Automatic Code Generation from Design Patterns

Frank Budinsky, Marilyn Finnie, Patsy Yu
Toronto Software Laboratory

John Vlissides
T.J. Watson Research Center

Abstract

Design patterns raise the abstraction level at which people design and communicate design of object-oriented software. But design patterns still leave the mechanics of their implementation to the programmer. This paper describes the architecture and implementation of a tool that automates the implementation of design patterns. The user of the tool supplies application-specific information for a given pattern, from which the tool generates all the pattern-prescribed code automatically. The tool has a distributed architecture that lends itself to implementation with off-the-shelf components.

1 Introduction

Design patterns are an attempt to capture expertise in building object-oriented software. A design pattern describes a solution to a recurring design problem in a systematic and general way. But beyond a description of the problem and its solution, software developers need deeper *understanding* to tailor the solution to their variant of the problem. Hence a design pattern also explains the applicability, trade-offs, and consequences of the solution. It gives the rationale behind the solution, not just a pat answer. A design pattern also illustrates how to implement the solution in standard object-oriented programming languages like C++ and Smalltalk.

Over the past two years, a vibrant research and user community has sprung up around this topic. Pattern-related discourse has flourished at object-oriented conferences, so much so that there is now a conference[1] devoted entirely to patterns. Books [9, 6, 4], articles [8, 3, 5], and at least one non-profit organization (The Hillside Group) have appeared to further the field. One of the most widely cited books is *Design Patterns: Elements of Reusable Object-Oriented Software* [9], a catalog of 23 design patterns culled from numerous object-oriented systems.

Design Patterns has proven popular with novice and experienced object-oriented designers alike. It gives them a reference of proven design solutions along with guidance on how to implement them. The discussions of consequences and trade-offs furnish the depth of understanding designers

[1] "Pattern Languages of Programming," held annually on the campus of the University of Illinois.

need to customize the implementation to their situation. And the names of the patterns collectively form a vocabulary for design that helps designers communicate better.

At least one burdensome aspect remains, however: the patterns must be implemented each time they are applied. Designers supply application-specific names for the key "participants"—classes and objects—in the pattern. Then they implement class declarations and definitions as the pattern prescribes. If this were all to implementing a pattern, then it wouldn't be a big chore. But different trade-offs may call for radically different arrangements of classes and their implementations. Moreover, trade-offs often work synergistically, resulting in a proliferation of variant implementations—too many to support through conventional code reuse techniques.

This paper describes an approach to this problem. We present a tool for generating design pattern code automatically from a small amount of user-supplied information. We also describe how the tool incorporates a hypertext rendition of *Design Patterns* to give designers an integrated on-line reference and development tool. This tool is not meant to replace the material in the book. Rather, it takes care of the mundane aspects of pattern implementation so that developers can focus on optimizing the design itself.

2 Design Pattern Format

Each design pattern is divided into sections according to the following template. The template lends a uniform structure to the information, making design patterns easier to learn, compare, and use. For a more detailed description of the design pattern form along with examples of actual design patterns, refer to the *Design Patterns* book [9].

Name The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

Intent A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As Other well-known names for the pattern, if any.

Motivation A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Applicability What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Structure A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [16].

Participants The classes and / or objects participating in the design pattern and their responsibilities.

Collaborations How the participants collaborate to carry out their responsibilities.

Consequences How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

Sample Code Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Known Uses Examples of the pattern found in real systems. Each pattern includes at least two examples from different domains.

Related Patterns What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

3 Generating Code Automatically—An Example

A design pattern only *describes* the solution to a particular design problem; it is not itself code. Consequently, some users find it difficult to make the leap from the pattern description to a particular implementation, even though the pattern includes sample code. Others might have no trouble translating the pattern into code, but they still find it a chore, especially when they have to do it repeatedly. Moreover, a design change might require substantial reimplementation, because different design choices in the pattern can lead to vastly different code.

Our design pattern tool helps alleviate these problems. From just a few pieces of information—normally application-specific names for the participants in a pattern along with choices for the design trade-offs—the tool can create class declarations and definitions that implement the pattern. The user then adds this code to the rest of his application, often enhancing it with other application-specific functionality.

The tool also incorporates an on-line, hypertext rendition of *Design Patterns*. The patterns lend themselves to a hypertext format because they are richly cross-referenced. The on-line version gives users convenient access to the material, letting them follow links between patterns instantaneously and search for information quickly.

3.1 Section Pages

The tool displays the sections of a pattern (Intent, Motivation, etc.) in separate **pages**. These pages mirror the corresponding sections in the book. For example, Figure 1 shows the Intent page for the Composite design pattern. From there, the user can access the other sections of the Composite pattern either randomly or in sequence. The user can jump to the Intent section of any other pattern as well, which is useful for comparison purposes. In addition, the text on the page may embed additional hypertext links to related discussions elsewhere, providing quick and easy cross-referencing.

