Edit Import List]
    import myconstants;

    int addNumber(int arg) {
        [Edit Body of addNumber Method]
        return arg + myconstants.Constants.SUITABLE_VALUE;
    }
```

Figure 5.15: Example of #user

indicate a hyperlink that links to an editor for the appopriate code fragment. The example shows the HTML displayed in the CO_2P_2S GUI after the user has entered some values.

```
#user_context
#end

#user <name>

#user_global <name>

#user_default <name>
#end

#user_global_default <name>
#end

#user_default_global <name>
#end
```

5.4.10 Whitespace Control

Though not much of an issue when the code can be run through a comprehensive beautifier, in general there is sometimes a need to have greater control over the places where whitespace is generated. For example, if we are generating a list of arguments, we do not want each argument to be on its own line.

Language Feature

By passing a string to the #emit call, the data will be output with no surrounding whitespace. See Figure 5.3 for an example of this command.

```
#emit <string expression>
```

5.4.11 Extension System

When learning a language, the fewer features there are, the easier it is to learn. CGL itself is fairly small, facilitating fast understanding of the system. However, being small also means that some desirable features cannot be included in the core language. Being able to add features without making the core language more complex is desirable. In addition, it is better if those extra features are as easy to find and learn as possible. Types serve to package common functionality and make it easier to find.

Language Feature

All types in CGL are first created in the type editor, described previously. In the type editor there is documentation for each type. When a user has learned the core of the CGL language, they will be equipped to handle many code generation tasks. If a task arises where the core language is inefficient, the user should look to the documented list of types within the type editor for a solution. The type editor's list of types is the only place the user would need to look to find new functionality.

Types are instantiated by using their name followed by a pair of parentheses. The parentheses may have arguments in them, depending on the type. The primitive types, String, Boolean, Integer, and Float, must have a single argument passed to their constructor. List constructors accept an optional list argument. Record types accept a list of zero or more fields. See the example below for syntax.

Unlike primitives in Java, the primitives in CGL are objects themselves. This means that the types may be treated as primitives for various operations such as addition, yet they may also have extra functionality or state attached to them in the form of methods and fields.

For examples of instantiating and using types, see Figure 5.16. In the results section, arbitrary contents for `varList` are provided. The first two examples show two different ways of doing nearly the same thing. Though the first technique is simpler and works in most cases, there may be some cases where extra functionality of the `Identifier` or `Type` types is needed.

```
--- Examples ---
    #rem  Creating a new Variable, a type with two fields:
    #assign var Variable(name='newVar', type='int')

    #rem  Or, using the correct types for the fields:
    #assign var Variable(name=Identifier('newvar'), type=Type('int'))

    #rem  Creating a new ResultType, which is a string-based type:
    #assign var ResultType('void')

    #rem  Accessing a user-defined function of the VariableList type,
    #rem  assuming varList if of type VariableList:
    @varList.arguments()

    #rem  Using a type's default string representation:
    @varList

--- Results of Last Two Examples ---

    iVar, bVar, oVar

    int iVar, boolean bVar, Object oVar
```

Figure 5.16: Type Usage

5.4.12 Complex Operations

The built-in operations presented so far are not sufficient for a fully functional code generator. As well, access to higher level operations and data structures could simplify the CGL code considerably in some cases.

Language Features and Examples

One notable category of functionality not presented yet is the ability to modify strings. CGL provides string manipulations through the Python `str` type. If an object is a string, you may call string operations directly on the object itself. However, if the object is not yet a string, it will first need to be converted. The Python documentation contains full details [6]. Examples of string operations can be seen in Figure 5.17.

Sometimes the conditions of execution or contents of a particular code fragment can be complex. Mathematical expressions are a powerful way to concisely express complex relationships. Instead of a table containing a large number of entries, a mathematical expression can be used to represent the relationship between the entries. CGL allows access to the usual Python expressions. These expressions can be placed where one would expect to find an rvalue. Figure 5.18 provides examples.

Another useful class of code generation tools are general purpose data structures. These

```
--- Example ---

    #assign s 'Some String'
    #assign i 102030

    #rem  Uppercase s before placing the result in the code.
    @s.upper()

    #rem  Remove all the zeros from an integer, after converting it to a str.
    @str(i).replace('0','')

--- Result ---

    SOME STRING

    123
```

Figure 5.17: String Operations

```
    @{10 * (var + 1)}

    #if x > y-10 and bitone ^ bittwo
        ...
    #end

    #for i in range(r+3)
        ...
    #end

    #assign u v/10.0
```

Figure 5.18: Mathematical Expressions

might include dictionaries or sets. A need for these sorts of data structures was observed in advanced patterns such as the Wavefront. The `#python` directive allows direct insertion of Python code. The `#exec` directive allows insertion of a single line of Python code. Figure 5.19 shows an example of using sets and dictionaries for the Wavefront's compass dependency. Also included is an example of list creation and appending.

5.5 Generator Design Issues

Having already discussed the language itself, the upcoming section elaborates on some of the important design decisions behind the language.

5.5.1 Composition versus Transformation

In creating a language for code generation, one of the most important decisions is whether to make the language compositional or transformational [17]. A compositional language relies on forward refinement, moving from a higher level of abstraction to a lower one. See Figure 5.20, adapted from [17], for an example.

A transformational generator is capable of performing horizontal transformations which cross module boundaries. Transformational generators may also perform oblique transformations which combine horizontal and vertical refinements (see Figure 5.20).

CGL is primarily a compositional system. Though the compositional approach of CGL can make it tricky to generate the shortest code possible, it fits most generation tasks well. Creating a reasonable transformation system would have taken a great deal more time than the current compositional system.

There are a number of benefits to the compositional approach. The techniques used for code generation in CO_2P_2S will be familiar to anyone who has used an object oriented language and a macro processor. This lowers the learning curve for the system. Because transformational systems are used to reshape code, the pattern designer would need to learn not only how to modify the code, but how to indicate which portions of the original code are to be modified. The use of the shaping tools is not always obvious, as it can involve referring to grammar elements of the language being transformed [14, 16].

Transformational systems are often dependant on the language they are generating. A transformational system for Java would have knowledge of Java syntax and semantics. Though this makes it more efficient at producing code for Java, the same system could not be used to process C code.

Transformational systems allow for better separation of concerns, but they also have the potential to obscure the operation of code. In a compositional system, any changes to the code are clearly visible in line with the rest of the code. With a transformational approach, changes may be performed on distant code fragments, with little clue that such

```
--- Example ---

    #python
    dirDict = { 'north' : Set( ['north', 'northwest', 'northeast'] ),
                'south' : Set( ['south', 'southwest', 'southeast'] ),
                'west'  : Set( ['west', 'southwest', 'northwest'] ),
                'east'  : Set( ['east', 'southeast', 'northeast'] ) }

    userDirections = Set ( ['northeast', 'east', 'southeast'] )
    #end

    #rem  Add north to the set of user directions.
    #exec userDirections.add('north')

    #rem  Calculate the set of directions that do not include north.
    #assign noNorth  userDirections - dirDict['north']


    #rem  Create a list and append to it.
    #assign l ['one', 'two', 'three']
    #exec l.append('four')

--- Result ---

    Contents of userDirections after first #exec:

        Set of ['northeast', 'north', 'east', 'southeast']

    Contents of noNorth after #assign:

        Set of ['east', 'southeast']

    Contents of l after final #exec:

        ['one', 'two', 'three', 'four']
```

Figure 5.19: Directly Including Python

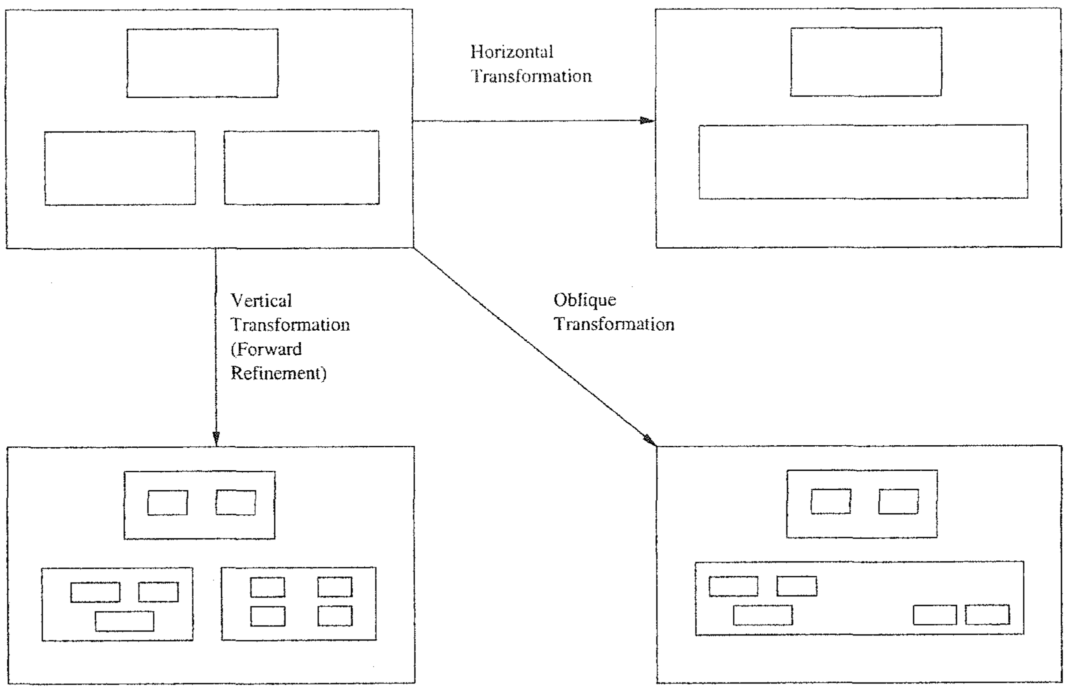

Horizontal Transformation
Vertical Transformation (Forward Refinement)
Oblique Transformation

Figure 5.20: Transformation Types

an operation is occuring. If a particular transformation needs to be split into two or more code fragments, CGL requires that those pieces be generated separately, perhaps through the use of macros or types. A transformational system would be able to separate individual aspects in a cleaner manner [24].

5.5.2 Size of CGL

The core of the CGL language is fairly small, acting as a macro processing layer with loops and other simple operations. Aside from that, it acts as a layer of glue, exposing functionality from the underyling scripting language.

Instead of adding many unfamiliar features, CGL seeks to leverage the power of an existing scripting language. That said, most code generation can be done with almost no knowledge of the underlying language itself. This approach is similar to those found in [13] and [26].

CGL has a simple syntax and only provides the common and necessary operations for code generation. This means that a user can quickly learn how to generate most code with the system. However, when complexity of the generated code threatens the simplicity of the template, the advanced data structures and operations of the scripting language can be used to simplify the code. Instead of inventing additional language features in CGL, the already well-considered features of the underlying language were used.

5.5.3 Choice of Language

For an underlying language, using a low level language, requiring the user to worry about memory management or string length checking, would be unacceptable. Clearly, a high level language must be used; however, there are many high level languages. Ruby, Scheme, TCL, Perl, and Python are all widely used languages which were considered.

Because of the object-oriented nature of the types, a language with native objects is desirable. In addition, dynamic manipulation of those types makes the implementation of the generator easier. Python and Ruby are the most clean and poweful in the category of objects.

During code generation, string manipulation occasionally needs to occur. Many scripting languages, including those above, provide powerful string manipulation facilities.

The presence of data structures such as lists, dictionaries, and sets were an important criteria, as they had demonstrated their usefulness in avoiding complexity. Along with the data structures, dynamic types were desired to avoid constant casting of results from container operations. Python not only provides those data structures, but also handy operations such as list splice operators, and list map and apply functions.

Python is well known for its ease of use. It takes only hours to get started writing useful Python programs. The syntax is simple and clean.

There was also excellent support for writing the code generator itself using Python. Instead of running the code using its own machinery, the CGL parser converts the code to Python and then runs the Python. Some Python tools that were useful during code generator construction were reflection, runtime stack examination, runtime object field modification, and inter-file code execution.

Though CGL itself is mostly language agnostic, Python was selected due to its ease of use and the general clarity of Python code. Using a single language provides a common ground for pattern designers viewing the work of other designers.

5.6 The New Decorator

Appendix B shows the template that the pattern designer wrote for the new Decorator pattern. The next chapter will provide a comparison of the old and new decorators, showing how the CO_2P_2S system has been improved.

Chapter 6

Evaluation and Conclusion

This chapter will examine the improvements brought about by the new code generation system. First, a quantitative evaluation of the Decorator pattern is presented. Included next is a summary of how the initial goals of the research were met, leading to an improved pattern creation process. Future work is discussed before the dissertation concludes.

6.1 Evaluation

This section provides a comparison of the Decorator pattern under the old and new CO_2P_2S systems. As well, improvements to the type system will be highlighted.

Figure 6.1 shows a quantitative comparison of the number of lines of code needed to implement the Decorator pattern in the old and new systems. In determining lines of code, the formatting of the source was standardized using a formatter (Jalopy) and by removing comments and blank lines. It should be noted that the original Decorator GUI provided the ability to automatically import the interface of an existing class. Since there was not time to implement automatic interface importing in the new version, the code for importing was removed for comparison purposes.

The most important result to notice is the reduction of the Decorator-specific code to 12% of its previous level. This new Decorator, in Appendix B, is much shorter than the old Decorator, shown in Appendix A. Also of note is the fact that the original Decorator pattern spread the static code for a single class among four different files, where the new system needs only one.

The number of possibilties for code reuse has increased. While the primary type originally used for the Decorator pattern was only useful in the Decorator pattern, the types provided now are more flexible and could easily be used within a pattern such as the Composite.

Though achieving approximately the same level of functionality, the types in the new system are only 45% the size of the types in the old system. In addition to being smaller,

Category	*Original System*	*New System*
Decorator-Specific Java GUI Source	194	0
Decorator-Specific Java Generator Source	173	0
Decorator-Specific GUI and Generator Source	40	0
Decorator-Specific Template Source	44	56
Decorator-Specific Total Lines of Code	451	56
Reusable Java GUI Source	818	484
Reusable Java Generator Source	71	0
Reusable Java GUI and Generator Source	641	0
Reusable Type Python/CGL Source	0	216
Reusable Type Total Lines of Code	1530	700
Total Lines of Code	1981	756
Template Files for Decorator Class	4	1
Number of Reuseable Types used in Decorator	4	23

Figure 6.1: Decorator Pattern Comparison

the unified API allowed for some additional functionality and consistency in the types. As an example, a "Cancel" button was added to the CO_2P_2S GUI to allow users to back out changes they may not have wanted. Previously, no such button existed as it required manual creation within every available type.

The number of reuseable types indicates that the granularity of the types is now much smaller. The new types are easy to combine and reuse in alternative combinations. The existing types did not have such flexibility. In order to implement the Decorator-specific type, the programmer needed to understand the API for the method list type and extend it for use in the Decorator method element type. In the new system, the types are combined more as building blocks rather than as extensions to existing frameworks. Instead of having a single type perform code generation for a single pattern option, the new system lets the programmer use multiple types during the code generation process.

During the recreation of the Decorator pattern, an additional optimization was discovered. Instead of having a prefix method for each argument, a list was used to return modified method values. Because Java 1.4 does not provide auto-boxing (conversion of primitives to objects), a `TypeWrapper` type was created to simplify the repetitive task. `TypeWrapper` is another example of a type that can be easily reused.

Learning the new types is easier than learning the old types. Documentation for the types has been centralized into the Type Editor. If the original types were to be extended, the programmer would need to analyze the structure of a large type involving hundreds of lines of code. In the new system, each type can be discovered on its own, and then combined into larger, more powerful types. Also, because the new system has many small generic types, the foundations for new types are more likely to be available.

6.1.1 Pattern Creation Process

In addition to the decrease in the quantity of code required, there were also improvements in the entire pattern creation process. This section compares the original pattern creation process with the new one.

Old System

Presented here, are the steps needed to create a typical pattern using the old CO_2P_2S system.

1. Determine options needed to configure the pattern for use in various applications.
2. Provide MetaCO_2P_2S with the following information:
   (a) Name of pattern.
   (b) Path to directory containing images used in the pattern's GUI.
   (c) Name of Java package containing extra classes used in the pattern GUI and for code generation.
   (d) List of all classes used in the pattern framework. In addition to class names that were user-specified, the system required the pattern designer to specify the fixed list of classes with names derived from the user-specified classes.
   (e) List of options used to configure the pattern. Other customization information was entered based on the type of the option.
   (f) Finally, a GUI for the pattern is defined using text and image elements.
3. The source code for the pattern must be provided. During this phase, the programmer either uses the very limited macro language or creates the resulting Java source using other Java code. As well as providing the outline for each class, the programmer needs to provide separate files for each method body. Methods could not be included inline with the class definition.
   (a) Write Java template using limited macro language.
   (b) Write Java for generator to create custom code of any complexity.
   (c) Create method bodies in separate files. These files were named with a specially formatted version of the method signature.
4. Though not always, it was frequently the case that the pattern designer required additional types and GUIs to represent the pattern options. If they were needed, the pattern designer learned the CO_2P_2S API and created Java source to perform GUI display and code generation for custom types.

New System

This section looks at the steps of pattern creation in the new system and compares them with the original steps.

1. Determine options needed to configure the pattern for use in various applications. This step remains essentially unchanged from the previous version of CO_2P_2S.

2. Provide MetaCO_2P_2S with the following information:

   (a) Name of pattern. The previous version also required the user to specify a directory for images and a package for user classes. Since each pattern has been collapsed into a single directory, the images for the pattern reside in a standard location that need not be specified. As well, the previous version also required the input of a Java package name where extra classes would be located. Now, no pattern-specific Java source need be written at all.

   (b) The MetaCO_2P_2S user no longer needs to specify the entire list of classes needed in the pattern. The names of classes are simply treated as regular options.

   (c) List of options used to configure the pattern. This step remains largely unchanged.

   (d) Finally, a GUI for the pattern is defined using text and image elements. The mechanisms behind the GUI display are more consistant in the new version. In addition, the actual specification of the GUI has been simplified slightly with the removal of unnecessary options.

3. The source code for the pattern must be provided.

   (a) Write a Java template using the CGL meta-language. Though the CGL language has a simple syntax and a limited number of built-in commands, it is far more flexible and powerful than the previous macro language. In addition, many types are available to assist in generation tasks. Instead of writing custom Java code for non-trivial templates, CGL can handle most cases. If additional power or complex data structures are needed, mechanisms have been provided to easily include arbitrary Python code.

   (b) The bodies for the methods are now included inline with the rest of the class definition. This makes it easier to read the class and understand its function.

4. With the increased number of available types, custom types will be needed far less frequently. If a custom type is needed, the architecture now allows for simple combination of existing types. The new types are smaller and are focused on GUI display

and basic code generation. The old types tightly coupled GUI display and code generation, creating many pattern-specific types. As an example, the Decorator method GUI could not be reused in a Composite pattern, since the Java code also performed code generation specific to the Decorator pattern. The code generation for the types within the new system is focused on the core tasks of the specific type and CGL is able to combine these core types in more flexible and powerful ways than was previously possible.

6.2 Work Completed and Goals Accomplished

The overall goal of this research was to simplify the creation of generative design patterns, specifically within the context of the CO_2P_2S system. To achieve this goal, the following tasks were completed:

- A system of types was created for use in generative design patterns. These types are object-oriented and are manipulated through a type editor, which was also created during the course of the research. Previously, creating a type involved the creation of large amounts of Java code. Now, only small amounts of code are needed for the creation of a new type.
- Along with the system of types, many types were created for use in constructing design patterns. Most of these types have parallels in the code they are creating, such as a `Method` type, representing a method signature, or a `Variable` type representing a named variable with a certain Java type. To learn about these types, one can visit the type editor where the documentation for each type is displayed in an editable format.
- The CO_2P_2S Generation Language was created. The language is capable of manipulating source on its own, but is intended to be used in combination with the many available types.
- Finally, CO_2P_2S and MetaCO_2P_2S were rewritten to support the new type system and code generator, creating a complete generative design pattern system.

6.2.1 Completed Goals

Generative design pattern systems, such as CO_2P_2S, need to provide many ready-made patterns in order to be of much use to application developers. If generative design patterns are difficult to create, few patterns will exist. Because of this, the primary goal of this research was to simplify the pattern creation process.

The system created through this research reduces the level of expertise required to create patterns. Where previously, a pattern designer was often required to implement new types in CO_2P_2S, now most patterns can be built using the existing types. Pattern designers no longer need the skillset of the type designer. The task of the type designer has also been simplified. A well defined structure has been provided in which type designers can work. Tasks that were previously the responsibility of the type designer are now handled by the system. The type designer need only handle the core functionality of the new type being created.

The original system contained a number of limitations and complexities. In removing these, the pattern creation process was simplified. As an example, some of the tags in the original system were used merely to overcome the limitiations of JavaDoc as a code generator.

Using object-oriented types lowers the learning curve for developers using the system. Since they will already be familiar with Java, the types in the new system will be familiar. The new types are smaller and more easily composable than the previous types. This makes individual types easier to understand and simplifies the job of the type designer.

The new system is easier to learn than the previous system. The documentation for each type is easily accessed from within the CO_2P_2S environment; there is no need to consult external documents. Previously, pattern designers needed to know the CO_2P_2S type API, the CO_2P_2S macro language, the JavaDoc processor tags, the use of MetaCO_2P_2S, and a number of directory locations. As part of this research, MetaCO_2P_2S and the directory structure was simplified. Instead of creating types and using the adhoc constructs for code generation, now pattern designers use the more flexible CGL to glue together some of the many available types.

When patterns are easier to create, the pattern designer has more time to focus on design issues without getting lost in the implementation details. Instead of dealing with the limitations of the system, the pattern designer is freed to work on the code for the design pattern itself. As well as easing the pattern creation process, one of the goals was to reduce the possibility of error. Though there are no explicit measures in place to catch errors, the possibility of errors has been reduced through the overall improvement to the system. As an example, where previously users had to view the contents of multiple files and Java source to see the code to be generated, now the generated code for a class can be determined primarily from the contents of a single file. The use of small and well-defined types also reduces the chance of errors within the system. Instead of building a single type with a great number of conditions, smaller types with few conditions can be combined while retaining their individual correctness. The amount of "copy and paste coding" has been reduced in the new system. Techniques such as loops, macros, data structures, and arithmetic all serve to reduce the redundancy present in the pattern's code. Since less code needs to be written, the chance of making a mistake is decreased.

6.3 Limitations and Future Work

Because the code generator is a compositional system, it has little ability to manipulate pieces of code from a higher level. Transformational elements could lead to better separation of concerns and cleaner code. In addition, the use a tranformational layer within the system could open the door for source-to-source optimizations and additional error checking not possible with a purely compositional system.

The new system provides a good foundation for inter-option checking, but the implementation is not complete. The ability to enable options based on the contents of other options is desirable. Additionally, validation of option values based on other options is also useful. MetaCO_2P_2S would need to be extended to allow the pattern designer to specify the relationships between different options.

Some of the types within the system are not intended for use on their own. As an example, the MethodGuard does not provide a full GUI. A way to prevent the direct use of these types would be desirable. Alternatively, a more advanced method of combining GUIs could be used to reduce the number of types that are not directly usable in pattern creation. Types used only during code generation could also be excluded from the list of types available to the pattern designer in MetaCO_2P_2S.

There are some types missing which need to be added to the system. An `Enumeration` type, which would allow the user to select one of many options, is the most important of these. Another useful type is the `CompassDependencies` type, allowing a user to specify dependencies for grid computations. Though these types are simple to create, time did not permit their present existence. Some pieces of the GUI models were left undeveloped. For example, the support for numeric fields is lacking in the current implementation.

Though there are limitations and areas for future improvement, the present system may be used to create flexible generative design patterns with little interference from the system itself.

6.4 Conclusion

This research has produced a system that simplifies the creation of generative design patterns. The system of types in combination with the code generation language provide a simple and flexible platform upon which to create generative patterns. Though not well tested, we expect the contributions of this research to make the pattern creation process easier, faster, and more reliable.

Bibliography

[1] Angie. http://angie.d-s-t-g.com/.

[2] Cops web site. http://www.cs.ualberta.ca/ systems/cops/.

[3] Jalopy (java source code formatter). http://jalopy.sourceforge.net/.

[4] Javadoc. http://java.sun.com/j2se/javadoc/.

[5] Perl. http://www.perl.org/.

[6] Python. http://www.python.org/.

[7] Python grammar. http://www.python.org/doc/current/ref/grammar.txt.

[8] Python method resolution order. http://www.python.org/2.3/mro.html.

[9] John Anvik. Asserting the utility of CO_2P_3S using the cowichan problems. Master's thesis, Department of Computing Science, University of Alberta, 2002.

[10] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reuseable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[11] Peter Bosch. Inheritance vs. delegation: Is one better than the other? http://www.python.org/ftp/python/doc/delegation.ps.

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

[13] Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[14] F. Castor and P. Borba. A language for specifying java transformations. In *V Brazilian Symposium on Programming Languages*, pages 236–251, May 2001.

[15] J. Cordy, C. Halpern, and E. Promislow. Txl: A rapid prototyping system for programming language dialects. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 280–285, 1988.

[16] J. Cordy and M. Shukla. Practical metaprogramming. In *Proceedings of the 1992 IBM Centre for Advanced Studies Conference*, pages 215–224, November 1992.

[17] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*, chapter 9. Addison Wesley, 2000.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1995.

[19] Zhuang Guo. Developing network server applications using design pattern templates. Master's thesis, Department of Computing Science, University of Alberta, 2003.

[20] J. Lindskov Knudsen. Name collision in multiple classification hierarchies. In *ECOOP (European Conference on Object-Oriented Programming)*, pages 93–109. Springer-Verlag, 1988.

[21] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 214–223, New York, NY, 1986. ACM Press.

[22] Steve MacDonald, Duane Szafron, Jonathan Schaeffer, John Anvik, Steve Bromling, and Kai Tan. Generative design patterns. In *17th IEEE International Conference on Automated Software Engineering (ASE)*, pages 23–34, Edinburgh, UK, September 2002.

[23] Marek Majkut. Syntactic unit trees for the implementation of software product lines. In *Lecture Notes in Computer Science. Volume 2323. Object-Oriented Technology. ECOOP 2001 Workshop Reader*, pages 135–149, 2001.

[24] Kevin Østerbye. Refill – a generative java dialect. In *Lecture Notes in Computer Science. Volume 2548. Object-Oriented Technology. ECOOP 2002 Workshop Reader*, pages 15–29, 2002.

[25] Vojislav D. Radonjic. A generative approach to expressing and using object-oriented design patterns. In *Lecture Notes in Computer Science. Volume 2323. Object-Oriented Technology. ECOOP 2001 Workshop Reader*, pages 135–149, 2001.

[26] Richard J. Rodger. Jostraca: a template engine for generative programming. In *Lecture Notes in Computer Science. Volume 2548. Object-Oriented Technology. ECOOP 2002 Workshop Reader*, pages 15–29, 2002.

[27] Lynn Andrea Stein. Delegation is inheritance. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 138–146. ACM Press, 1987.

[28] John Viega, Bill Tutt, and Reimer Behrends. Automated delegation is a viable alternative to multiple inheritance in class based languages. Technical Report CS-98-03, 2, 1998. ftp://ftp.cs.virginia.edu/pub/techreports/CS-98-03.ps.Z.

[29] Kris De Volder. Generative logic meta programming. In *Lecture Notes in Computer Science. Volume 2323. Object-Oriented Technology. ECOOP 2001 Workshop Reader*, pages 135–149, 2001.

Appendix A

Original Decorator Pattern

This Appendix shows the code created by the pattern designer to implement the Decorator pattern in the original CO_2P_2S system.

A.1 Decorator Class

In the original CO_2P_2S system, the pattern designer needed to create a Java source file for each of the classes in their framework. Note that almost no code generation is done inside the source file itself. Looking at the class source provides no idea of what the decorator code might look like. What follows are the contents of the `FrameworkCLASS_Decorator.java` file. Note that the `@extParam` and `@frameworkSuperclass` tags do most of the work by delegating to the Java code.