Clicking on the right arrow button beside the “Intent” heading advances to the next section in sequence, namely Composite’s Motivation section (Figure 2). Notice the similarity between this page and the preceding Intent page. We have given every page the same basic look-and-feel to ensure a consistent and intuitive interface. To jump to any another section in the Composite pattern, the user clicks on the name of the section in the list near the top of the page. Clicking on the name of a pattern in the list near the bottom of the page jumps to the same section in that pattern.

Figure 1: Intent page of Composite pattern

3.2 Code Generation Page

Augmenting the sections from the book is an additional page titled "Code Generation" for each design pattern. This page comes immediately after the pattern's "Related Patterns" page and can be accessed either sequentially or randomly just as other pages. The Code Generation page lets the user enter information from which to generate a custom implementation of the pattern. The page is fully integrated with the book text: references to participants and other details are actually hyperlinks back to the relevant discussion in the pattern. The effect is similar to a context-sensitive help system.

Composite's code generation page appears in Figure 3. Some parts of this page are specific to code generation for Composite, other parts are specific to all code generation pages, and the remaining parts are common to all pages:

- Composite-specific parts are input fields marked "`Component:`," "`Composite:`," and "`Leaf:`."
- Code generation-specific parts include the selection list marked "`Goal:`" (currently set to "*Generate declarations*") and the "OK" button to its right.
- Other parts are navigation aids common to all pages.