```
/*****************************************************************************
 * This class represents the abstract superclass of a concrete Decorator
 * from the Decorator pattern.
 *
 * @userImports
 * @userCodeAllowed
 * @extParameter methods
 * @frameworkSuperclass FrameworkCLASS_DecoratorSuperclass
 */

public abstract class FrameworkCLASS_Decorator {

  /**
   * The decorated component.
   */
  private FrameworkCLASS_ComponentClass component;

  /**
   * Constructor that sets the decorated component to the given component.
   */
  public FrameworkCLASS_Decorator(FrameworkCLASS_ComponentClass c){

  }

  /**
   * Returns the decorated component.
   */
  public FrameworkCLASS_ComponentClass getDecoratedComponent(){

  }

  /**
   * Sets the decorated component to the given component.
   */
  public void setDecoratedComponent(FrameworkCLASS_ComponentClass c){

  }

  // Insert user methods here.
}
```

A.2 Decorator Methods

Instead of having normal method bodies, the old system required that the method bodies be placed in files of their own. The following files were created to fill the method bodies.

A.2.1 FrameworkCLASS_Decorator.FrameworkCLASS_ComponentClass

```
this.component = c;
```

A.2.2 getDecoratedComponent

```
return this.component;
```

A.2.3 setDecoratedComponent.FrameworkCLASS_ComponentClass

```
this.component = c;
```

A.3 Decorator Method List Element

The original CO_2P_2S system provided a MethodList class which allowed the user to work with a list of methods. The type of the methods in the list was specified by entering the name of a custom class in MetaCO_2P_2S. What follows is the custom class that was created for the Decorator pattern. Note that this file contains GUI display and code generation source. To display and generate the code for the list of methods, the method list system would make calls to certain methods in this file. What follows are the contents of the `DecoratorMethodListElement.java` file.