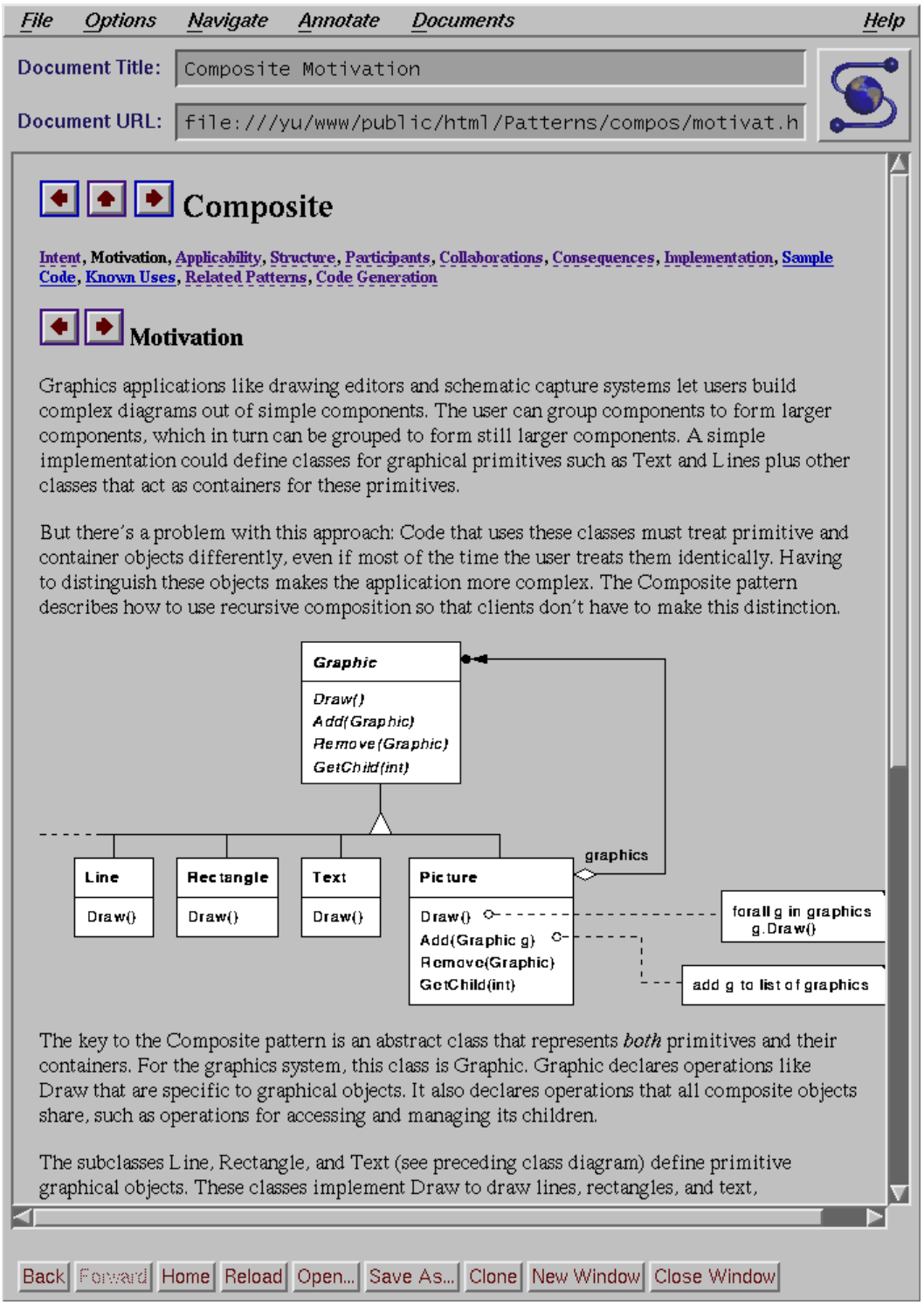

Figure 2: Motivation page of Composite pattern

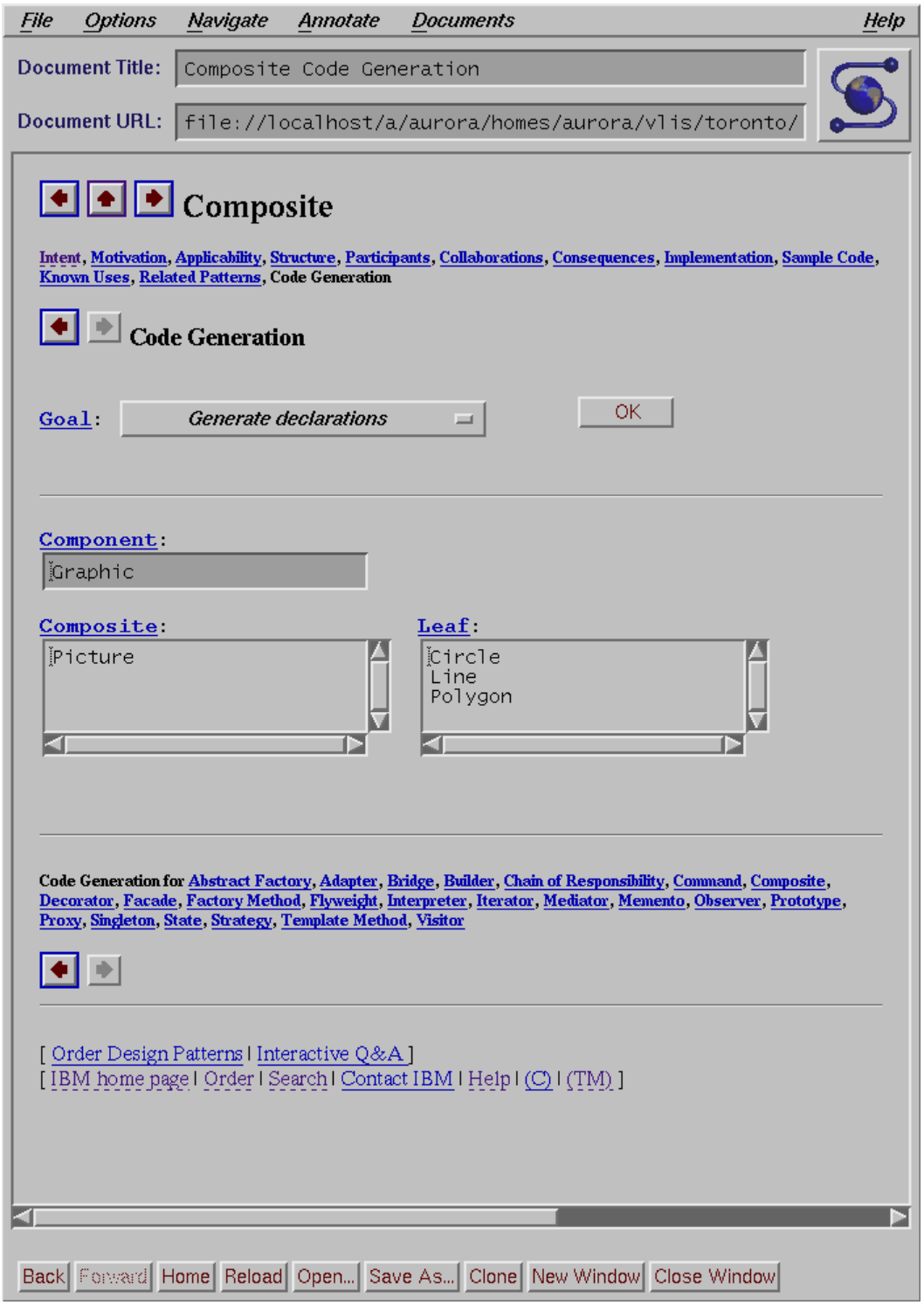

Figure 3: Code Generation page of Composite pattern

The selection list lets the user select one of several tasks. "*Generate declarations,*" the current selection, produces declarations for the classes that implement the pattern; we'll describe other tasks shortly. To carry out the selected task, the user presses "OK."

The input fields let the user specify application-specific names for the pattern participants—in this case a Component, one or more Composites, and one or more Leaves. If the user needs help remembering the roles of these participants, he can click on the corresponding labels above the input fields to jump to the description on the Participants page. The user can also see input values that implement the example in the Motivation section. To fill the input fields with these values, the user selects "*An Example*" from the "`Goal:`" selection list and presses "OK."

3.3 Generating Declarations

Figure 4 shows the result of choosing "*Generate declarations*" as the goal and pressing "OK" with the inputs shown in Figure 3. The page that appears contains the text of a C++ header file declaring classes for the specified participants. The page in Figure 4 is scrolled to show declarations for the `Graphic` Component abstract base class and the `CompositeGraphic` Composite abstract base class. The user may save the generated code in a file using the browser's "Save As..." command.

The operations shown in the class declarations were generated automatically based on the pattern's prescription of participant responsibilities. These responsibilities can vary according to the design trade-offs articulated in the pattern, and the code we see in Figure 4 reflects one set of trade-offs. One such trade-off concerns child management operations (`Include` and `Exclude` in this case), which are defined in the Composite class only. As a consequence, the `Graphic` base class declares a `GetComposite` smart downcast to let clients recover the Composite interface when all they have are references to `Graphic` objects.

3.4 Selecting Different Trade-offs

Users are not forced to accept these trade-offs, of course. To choose different ones, the user selects "*Choose implementation trade-offs*" from the "`Goal:`" selection list (Figure 5) and presses "OK." The resulting trade-offs page appears in Figure 6.

The trade-offs page lists the trade-offs in the pattern. The user selects among them by clicking on the corresponding buttons. Some buttons are exclusive, others are not. For example, the user may choose to include child management operations in the base class simply by pressing the button marked "all classes" under the heading "Declare child management operations in" near the bottom of the page. Doing so maintains a uniform interface for Leaf and Composite classes, but it raises the possibility of run-time error should a client try to add or remove a child from a Leaf object. The alternative puts these operations solely in the Composite class, as was the case when we generated the declarations earlier. Either way, the user can include any or all of the child management operations listed under "Child management operations:".

Here again, key words in the button labels are hypertext links. Should the user forget the details behind a trade-off, he can click on the appropriate link to jump back to the corresponding discussion in the pattern. Once the user has finished choosing trade-offs, he presses "OK" to commit the changes and return to the code generation page.

Figure 4: Generated declarations

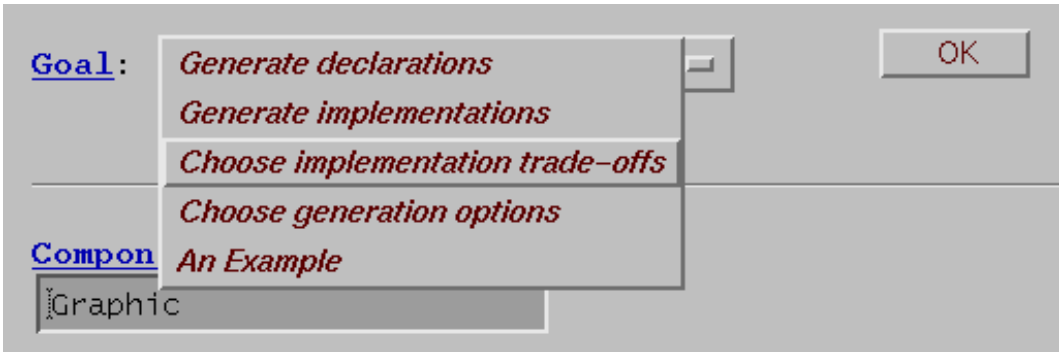

Figure 5: Selecting the trade-offs page

3.5 Global Code Generation Options

Users can also control certain generation parameters that apply to all the patterns. Selecting "*Choose generation operations*" from the "`Goal:`" selection list and pressing "OK" yields the page shown in Figure 7. The user can choose to

- limit files names to an eight-plus-three character format (to generate `#include` directives properly for the target platform),
- include standard C++ operations such as copy constructors and virtual destructors,
- include operations that generate an execution trace at run-time, and
- include a `main` routine that implements an executable example based on the Motivation section. The user can compile, run, and exercise the code through a simple command-line interface catered to each example.

Figure 8 shows part of the result of choosing "*Generate implementations*" from the "`Goal:`" selection list with the generation options selected in Figure 7. Note the copy constructor and assignment operator implementations as well as the `main` routine implementation.

4 System Architecture

The design pattern tool's architecture characterizes the implementation-independent aspects of its design. We describe the architecture here both to clarify our design goals and to provide a backdrop for the discussion of the implementation that follows.

4.1 Goals

There are two contexts in which to discuss design goals: our goals for the development and maintenance of the tool, and our goals for the tool itself. Development and maintenance goals affect us as designers and implementors of the tool; goals for the tool itself impact how well the end-user receives and exploits the tool. We'll refer to the former simply as "development goals" and the latter as "end-user goals."

We have three primary development goals:

Figure 6: Composite trade-offs page

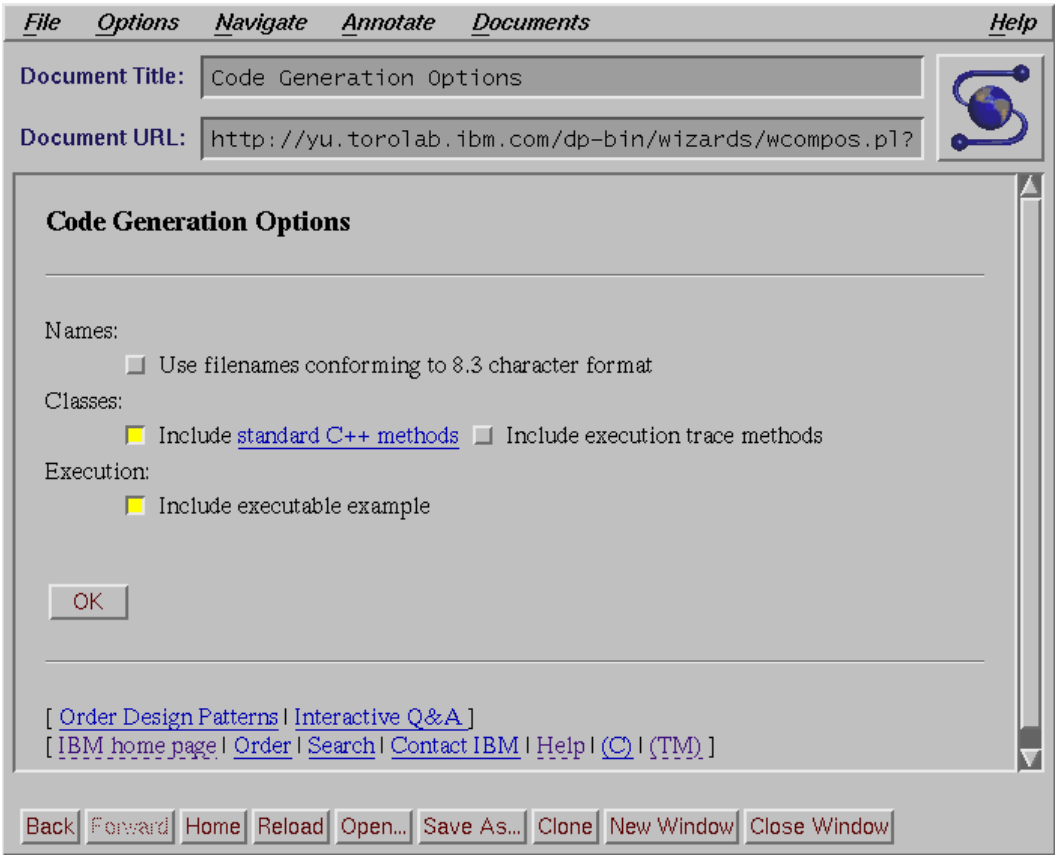

Figure 7: Global code generation options

1. *Fast turn-around.* Because most of this work is new and experimental, we must be able to modify the system as quickly as possible. We can't afford delays in implementing or testing new functionality—we have too many degrees of freedom to explore.

2. *Flexibility.* Fast turnaround means little if adding a new feature requires reimplementing a sizable chunk of the system. A minor change in functionality should incur a correspondingly small implementation effort. But even major changes should be well-contained: support for generating code in a different programming language shouldn't force an overhaul of the entire system.

3. *Ease of specification.* Automatic code generation can be difficult to implement, especially if the only medium of expression is a conventional programming language. We wanted a higher-level way to specify how code gets generated without limiting flexibility—two conflicting requirements.

For end-users, three additional goals are paramount:

1. *Utility.* The tool must be easy to use, and the code it generates should be ready-to-use; that is, the user should have to make a minimum of changes to make the generated code work in his application.

2. *Seamless integration.* Given a tool that leverages the material in *Design Patterns*, the user will want to refer to that material as he uses the tool. To be most effective, therefore, the

Figure 8: Generated implementations

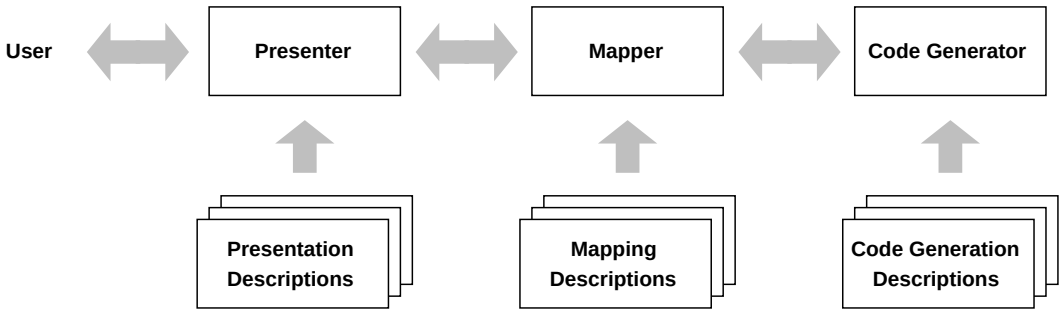

Figure 9: Architecture of design pattern tool

book's contents—the precise topic of interest to the user—should be accessible from the tool. Hence the tool and the book material must be tightly integrated to make going from one to the other easy.

3. *Transparent distribution*. This work is experimental and therefore on-going. We couldn't expect to deliver a polished product without feedback from users. We foresaw rapid evolution of the tool as users discovered what worked well and what didn't. Incorporating such feedback quickly was a challenge, and supplying users with updated versions of the tool was an even greater challenge. We needed a way to keep users up-to-date with the latest functionality without major disruption or prohibitive upgrade costs.