```
package cops.gui.patterns.decorator;

import cops.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.util.*;

public class DecoratorMethodListElement extends MethodListElement
{
  // Constants.
  private static final String DELEGATOR = "delegator";
  private static final String PREFIX = "prefix";
  private static final String SUFFIX = "suffix";
  private static final String GUARD = "guard";
  private static final String UNSET = "unset";
  private static final String METHOD_NAME_ERR =
    "You must enter valid method names for prefix, suffix and guard methods.";

  // Instance variables.
  private DecoratorEditPane editPane;

  /**
   * Class Constructor.
   *
   * @param id the id of this parameter
   * @param name the visual name of this parameter
```

```
     * @param menuText the menu text for setting this parameter
     */
    public DecoratorMethodListElement(String id, String name, String menu)
    {
      super(id, name, menu);

      editPane =
        new DecoratorEditPane(true, UNSET, true, UNSET, true, UNSET, true);
    }


    /**
     * Gets the body of the method mapped to the specified key for the code
     * generator, as it should appear in the specified class.
     *
     * @param className the name of the class to retrieve the method body for.
     * @param key the key mapped to the method to retrieve a body for.
     * @return the method body text to insert.
     */
    public String getCodegenBodyByKey(String className, String key)
    {
      boolean returnsVoid = getReturnType().equals("void");

      if (!className.equals("FrameworkCLASS_Decorator"))
        return "";

      if (key.startsWith(PREFIX))
      {
        String argName = key.substring(PREFIX.length());

        if (argName.equals(""))
          return "";

        else
          return "return " + argName + ";";
      }

      else if (key.equals(SUFFIX))
      {
        if (returnsVoid)
          return "";
        else
          return "return returnValue;";
      }

      else if (key.equals(GUARD))
        return "return true;";

      else if (key.equals(DELEGATOR))
      {
        CopsMethod suffix = getUtilityMethod(SUFFIX);
        CopsMethod guard = getUtilityMethod(GUARD);
        List args = getArguments();
        String returnType = getReturnType();
        StringWriter sw = new StringWriter();
        PrintWriter pw = new PrintWriter(sw);
        StringBuffer argBuf = new StringBuffer();
        StringBuffer newArgBuf = new StringBuffer();
        String argText = null;
        String newArgText = null;

        Iterator argsIt = args.iterator();
        while (argsIt.hasNext())
        {
          String argName = ((CopsArgument) argsIt.next()).getName();
    argBuf.append(argName);
    newArgBuf.append("_").append(argName).append("_");

    if (argsIt.hasNext())
    {
      argBuf.append(", ");
      newArgBuf.append(", ");
    }
  }

  argText = argBuf.toString();
  newArgText = editPane.hasPrefix() ? newArgBuf.toString() : argText;

  if (!returnsVoid)
  {
    pw.print(new CopsArgument(returnType, getArrayDimension(),
       "returnValue"));
    // for (int i = 0; i < getArrayDimension(); i++)
    //   pw.print("[]");

    pw.print(" = ");

    if (!CopsUtility.isPrimitiveType(returnType) ||
        (getArrayDimension() > 0))
      pw.print("null");

    else if (returnType.equals("boolean"))
      pw.print("false");

    else
      pw.print("0");

    pw.println(";");
    pw.println();
  }

  if (editPane.hasPrefix())
  {
    String prefixName = editPane.getPrefixName();

    // If the method has arguments, make calls to each argument's prefix.
    if (args.size() > 0)
    {
      argsIt = args.iterator();
      while (argsIt.hasNext())
      {
        CopsArgument arg = (CopsArgument) argsIt.next();
        String argName = arg.getName();
        CopsArgument newArg = new CopsArgument(arg);
        newArg.setName("_" + argName + "_");
        pw.println(newArg.toString() + " = " + prefixName + "_" + argName +
            "(" + argText + ");");
      }
    }

    // Otherwise call the single generic prefix.
    else
      pw.println(prefixName + "(" + argText + ");");

    pw.println();
  }

  if (guard != null)
    pw.println("if (" + guard.getName() + "(" + newArgText + "))");
```

```
      if (!returnsVoid)
      {
        pw.println("returnValue = this.component." + getName() + "(" +
                   newArgText + ");");
      }

      else
        pw.println("this.component." + getName() + "(" + newArgText + ");");

      pw.println();

      if (suffix != null)
      {
        if (!returnsVoid)
        {
          pw.print("returnValue = " + suffix.getName() + "(" + newArgText);

          if (newArgText.length() > 0)
            pw.print(", ");

          pw.print("returnValue);");
        }

        else
          pw.print(suffix.getName() + "(" + newArgText + ");");
      }

      if (!returnsVoid)
        pw.println("return returnValue;");

      try
      {
        pw.close();
        sw.close();
        return sw.toString();
      }

      catch (IOException exception)
      {
        return "";
      }
    }

    else
      return "";
  }


  /**
   * Gets the comment for the method mapped to the specified key for the code
   * generator, as it should appear in the specified class.
   *
   * @param className the name of the class to retrieve the method body for.
   * @param key the key mapped to the method to retrieve a body for.
   * @return the method body text to insert.
   */
  public String getCodegenCommentByKey(String className, String key)
  {
    if (!className.equals("FrameworkCLASS_Decorator"))
      return "";

    String prototype = getBasicMethod().toString();

    if (key.startsWith(PREFIX))
    {
```

```
    String argName = key.substring(PREFIX.length());

    if (argName.equals(""))
    {
      return "Called by: " + prototype +
      "\nto do any necessary computations before delegation occurs.";
    }

    else
    {
      return "Called by: " + prototype +
      "\nto precompute the value of " + argName + " passed to the " +
      "decorated component.\nThis method may have side effects or do any" +
      " other computations required\nbefore delegation occurs. The " +
      "original value is returned by default.";
    }
  }

  else if (key.equals(SUFFIX))
  {
    boolean returnsVoid = getReturnType().equals("void");
    return "Called by: " + prototype +
    "\nto do any necessary computations after delegation occurs" +
    (returnsVoid ? ". Does\nnothing by default." : ", as well as\n" +
    "generate a return value. The value returned by the decorated component"
    + "\nis returned by default.");
  }

  else if (key.equals(GUARD))
  {
    return "Called by: " + prototype +
    "\nto determine if delegation should occur. Returns true if delegation" +
    "\nshould occur and false otherwise. Returns true by default.";
  }

  else if (key.equals(DELEGATOR))
  {
    return "Delegates work to the method with the same prototype of the" +
    "\n decorated component.";
  }

  return "";
}


/**
 * Returns whether the method mapped to the provided key should appear
 * in the provided class.
 *
 * @return whether or not the method should appear in the class.
 */
public boolean appearsInClass(String className, String key)
{
  return editPane.getDelegate();
}


/**
 * Subclasses supply a String that will be used to refer to the basic method.
 *
 * @return a String mapped to the basic method.
 */
public String getBasicMethodKey()
{
  return DELEGATOR;
```

```
}


/**
 * Objects of a subclass can save any other fields they declare so that
 * they may be restored later.
 *
 * @return subclass fields encoded as a <code>String</code>.
 */
public String saveFieldsToString()
{
  StringBuffer buf = new StringBuffer();
  buf.append(editPane.getDelegate());
  buf.append(",").append(editPane.hasPrefix());
  buf.append(",").append(editPane.hasSuffix());
  buf.append(",").append(editPane.hasGuard());
  buf.append(",").append(editPane.getPrefixName());
  buf.append(",").append(editPane.getSuffixName());
  buf.append(",").append(editPane.getGuardName());
  return buf.toString();
}


/**
 * Objects of a subclass can restore their fields that have been saved as
 * a String.
 *
 * @param source the <code>String</code> to read from.
 */
public void loadFieldsFromString(String source)
{
  StringTokenizer tokenizer = new StringTokenizer(source, ",");
  editPane.setDelegate((new Boolean(tokenizer.nextToken())).booleanValue());
  editPane.setHasPrefix((new Boolean(tokenizer.nextToken())).booleanValue());
  editPane.setHasSuffix((new Boolean(tokenizer.nextToken())).booleanValue());
  editPane.setHasGuard((new Boolean(tokenizer.nextToken())).booleanValue());
  editPane.setPrefixName(tokenizer.nextToken());
  editPane.setSuffixName(tokenizer.nextToken());
  editPane.setGuardName(tokenizer.nextToken());
  validateEditPane();
}


/**
 * Returns a pane that will be added to a <code>MethodListElementDlg</code>
 * for editing this DecoratorMethodListElement.
 *
 * @return a component with gui widgets.
 */
public JComponent getEditPane()
{
  return editPane;
}


/**
 * Tells this MethodListElement to update itself according to the user
 * input from the edit pane.
 *
 * @return <code>null</code> if the settings in the pane are valid, or
 *         an error message otherwise.
 */
public String validateEditPane()
{
  String returnType = getReturnType();
```

```
clearUtilityMethods();

if (editPane.getDelegate())
{
  if (editPane.hasPrefix())
  {
    Iterator args = getArguments().iterator();
    String prefixName = editPane.getPrefixName();
    if (!CopsUtility.isValidMethodName(prefixName))
      return METHOD_NAME_ERR;

    // If the method has arguments, generate one prefix method for each.
    if (args.hasNext())
    {
      while (args.hasNext())
      {
        CopsArgument arg = (CopsArgument) args.next();
        CopsMethod prefix = new CopsMethod(getBasicMethod());
        prefix.setName(prefixName + "_" + arg.getName());
        prefix.setType(arg.getType());
        prefix.setArrayDimension(arg.getArrayDimension());
        putUtilityMethod(PREFIX + arg.getName(), prefix);
      }
    }