One approach to achieving these goals would be to build a custom application from scratch. It would implement the hypertext presentation and code generation for all the patterns in one large application. We would use conventional development tools, including a general-purpose programming language and environment, user interface development tools, and support libraries. But we quickly dismissed this approach as too slow and expensive, with a high likelihood of producing an insufficiently flexible design. Too much had to be home-grown rather than reused.

4.2 Characteristics

Over time we developed an architecture having three fundamental characteristics:

1. It accommodates existing tools and applications wherever possible.
2. It makes pervasive use of interpreted specifications to minimize turn-around time during development.
3. It partitions functionality so that key components can be distributed, letting us upgrade the system without involving or disturbing users.

Figure 9 depicts the architecture. It has three components:

1. The **Presenter** implements the user interface specified by **Presentation Descriptions**, which it interprets.
2. The **Code Generator** generates code that implements a pattern. It interprets **Code Generation Descriptions**, each of which captures how to generate the code for a given pattern.

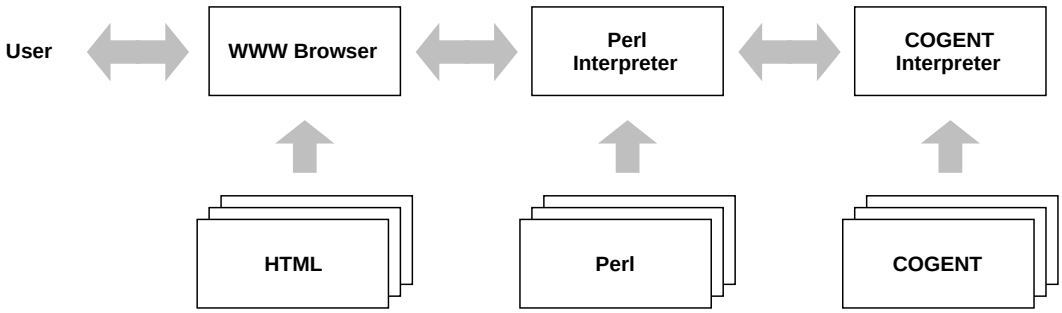

Figure 10: Implementation of design pattern tool

3. The **Mapper** specifies how the user interface and code generator components cooperate. It interprets **Mapping Descriptions** that specify connections and interactions between the other two components.

This partitioning yields several advantages over more monolithic approaches. The components are decoupled from each other, letting us change them independently. We can make an aesthetic modification to the user interface without any changes to the other components. Even substantive changes to the user interface tend to propagate no further than the Mapper, because the mechanics of code generation are largely independent of the mechanics of collecting information from the user. Conversely, improvements to the generated code and changes in the underlying implementation infrastructure can be accommodated without modifying Mapper or Presenter functionality. In fact, the most common sort of changes—bug fixes—rarely span two or more components. Best of all, the interpreted nature of the components lets us see the effects of a change immediately.

Another nice property of the partitioning is that it maps well to existing software, both in granularity and capability. This greatly reduced the development burden and freed us to focus on more novel aspects of the implementation. We can also distribute the components across machines and computing platforms. Distribution lets you take advantage of more powerful hardware than you might have on your desk, and a multiplatform capability widens the tool's appeal.

5 Implementation

Figure 10 is a more detailed version of Figure 9 revealing the implementation technologies we use.

The Presenter is simply a World-Wide Web (WWW) browser such as WebExplorer, Mosaic, or Netscape that displays pages specified in HTML [10]. Whenever the user enters information into a Code Generation page, the browser transmits the information to the Mapper through the Web-standard CGI interface [10]. Our Mapping Descriptions are Perl scripts [18]; hence the Mapper is a Perl interpreter. The Perl scripts invoke a COGENT (COde GENeration Template) interpreter, which serves as the Code Generator. COGENT is a simple code generation specification language we developed. COGENT lets us describe the generated code succinctly and change what's generated quickly.

The Web-based approach provides a natural partition between user interface and input processing components. In fact, the Web architecture itself makes it just as easy to process inputs remotely as it does locally. That means we can distribute the processing by putting the Mapper and Code Generator on a server with only the Presenter on the client—that is, the user's machine.