    // Otherwise generate one generic prefix method.
    else
    {
      CopsMethod prefix = new CopsMethod(getBasicMethod());
      prefix.setName(prefixName);
      prefix.setType("void");
      prefix.setArrayDimension(0);
      putUtilityMethod(PREFIX, prefix);
    }

  }

  if (editPane.hasSuffix())
  {
    String suffixName = editPane.getSuffixName();
    if (!CopsUtility.isValidMethodName(suffixName))
      return METHOD_NAME_ERR;

    CopsMethod suffix = new CopsMethod(getBasicMethod());
    suffix.setName(suffixName);

    // Add an extra argument if the basic method returns a value.
    if (!returnType.equals("void"))
      suffix.addArgument(returnType, getArrayDimension(), "returnValue");

    putUtilityMethod(SUFFIX, suffix);
  }

  if (editPane.hasGuard())
  {
    String guardName = editPane.getGuardName();
    if (!CopsUtility.isValidMethodName(guardName))
      return METHOD_NAME_ERR;

    CopsMethod guard = new CopsMethod(getBasicMethod());
    guard.setName(guardName);
    guard.setType("boolean");
    guard.setArrayDimension(0);
    putUtilityMethod(GUARD, guard);
  }
  }

  return null;
}


/**
 * Called after all of the data of an imported method is set to allow
 * this object to set its own fields accordingly.
 */
public void importComplete()
{
  String methodName = getName();
  editPane.setPrefixName(methodName + '_' + PREFIX);
  editPane.setSuffixName(methodName + '_' + SUFFIX);
  editPane.setGuardName(methodName + '_' + GUARD);
  validateEditPane();
}


/*****************************************************************************
 * This is the pane that DecoratorMethodListElements add to instances of
 * <code>MethodListElementDlg</code> for editing fields particular to the
 * former.
 */
private class DecoratorEditPane extends JPanel implements ActionListener
{
  // Constants.
  private static final String PREFIX = " Prefix Method ";
  private static final String SUFFIX = " Suffix Method ";
  private static final String GUARD = " Guard Method ";
  private static final String PREFIX_NAME = " Prefix Name ";
  private static final String SUFFIX_NAME = " Suffix Name ";
  private static final String GUARD_NAME = " Guard Name ";
  private static final String DELEGATE =
    " Delegate work to decorated component ";

  // Gui widgets.
  private JCheckBox hasPrefix, hasSuffix, hasGuard, delegate;
  private JTextField prefixName, suffixName, guardName;

  /**
   * Constructor.
   *
   * @param hasPrefix whether or not this method has a prefix method.
   * @param prefixName the name of the prefix method.
   * @param hasSuffix whether or not this method has a suffix method.
   * @param suffixName the name of the suffix method.
   * @param delegate whether or not this method should delegate work to
   *        the decorated component.
   */
  public DecoratorEditPane(boolean hasPrefix, String prefixName,
                           boolean hasSuffix, String suffixName,
                           boolean hasGuard, String guardName,
                           boolean delegate)
  {
    this.setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));

    this.delegate = new JCheckBox();
    this.delegate.setActionCommand(DELEGATE);
    this.delegate.addActionListener(this);
    this.delegate.setSelected(delegate);

    this.hasPrefix = new JCheckBox();
    this.hasPrefix.setActionCommand(PREFIX);
    this.hasPrefix.addActionListener(this);
    this.hasPrefix.setSelected(hasPrefix);

    this.hasSuffix = new JCheckBox();
    this.hasSuffix.setActionCommand(SUFFIX);
    this.hasSuffix.addActionListener(this);
    this.hasSuffix.setSelected(hasSuffix);

    this.hasGuard = new JCheckBox();
    this.hasGuard.setActionCommand(GUARD);
    this.hasGuard.addActionListener(this);
    this.hasGuard.setSelected(hasGuard);

    this.prefixName = new JTextField(prefixName);
    this.suffixName = new JTextField(suffixName);
    this.guardName = new JTextField(guardName);

    Box delegateBox = Box.createHorizontalBox();
    delegateBox.add(new JLabel(DELEGATE));
    delegateBox.add(this.delegate);
    this.add(delegateBox);

    Box prefixBox = Box.createHorizontalBox();
    prefixBox.add(new JLabel(PREFIX));
    prefixBox.add(this.hasPrefix);
    prefixBox.add(new JLabel(PREFIX_NAME));
    prefixBox.add(this.prefixName);
    this.add(prefixBox);

    Box suffixBox = Box.createHorizontalBox();
    suffixBox.add(new JLabel(SUFFIX));
    suffixBox.add(this.hasSuffix);
    suffixBox.add(new JLabel(SUFFIX_NAME));
    suffixBox.add(this.suffixName);
    this.add(suffixBox);

    Box guardBox = Box.createHorizontalBox();
    guardBox.add(new JLabel(GUARD));
    guardBox.add(this.hasGuard);
    guardBox.add(new JLabel(GUARD_NAME));
    guardBox.add(this.guardName);
    this.add(guardBox);

    enableComponents();
  }


  /**
   * Sets wether or not this method has a prefix method.
   *
   * @param hasPrefix whether or not this method has a prefix.
   */
  public void setHasPrefix(boolean hasPrefix)
  {
    this.hasPrefix.setSelected(hasPrefix);
    enableComponents();
  }


  /**
   * Returns whether or not this method has a prefix method.
   *
   * @return whether or not this method has a prefix method.
   */
  public boolean hasPrefix()
```

```
{
  return hasPrefix.isSelected();
}


/**
 * Returns the name of the prefix method.
 *
 * @return the name of the prefix method.
 */
public String getPrefixName()
{
  return prefixName.getText().trim();
}


/**
 * Sets the name of the prefix method.
 *
 * @param name the new name of the prefix method.
 */
public void setPrefixName(String name)
{
  prefixName.setText(name);
}


/**
 * Sets whether or not this method has a suffix method.
 *
 * @param hasSuffix whether or not this method has a suffix method.
 */
public void setHasSuffix(boolean hasSuffix)
{
  this.hasSuffix.setSelected(hasSuffix);
  enableComponents();
}


/**
 * Returns whether or not this method has a suffix method.
 *
 * @return whether or not this method has a suffix method.
 */
public boolean hasSuffix()
{
  return hasSuffix.isSelected();
}


/**
 * Returns the name of the suffix method.
 *
 * @return the name of the suffix method.
 */
public String getSuffixName()
{
  return suffixName.getText().trim();
}


/**
 * Sets the name of the suffix method.
 *
 * @param name the new name of the suffix method.
 */
public void setSuffixName(String name)
{
  suffixName.setText(name);
}


/**
 * Sets whether or not this method has a guard method.
 *
 * @param hasGuard whether or not this method has a guard method.
 */
public void setHasGuard(boolean hasGuard)
{
  this.hasGuard.setSelected(hasGuard);
  enableComponents();
}


/**
 * Returns whether or not this method has a guard method.
 *
 * @return whether or not this method has a guard method.
 */
public boolean hasGuard()
{
  return hasGuard.isSelected();
}


/**
 * Returns the name of the guard method.
 *
 * @return the name of the guard method.
 */
public String getGuardName()
{
  return guardName.getText().trim();
}


/**
 * Sets the name of the guard method.
 *
 * @param name the new name of the guard method.
 */
public void setGuardName(String name)
{
  guardName.setText(name);
}


/**
 * Returns whether or not this method delegates work to the decorated
 * component.
 *
 * @return whether or not work is delegated.
 */
public boolean getDelegate()
{
  return delegate.isSelected();
}


/**
```

```
     * Sets whether or not this method delegates work to the decorated
     * compoenent.
     *
     * @param delegate wether or not work is delegated.
     */
    public void setDelegate(boolean delegate)
    {
      this.delegate.setSelected(delegate);
      enableComponents();
    }


    /**
     * Callback for the check boxes.
     *
     * @param event  generated by the check boxes when their state changes.
     */
    public void actionPerformed(ActionEvent event)
    {
      enableComponents();
    }


    /**
     * Enables or disables the gui widgets according to the state of the
     * check boxes.
     */
    private void enableComponents()
    {
      if (delegate.isSelected())
      {
        hasPrefix.setEnabled(true);
        hasSuffix.setEnabled(true);
        hasGuard.setEnabled(true);

        prefixName.setEnabled(hasPrefix.isSelected());
        suffixName.setEnabled(hasSuffix.isSelected());
        guardName.setEnabled(hasGuard.isSelected());
      }

      else
      {
        hasPrefix.setEnabled(false);
        hasSuffix.setEnabled(false);
        hasGuard.setEnabled(false);
        prefixName.setEnabled(false);
        suffixName.setEnabled(false);
        guardName.setEnabled(false);
      }
    }
  }
}
```

Appendix B

CGL Decorator Pattern

This Appendix shows the code created by the pattern designer to implement the Decorator pattern in CGL.

B.1 Decorator Class

```
#output decorator
#template

#user "User Imports"

/*****************************************************************************
 * This class represents the abstract superclass of a concrete Decorator
 * from the Decorator pattern.
 */

public abstract class @decorator @inherits(decoratorSuperclass) {
    /**
     * The decorated component.
     */
    private @componentClass component;

    /**
     * Constructor that sets the decorated component to the given component.
     */
    public @decorator (@componentClass c){
        this.component = c;
    }

    /**
     * Returns the decorated component.
     */
    public @componentClass getDecoratedComponent(){
        return this.component;
    }

    /**
     * Sets the decorated component to the given component.
     */
    public void setDecoratedComponent(@componentClass c){
        this.component = c;
    }

    #user "User Methods"
    /*
     * Methods Generated From Delegated Method List
     */

    #for method in methods

    #rem  This next line also sets an internal variable that is used to pass
    #rem  the appropriate result variable to the method suffix.

    #assign resVar method.resultVariable("result")

    #rem  If there is no suffix, our code requires no result variable.  We set
    #rem  the type to "void" to avoid its creation.  These six lines of code
    #rem  merely simplify the resulting code without changing its correctness.