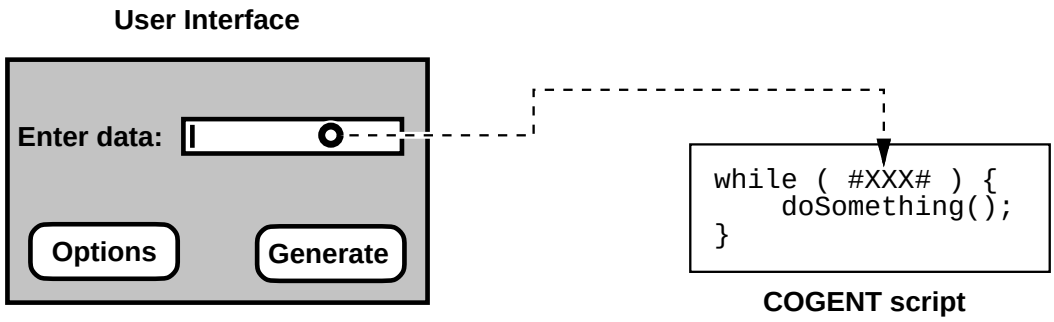

Figure 11: Mapping an input value to a COGENT parameter

5.1 Web Browser as Presenter

The Web browser-HTML combination offers a ready platform for presenting and navigating text of rich structure and formatting. The preexistence of this platform saved us much implementation effort. All we had to do was translate the book contents into HTML and GIF files, a straightforward exercise. We added just two enhancements to the text: we incorporated a user interface for navigation from page to page, and we replaced textual cross-references with hypertext links. Designing an easy-to-use navigation interface took some trial and error, but overall the enhancements were easy to make.

Adding the Code Generation pages was more difficult, largely because the typical Code Generation page is not static like the pages from the book—its appearance may depend on the trade-offs the user selects. For example, some patterns require additional input fields when certain trade-offs are in effect. Unfortunately, current versions of HTML do not allow modifying a page in-place. (We describe our solution to this problem later when we discuss the Mapper implementation.)

On the other hand, the pages for specifying implementation trade-offs (e.g., Figure 6) and global code generation options (Figure 7) are simple HTML forms. Providing context-sensitive help for these pages was just a matter of linking key words and phrases back to the relevant discussions in the pages from the book. The result is an effective interface for a modest effort.

One other aspect of the user interface was difficult to support in HTML: persistent fields. We wanted users to see default input values on their initial visit to a Code Generation page. If they changed any values, the changes should reappear the next time they visited the page—that is, the last input values should persist across visits. HTML does not support these semantics, so we relegated their implementation to the Mapper, as explained in the next section.

5.2 Perl Interpreter as Mapper

The Mapper has two basic responsibilities:

1. Connect user interface elements to COGENT parameters.
2. Respond to user actions by either
   (a) displaying another page, or
   (b) selecting an appropriate COGENT script for the Code Generator to interpret.

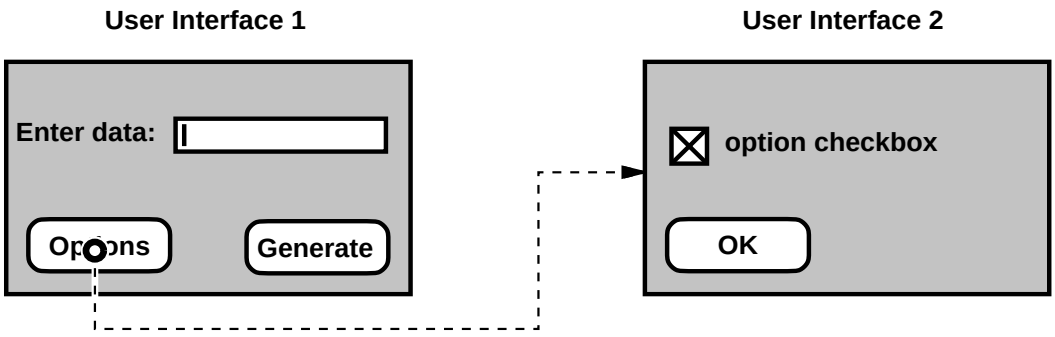

Figure 12: Invoking another user interface

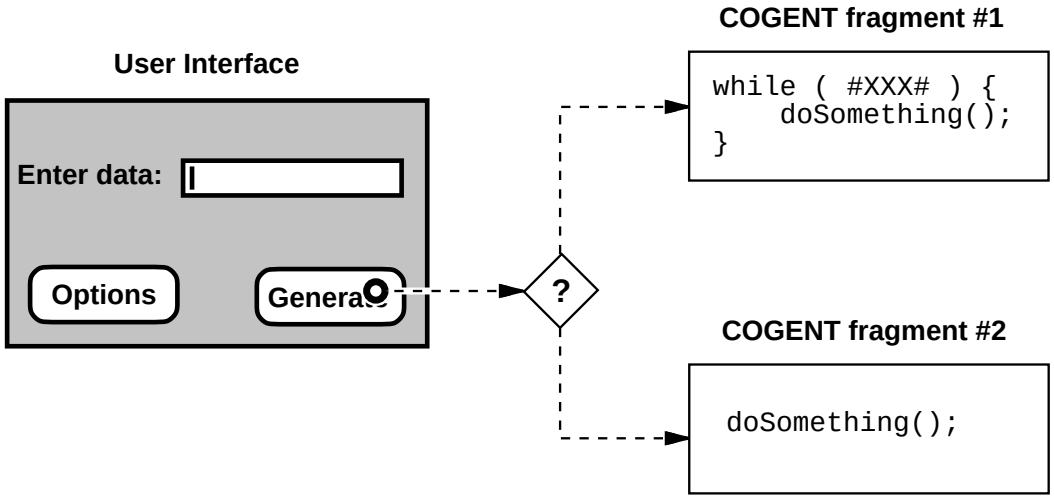

Figure 13: Selecting COGENT code and invoking the COGENT interpreter

Figure 11 illustrates the first responsibility. The Mapper takes values from input fields in the user interface and supplies them to the code generator in the form of parameters to the COGENT script. By adhering to standard naming conventions in the HTML forms and COGENT files, this mapping can be made through a single Perl library function.