    #if not method.suffixRequired
      #assign resVar.type "void"
      #assign handleResult "return"
    #else
      #assign handleResult resVar.assign()
    #end

    /**
     * Delegation Method.  This method is responsible for performing the actual
     * decoration.  The delegated method is called according the the results
     * from the optional prefix and guard methods.  The optional suffix method
     * can modify the return value of this method, if there is one.
     */
    @method {
        @resVar.declareWithDefault()

        @method.prefix()

        @method.guardStart()
            @handleResult this.component.@method.call();
        @method.guardEnd()

        @method.suffix()

        @resVar.optionalReturn()
    }

    @method.prefixMethod()

    @method.guardMethod()

    @method.suffixMethod()

    #end
}
```

Appendix C

GUI Skeleton Java Source

C.1 Display GUI Skeleton Java Source

```
/* This class is used inside COPS to display an option of this type.  The
   option will be displayed in a dialog box, however you only need to worry
   about filling a panel.  DataCopsGui eventually extends from JPanel and
   that is the panel you can fill with widgets.

   Note that the Data* classes provide many models that can immediately be
   combined with the Java GUI widgets.  Often you will be able to create the
   GUI inside of initializePanel() and then forget about it.  If it is tricky
   to associate one of the existing models with a widget, you can use the
   dumpToModel() method to do the data transfer from the GUI.  Note that
   dumpToModel() will always be called, even if the dialog box is cancelled.
   It can be used to do GUI cleanup if needed.

   In addition to the regular data access methods that are part of the Data*
   types, the following methods may be used:
       model.getParameters()
           If you set parameters for a type within the MetaCOPS code, you will
           need to retrieve and use them here.  This method returns a
           DataRecord that contains the parameter fields for the particular
           type being accessed.  Note that the parameters are shared between
           all list elements within a DataList.
       model.getPanel()
           Get the JPanel connected with the type.  This will call
           the appropriate initializePanel() and dumpToModel() methods
           automatically.
       model.toString()
           Get the string representation of the data item.
       getSuperPanel(String type)
           If a type has an ancestor, you may grab the GUI panel of the
           ancestor by specifying the name of the ancestor.  If the ancestor
           panel cannot be found, a panel with an error message will be
           returned.  Notice that the method is not part of the model,
          it is called directly.
       getSuperString(String type)
           If a type has an ancestor, you may grab its string representation
           by calling this method.  If the ancestor cannot be found, an error
           string will be returned.

   The values in the parameters DataRecord structure are those that have been
   set by the pattern designer in MetaCOPS.  As an example, suppose there was
   a numeric type that allowed the pattern designer to set the maximum and
   minimum values that the COPS user could enter.  Those parameters would
   be set in MetaCOPS and this class would be responsible for using the
   parameters to limit the user's choice in some manner.
   By default the JPanel has a vertical BoxLayout.  If you wish, you can
   simply add() a widget to it and then call addSpacer() to leave a small
   gap before you add your next widget.
       void addSpacer()

   Note that the add(Component) method is overriden.  The overriden version
   sets the maximum height of the component to the preferred height.
       Component add(Component c)

   There is also the addLabel(String) utility method that may be used to add
   an informative text box label.  It will also add a spacer below itself.
       void addLabel(String message)

   IMPORTANT:
     If the COPS system is unable to instantiate this class, it will not appear
     in the type list in MetaCOPS, nor will it be usable from COPS.  The
     initializePanel(Data model) method must be specified in order to get the
     type to appear in the type list.
*/

package cops.types.TYPENAME.gui;

import cops.typesystem.data.*;

public class TYPENAME_cops extends DataCopsGui {
    /* If you need access to any part of the GUI after it's created, you will
       want to create private instance variables to store that state. */


    /* This method is responsible for initializing the model of this type.
       If anything in the model should have a non-default (zero, false, "")
       value, it should be set here.  If this method is not provided,
       the default values will be used. */

    // public void initializeModel(Data model) {
    // }

    /* This method is responsible for creating the GUI and adding it to the
       panel represented by 'this'.  The GUI's values should be initialized
       according to the values stored in the model. */

    // public void initializePanel(Data model) {
    // }

    /* This method is responsible for collecting data from the GUI and placing
       it in the model that is passed in.  If you discover an error at this
       point (for example, an alphabetic string was entered for an integer
       value), simply return a String indicating what has gone wrong.  If no
       error occured, return null.  This method will be called whether the user
       accepts or rejects the current changes.  It can be used to perform
       operations like stoping edits on tables. */

    // public String dumpToModel(Data model) {
    //     return null;
    // }

    /* Before storing, the model is validated to ensure that it contains no
       faulty data.  This will be called when the user tries to close the
       dialog showing the option.  If validation does not succeed, the dialog
       will persist.  If an error is detected, return a String describing the
       error.  If no error is found, simply return null.  Our validation
       method will be called before the validation methods of our ancestors
       and our children. */
    // public String validateModel(Data model) {
    //     return null;
    // }

    /* COPS may desire to display the data for this type as a string.  By
       providing this method, you can control the string that is generated. */

    // public String toString(Data model) {
    //     return "";
    // }
}
```

C.2 Parameters GUI Skeleton Java Source

```
/* This class is used inside MetaCOPS to display the parameters for options of
   this type.  DataMetaGui eventually extends from JPanel and that is the panel
   you must fill with widgets.

   Note that the Data* classes provide many models that can immediately be
   combined with the Java GUI widgets.  Often you will be able to create the
   GUI inside of initializePanel() and then forget about it.  If it is tricky
   to associate one of the existing models with a widget, you can use the
   dumpToModel() method to do the data transfer from the GUI.  Note that
   dumpToModel() will always be called, even if the dialog box is cancelled.
   It can be used to do GUI cleanup if needed.

   The tree passed in corresponds to the structure of the type specified in
   MetaCOPS.  For example, take the following structure:

        Record (TypeA)
          +fieldB (TypeB)
          +fieldC (List)
             +Boolean

   In this example, the user-created types are TypeA and TypeB.  If we wish
   to access the parameters of the TypeA record, we simply take the DataTree
   that was passed in and call getParameters() on it.
        tree.getParameters()
   If we wanted to access the parameters of TypeB, we would go:
        tree.getField("fieldB").getParameters()
   If we wanted to access the parameters of the Boolean value, we would go:
        tree.getField("fieldC").getElement().getParameters()

   In addition to the methods used to access the tree, you can also use the
   following methods:
       tree.getPanel()
           Gets the JPanel connected with the type.  This will call
           the appropriate initializePanel() and dumpToModel() methods
           automatically.
       getSuperPanel(String type)
           If a type has an ancestor, you may grab the GUI panel of the
           ancestor by specifying the name of the ancestor.  If the ancestor
           panel cannot be found, a panel with an error message will be
           returned.  Notice that the method is not part of the model,
           it is called directly.

   The DataTree tree structure mirrors the main structure of the type.  The
   DataRecord returned by getParameters() has its contents defined by the
   Parameters structure defined in the type editor.

   By default the JPanel has a vertical BoxLayout.  If you wish, you can
   simply add() a widget to it and then call addSpacer() to leave a small
   gap before you add your next widget.
```

```
        void addSpacer()

    Note that the add(Component) method is overriden.  The overriden version
    sets the maximum height of the component to the preferred height.
        Component add(Component c)

    There is also the addLabel(String) utility method that may be used to add
    an informative text box label.  It will also add a spacer below itself.
        void addLabel(String message)

    If you're using the default method for creating your panel, you will likely
    want to call addFiller() when you're done adding widgets to the panel.  This
    will fill any space that remains.
        void addFiller()
*/

package cops.types.TYPENAME.gui;

import cops.typesystem.data.*;

public class TYPENAME_metacops extends DataMetaGui {
    /* If you need access to any part of the GUI after it's created, you will
       want to create private instance variables to store that state. */


    /* This method is responsible for initializing the model of this type.
       If anything in the model should have a non-default (zero, false, "")
       value, it should be set here.  If this method is not provided,
       the default values will be used. */

    // public void initializeModel(DataTree tree) {
    // }

    /* This method is responsible for creating the GUI and adding it to the
       panel represented by 'this'.  The GUI's values should be initialized
       according to the values stored in the model. */

    // public void initializePanel(DataTree tree) {
    // }

    /* This method is responsible for collecting data from the GUI and placing
       it in the model that is passed in.  If you discover an error at this
       point (for example, an alphabetic string was entered for an integer
       value), simply return a String indicating what has gone wrong.  If no
       error occured, return null.  This method will be called whether the user
       accepts or rejects the current changes.  It can be used to perform
       operations like stoping edits on tables. */

    // public String dumpToModel(DataTree tree) {
    //     return null;
    // }

    /* Before storing, the model is validated to ensure that it contains no
       faulty data.  This will be called when the user tries to close the
       dialog showing the option.  If validation does not succeed, the dialog
       will persist.  If an error is detected, return a String describing the
       error.  If no error is found, simply return null.  Our validation
       method will be called before the validation methods of our ancestors
       and our children. */

    // public String validateModel(DataTree tree) {
    //     return null;
    // }
}
```

Appendix D

DTDs for XML Used in Type System

To promote the ability to separate the CO_2P_2S modules, XML has been used as the standard inter-module communication mechanism. The DTDs in this appendix show the structure of those XML files.

D.1 DTD for CO_2P_2S Types

Types in CO_2P_2S system are stored in the following format.

```
<!ELEMENT Type (Species,
                TypeName,
                Ancestors?,
                Fields?,
                ElementType?,
                Parameters?,
                Documentation?)>
<!ELEMENT Species (#PCDATA)>
<!ELEMENT TypeName (#PCDATA)>
<!ELEMENT Ancestors ((AncestorName)*)>
<!ELEMENT AncestorName (#PCDATA)>
<!ELEMENT Fields ((Field)*)>
<!ELEMENT Field (FieldName, (FieldReference|Type))>
<!ELEMENT FieldName (#PCDATA)>
<!ELEMENT FieldReference (#PCDATA)>
<!ELEMENT ElementType (ElementReference|Type)>
<!ELEMENT ElementReference (#PCDATA)>
<!ELEMENT Parameters (Type)>
<!ELEMENT Documentation (#PCDATA)>
```

D.2 DTD for Type Data

When the types are used, their instance data is stored in the formats indicated in this file. `DataTree` is used to store the tree structure of which the parameter records are a part.

```
<!ELEMENT CopsData:DataTree (CopsData:DataTreeParameters?,
                             CopsData:DataTreeRecord?,
                             CopsData:DataTreeList?)>
<!ATTLIST CopsData:DataTree type NMTOKEN #REQUIRED>
<!ELEMENT CopsData:DataTreeParameters (CopsData:Data)>
<!ELEMENT CopsData:DataTreeRecord (CopsData:DataTreeField*)>
<!ELEMENT CopsData:DataTreeField (CopsData:DataTreeKey, CopsData:DataTree)>
<!ELEMENT CopsData:DataTreeKey (#PCDATA)>
```

```
<!ELEMENT CopsData:DataTreeList (CopsData:DataTree)>

<!ELEMENT CopsData:Data (CopsData:Record|
                         CopsData:List|
                         CopsData:String|
                         CopsData:Float|
                         CopsData:Integer|
                         CopsData:Boolean)>
<!ATTLIST CopsData:Data type NMTOKEN #REQUIRED>
<!ELEMENT CopsData:Record (CopsData:Field*)>
<!ELEMENT CopsData:Field (CopsData:Key, CopsData:Data)>
<!ELEMENT CopsData:Key (#PCDATA)>
<!ELEMENT CopsData:List (CopsData:Data*)>
<!ELEMENT CopsData:String (#PCDATA)>
<!ELEMENT CopsData:Float (#PCDATA)>
<!ELEMENT CopsData:Integer (#PCDATA)>
<!ELEMENT CopsData:Boolean (#PCDATA)>
```

D.3 DTD for Pattern Definition

This DTD specifies the format of the XML files that define the user interface provided for each pattern in CO_2P_2S. Note the use of CopsData.

```
<!-- Include CopsData:* declarations -->
<!ENTITY % datadecls SYSTEM "CopsData.dtd">
%datadecls;

<!ELEMENT CopsPattern:patternInfo (CopsPattern:patternName,
                                   CopsPattern:constants,
                                   CopsPattern:options,
                                   CopsPattern:guiInfo)>
<!ATTLIST CopsPattern:patternInfo
      xmlns:CopsPattern CDATA #REQUIRED
>

<!ELEMENT CopsPattern:patternName (#PCDATA)>

<!ELEMENT CopsPattern:constants (CopsPattern:constant)*>
<!ELEMENT CopsPattern:constant (CopsPattern:constantID,
                                CopsPattern:constantValue)>
<!ELEMENT CopsPattern:constantID (#PCDATA)>
<!ELEMENT CopsPattern:constantValue (#PCDATA)>

<!ELEMENT CopsPattern:options (CopsPattern:option*)>
<!ELEMENT CopsPattern:option (CopsPattern:optionMenu,
                              CopsPattern:representsPatternName?,
                              CopsPattern:optionType,
                              CopsPattern:enabled?,
                              CopsPattern:parameters)>
<!ATTLIST CopsPattern:option optionIdentifier CDATA #REQUIRED>
<!ELEMENT CopsPattern:optionMenu (#PCDATA)>
<!ELEMENT CopsPattern:optionType (#PCDATA)>
<!ELEMENT CopsPattern:representsPatternName (#PCDATA)>
<!ELEMENT CopsPattern:enabled (#PCDATA)>
<!ELEMENT CopsPattern:parameters (CopsData:DataTree)>

<!ELEMENT CopsPattern:guiInfo (CopsPattern:visualElements)>
<!ELEMENT CopsPattern:visualElements (CopsPattern:tElement*,
                                      CopsPattern:gElement*)>
<!ELEMENT CopsPattern:gElement (CopsPattern:gElementID,
                                CopsPattern:gElementLocationX,
                                CopsPattern:gElementLocationY,
                                CopsPattern:gElementImages?,
                                    CopsPattern:gElementCurImageParts?)>
<!ELEMENT CopsPattern:tElement (CopsPattern:tElementID,
                                CopsPattern:tElementLocationX,
                                CopsPattern:tElementLocationY,
                                CopsPattern:tElementMaxLength,
                                CopsPattern:tElementJustification,
                                CopsPattern:tElementUpdateType,
                                CopsPattern:tElementText?,
                                CopsPattern:tElementUpdateVal?)>
<!ELEMENT CopsPattern:gElementID (#PCDATA)>
<!ELEMENT CopsPattern:gElementLocationX (#PCDATA)>
<!ELEMENT CopsPattern:gElementLocationY (#PCDATA)>
<!ELEMENT CopsPattern:gElementImages (CopsPattern:gElementImage)*>
<!ELEMENT CopsPattern:gElementImage (CopsPattern:gElementImageName,
                                     CopsPattern:gElementImageLoc)>
<!ELEMENT CopsPattern:gElementImageName (#PCDATA)>
<!ELEMENT CopsPattern:gElementImageLoc (#PCDATA)>
<!ELEMENT CopsPattern:gElementCurImageParts (CopsPattern:gElementCurImagePart*)>
<!ELEMENT CopsPattern:gElementCurImagePart (CopsPattern:gElementCurImagePartVal,
                                            CopsPattern:gElementCurImagePartType)>
<!ELEMENT CopsPattern:gElementCurImagePartVal (#PCDATA)>
<!ELEMENT CopsPattern:gElementCurImagePartType (#PCDATA)>
<!ELEMENT CopsPattern:tElementID (#PCDATA)>
<!ELEMENT CopsPattern:tElementLocationX (#PCDATA)>
<!ELEMENT CopsPattern:tElementLocationY (#PCDATA)>
<!ELEMENT CopsPattern:tElementMaxLength (#PCDATA)>
<!ELEMENT CopsPattern:tElementText (#PCDATA)>
<!ELEMENT CopsPattern:tElementJustification (#PCDATA)>
<!ELEMENT CopsPattern:tElementUpdateType (#PCDATA)>
<!ELEMENT CopsPattern:tElementUpdateVal (#PCDATA)>
```

Appendix E

Type System API

This appendix contains a description of the entire type system API. Because this section could be used as a stand-alone reference, some pieces have been included from Chapter 4.

The classes are divided into three categories. Data access classes are used to read and write the data contained within the type instances. Framework classes make up the inherited framework that the type designer works within. Finally, the GUI classes provide some useful widgets for the type designer to work with.

E.1 Data Access Classes

E.1.1 DataTree

The `DataTree` class is used within the Parameters GUI to access the parameters that are part of the type. This was explained in Section 4.2. The `getPanel()` method allows the type designer to use the Parameter GUIs provided by the children shown in Section 4.2.

```
class DataTree {
    public DataRecord getParameters();
    public DataTree getField(String key);
    public DataTree getElement();

    public JPanel getPanel();
}
```

E.1.2 Data

This is the superclass for the remainder of the data access classes. It provides common functionality.

```
class Data {
    // Access Methods
    public DataRecord getParameters();
    public JPanel getPanel();
    public boolean createDialog();
    public String toString();

    // Casting Methods
    public DataString asString();
    public DataBoolean asBoolean();
    public DataBoolean asInteger();
    public DataInteger asFloat();
    public DataList asList();
    public DataRecord asRecord();

    // Low Level Access Methods
    public OptionType getType();
    public DataTree getDataTree();
}
```

getParameters() Returns the parameters associated with this data item. If none are available, `null` is returned.

getPanel() Returns the `JPanel` for this data item. This method is used to access the GUIs of child elements. If no panel is available, `null` will be returned.

createDialog() Creates a dialog whose contents are the same as the panel returned by `getPanel()`. The dialog will have "OK" and "Cancel" buttons. If "Cancel" is selected, any changes to the model will be rolled back. Cancelling a higher level dialog will also cancel the changes made by dialogs created from inside the higher level dialog. `createDialog()` returns `true` if the dialog is "OKed", `false` otherwise.

toString() Returns the string representation of this data item.

Casting Methods These methods simply perform a cast to one of the indicated Data types.

getType() Returns the underlying `OptionType`. See Section E.2.4 for details.

getDataTree() Returns the underlying `DataTree` node, or `null` if no such node exists.

E.1.3 DataString

`DataString` is a container for a Java `String`. This class implements the `Document` interface, providing a convenient model for use in text widgets.

```
class DataString extends Data
  implements Document
{
    public String get();
    public void set(String s);
}
```

get() This method returns the underlying `String`. Each of the other types uses similar accessor methods, the only difference being the type of the value passed into or returned from the methods.

set() This method sets the underlying `String` to the passed in value.

E.1.4 DataBoolean

`DataBoolean` acts as a wrapper for a `boolean` Java primitive. It also provides the `ButtonModel` interface for easy manipulation using various button-like widgets. `JCheckBox` is able to take advantage of the ButtonModel interface.

```
class DataBoolean extends Data
  implements ButtonModel
{
    public boolean get();
    public void set(boolean b);
}
```

E.1.5 DataInteger

This class acts as a wrapper for a `long` Java primitive.

```
class DataInteger extends Data {
    public long get();
    public void set(long l);
}
```

E.1.6 DataFloat

This class acts as a wrapper for a `double` Java primitive.

```
class DataFloat extends Data {
    public double get();
    public void set(double d);
}
```

E.1.7 DataList

This class acts as a full featured list data type. It implements a number of Java interfaces for easy access to the underlying type. Operations such as sorting can be accomplished through the `List` interface. GUIs can be created based on `ListModel` or `ComboBoxModel`. Note that the DataList performs checks to ensure that only elements of the right type can be added. Elements of the right type can be created using the `create` factory methods shown below. Creating an element does not add it to the list; this must be performed as usual through the `List` interface methods.

```
class DataList extends Data
  implements List,
             RandomAccess,
             ListModel,
             ComboBoxModel
{
    // Factory Methods
    public Data create();
    public DataString createString();
    public DataBoolean createBoolean();
    public DataInteger createInteger();
    public DataFloat createFloat();
    public DataList createList();
    public DataRecord createRecord();

    // Casting Accessor Methods
    public Data getData(int index);
    public DataString getString(int index);
    public DataBoolean getBoolean(int index);
    public DataInteger getInteger(int index);
    public DataFloat getFloat(int index);
    public DataList getList(int index);
    public DataRecord getRecord(int index);

    // Internal Method
    public void fireContentsChanged(int index0, int index1);
}
```

create() This method is used to create a new `Data` item. The `DataList` knows the structure of its children and the new item will be created fully formed.

getData() Given an index, this method returns the `Data` item at the appropriate location in the list.

Casting Methods The remainder of the get and create methods merely cast the result of the previous methods into the different data types.

fireContentsChanged() Though normally this method would not be needed, the current list is unaware of when its child elements change. If a child element changes in a way that modifies the appearance of the list, this method must be called to update the associated GUI list models.

E.1.8 DataRecord

This class provides the `Map` interface for manipulating records. Note that the optional `clear()`, `put()`, `putAll()`, and `remove()` methods are not supported.

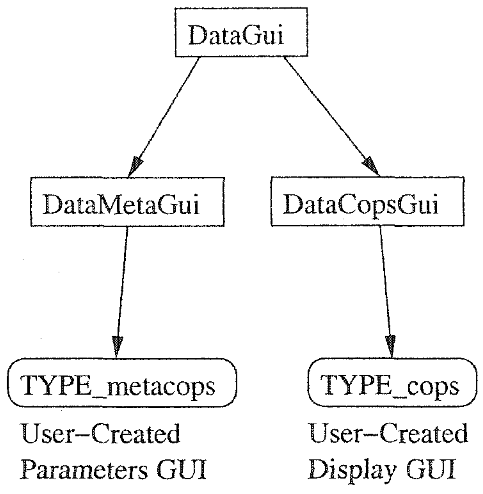


Figure E.1: Type Framework

```
class DataRecord extends Data
  implements Map
{
    public Data getData(String key);
    public DataString getString(String key);
    public DataBoolean getBoolean(String key);
    public DataInteger getInteger(String key);
    public DataFloat getFloat(String key);
    public DataList getList(String key);
    public DataRecord getRecord(String key);
}
```

getData() Given a key, this method returns the `Data` element associated with the given key. If the key is not in the record, `null` will be returned.

Casting Methods These methods simply cast the `getData()` result to the appropriate form.

E.2 Framework Classes

The types in this section are used in the creation of the display and parameters GUIs. Remember that the GUI source includes not only GUI creation, but also initialization and validation. The class hierarchy is shown in Figure E.1 with the user created types at the bottom.

E.2.1 DataGui

At the highest level, the `DataGui` class provides functionality that is common to both the parameter and display GUIs. This class is itself a `JPanel`. In the constructor, the JPanel is set to use a vertical box layout.

```
class DataGui extends JPanel {
    public Component add(Component c);
    void addSpacer();
    void addFiller();
    void addLabel(String text);
}
```

add() The add method is similar to `JPanel`'s `add()` method, only it sets the maximum height of the widget to be the preferred height of the widget. Normally widgets should be added using this method to avoid problems with display in MetaCO_2P_2S.

addSpacer() Adds a small blank area to the GUI below the last component that was added.

addFiller() Used in the parameters GUI, this method should be called to fill the empty space after all the other components have been added.

addLabel(String text) This can be used as a convenient way to add a stand-alone label to the GUI. It will produce the type of box-framed labels that can be seen in MetaCO_2P_2S.

E.2.2 DataMetaGui

This class is used as the superclass for all of the "Parameter GUI" classes. It provides default implementations of the four methods that need to be overridden.

```
class DataMetaGui extends DataGui {
    // Type designer should override some of these methods
    public void initializeModel(DataTree tree);
    public void initializePanel(DataTree tree);
    public String dumpToModel(DataTree tree);
    public String validateModel(DataTree tree);

    // This method provides access to ancestors.
    protected JPanel getSuperPanel(String ancestor);
}
```

initializeModel() This method is responsible for initializing the contents of the given tree of parameters. If it is not overridden, the model will be initialized with empty lists, zero numeric values, empty strings, and false booleans.

initializePanel() Responsible for creating the GUI panel. If this method is not overridden, the Parameters panel will not be provided in MetaCO_2P_2S. The contents of the panel must reflect the information in the provided tree.

dumpToModel() If the GUI does not automatically update the underlying models, this method can be used to store the values from the GUI into the model. If all is well, `null` should be returned. If an error occured, a string indicating the problem should be returned.

validateModel() This method is used to confirm that the contents of the tree are correct. If they are, `null` should be returned. If they are not, an error string should be returned.

getSuperPanel() Given the name of an ancestor type, this method will retrieve the GUI panel used to display and manipulate the ancestor's parameters.

E.2.3 DataCopsGui

This class is used as the superclass for all of the "Display GUI" classes. It provides default implementations of the five methods that need to be overridden.

```
class DataCopsGui extends DataGui {
    public void initializeModel(Data model);
    public void initializePanel(Data model);
    public String dumpToModel(Data model);
    public String validateModel(Data model);
    public String toString(Data model);

    protected JPanel getSuperPanel(String ancestor);
    protected String getSuperString(String ancestor);
}
```

initializeModel() This method is responsible for initializing the contents of the given model. If it is not overridden, the model will be initialized with empty lists, zero numeric values, empty strings, and false booleans.

initializePanel() Responsible for creating the GUI panel. If this method is not overridden, the type will not be available for selection in MetaCO_2P_2S. The contents of the panel must reflect the information in the provided model.

dumpToModel() If the GUI does not automatically update the underlying models, this method can be used to store the values from the GUI into the model. In CO_2P_2S, this method will be called when a dialog is accepted or even cancelled. If all is well, `null` should be returned. If an error occured, a string indicating the problem should be returned.

validateModel() This method is used to confirm that the contents of the model are correct. If they are, `null` should be returned. If they are not, an error string should be returned.

toString() Returns the string representation of the type. If `toString()` is not provided and the type is based on a String, Boolean, Integer, or Float value, the simple string representation of the value will be used. For Records and Lists, default strings will be automatically generated but they are not particularly helpful. If the type designer is creating a Record or List based type that might be displayed in a list or as part of a pattern GUI, the designer should override this method.

getSuperPanel() Given the name of an ancestor type, this method will retrieve the GUI panel used to display and manipulate the ancestor's structure.

getSuperString() Given the name of an ancestor type, this method will retieve the string generated by the ancestor type.

E.2.4 OptionType

Though this class should not need to be used by the type designer, it is mentioned due to its importance. `OptionType` is the Java class that contains the entire description of each type. If the type designer wished to create a complicated GUI that could be used for different types, the `OptionType` class would be used to access the internal description of the type. See the `OptionType` source[2] for full details on type internals.

E.3 GUI Classes

E.3.1 DataJTable

In combination with `DataJTableModel`, the `DataJTable` type provides a flexible way to display lists of editable items. It is based on the Java `JTable` class. Figure 3.7 includes two of these tables.

```
class DataJTable extends JPanel {
    public DataJTable(DataList list, String title);

    public DataJTable(DataList list);
    public void addField(String field, String title);

    public boolean stopEditing();
}
```

DataJTable(list, title) This constructor is used to created single column tables. It currently supports the display of lists of string values. A title should be provided for the table column.

DataJTable(list) This constructor is used to create a table based on a list with record children. Afer the table is created, the `addField()` should be called.

addField() Given the name of a field and a title for the table column, this method adds the record field as an editable column within the `JTable`. It currently only supports string fields.

stopEditing() This method should be called when the dialog that created the JTable is closed. It can be called from `dumpToModel()` methods. The method will stop any currently occuring edits in the table and commit the changes. If the entry could not be accepted, `stopEditing()` would return `false`. Presently, this method should always return `true`.

E.3.2 DataJList

The `DataJList` class provides an editable list of arbitrary elements. The only requirement is that the list elements provide a sane `toString()` implementation. The widget will created a titled frame if a non-empty and non-null title is provided. Figure 3.6 provides an example of the list.

```
class DataJList extends JPanel {
    public DataJList(DataList list, String title);
}
```

Appendix F

Generator Skeleton Python Source

```
# The code in this file will end up in a class, similar to this:
#
# class TYPENAME(ancestor1, ancestor2, ancestor3):
#     THE
#     CODE
#     FROM
#     THIS
#     FILE
#
# The self parameter must be specified as the first parameter for all methods
# in this file.  It represents the instance of the type and is used to access
# the fields and methods within the type.

# This method is used to initialize the state of the type.  By default, it
# passes and does nothing.  Note that the the values of primitive types
# cannot be set from this method.  The primitive types are those based on
# String, Boolean, Integer, or Float.  Additional fields may be added to the
# type simply by assigning to the field (ex. self.myfield = 3).  The init()
# method will be called for all types in the hierarchy, starting from the
# least specific (last entry in method resolution order).
def init(self):
    pass

# This method is used to convert the type to a string.  If your type is already
# a primitive, you don't need to supply this method.  If your type is not a
# primitive, you should supply it.  Inside of this method you are free to use
# cgl() calls.  If you return a string value, it will be used in place of
# any code produced by means of cgl() calls.  To find the correct string()
# method, the system will use the default python method resolution order.
def string(self):
    pass

# If you desire, the methods may be specified using CGL code rather than
# python code.  The r''' string syntax is used to avoid the need to
# escape backslashses.  Here is an example:
#         cgl(r'''
#           #macro string(self)
#             #if self.conditionVariable
#               x = @self.varOne;
#             #else
#               x = @self.varTwo;
#             #end
#           #end
#         ''')
```