The other Mapper responsibility is to respond to user actions. Figure 12 shows a case where a user action—pressing an "Options" button—produces another HTML page for specifying user options.

In the previous section, we mentioned that the appearance of a Code Generation page often depends on the trade-offs a user selects, but HTML does not allow changing a page dynamically. Thus an existing Code Generation page can't change to reflect a change in trade-offs. The Mapper helps us get around this problem. It may not be apparent to the user, but Code Generation pages are actually *generated* whenever they are accessed, as opposed to simply retrieved. When the user commits a set of trade-offs, the browser returns not to the previous Code Generation page but to a newly created one, which may reflect the new trade-offs in its user interface. The Mapper is the appropriate place to implement this behavior: architecturally, the Presenter only presents the user interface as specified by Presentation Specifications. How the user interface reflects the semantics of code generation is the Mapper's responsibility.

An example of a user action that results in code generation appears in Figure 13. Here, the driver selects one of two possible COGENT code fragments depending on some criterion (the state of the option checkbox in Figure 12, for example). Based on this criterion, the Mapper will either choose the COGENT fragment that calls `doSomething()` repeatedly, passing in the user-supplied value `#XXX#`, or it will choose another fragment that executes `doSomething()` just once.

In general, such logic can be expressed in either the Mapping Description (i.e., Perl) or in the Code Generation Description itself (i.e., COGENT). For example, the Perl script can choose between two hard-wired COGENT fragments, or it may supply a parameter to the COGENT code to let it make the choice. Deciding which of these approaches is best boils down to a complexity trade-off between the Mapping and Code Generation Descriptions. We discuss this issue further in the next section.

An added responsibility of the Mapper arises as an artifact of our implementation. Beyond choosing the page to display based on a user's inputs, the Mapper must initialize inputs to the proper defaults or—if the user changed the inputs at any point—the most recent inputs. Since HTML cannot express this logic, it is the Mapper's responsibility. Persistence is actually a side-effect of the Mapper's other responsibilities: the same mechanism for storing and retrieving user settings lets the Mapper store and retrieve other input values as well, such as variable names. Thus the Mapper ensures that the most recent input values persist across visits to a pattern's Code Generation page.

5.3 COGENT Interpreter as Code Generator

Once the Mapper has chosen the appropriate COGENT script, it tells the Code Generator to interpret it. The prime benefit of using COGENT is that we can modify the generated code without disturbing other parts of the system. For example, we can generate Smalltalk instead of C++ simply by writing new COGENT scripts.

We were careful to design COGENT so that it's easy to transform source files into COGENT scripts. We wanted to be able to write prototypical code fragments (such as the sample code in *Design Patterns*) and convert them to COGENT format with a minimum of editing. To speed the process, we augmented our local text editors (`emacs` and `lpex`) with COGENT parsers to make the editing job even easier.

An important thing to consider when implementing a code generator is how much variation logic belongs in the Mapper component (Perl) versus the Code Generator component (COGENT). The more decision-making done in COGENT, the more information the Mapper must pass to the Code Generator, and the more complex the COGENT scripts. Fear of inventing yet another Turing-complete language constrained our design of COGENT to a bare minimum of constructs—a set sufficient to express five common variations that support the variability we've encountered in design pattern implementations:

1. *Simple macro replacement with optional transformation.* Optional transformation functions modify a macro before it is expanded. For example, `XXX:toupper` expands to `XXX` with each character capitalized: supplying `Hello` as a parameter would yield `HELLO` upon expansion. Several common transformations are built into the language. COGENT lets you define your own transformation function as well.

2. *Conditional inclusion.* Include other COGENT code according to a predicate, such as the definition or non-definition of a macro.

3. *Repetitive inclusion.* Include other COGENT code repeatedly. Macros can be assigned multiple values. Each time a macro is assigned a value, it actually appends the value to the end of its list of values. When a macro that contains a list of values is expanded normally, the values are concatenated and returned as a single value. But when the macro is supplied as a parameter to a `<repeat>` directive, the number of values in the list determines the number of iterations. Each consecutive iteration uses the consecutive value in the list rather than the concatenation of those values.

4. *Code reuse.* A **code segment** is simply a named section of COGENT code that can be interpreted or expanded where referenced. Code segments give you a way to decompose code into smaller, reusable pieces, much like procedures do.

5. *Macro assignment in code segments.* At one extreme, macros are *always* assigned values ahead of time (e.g., through arguments to the interpreter at invocation), and then code segments are expanded with those values. This extreme is simple but not very flexible—it's hard to vary the way macros get initialized. At another extreme, macros are assigned *only* by expanding code segments that define macros, thereby delegating the assignments to the code segments. That lets you vary the assignments without modifying code that uses the macros.

This set of constructs, coupled with user-defined transformation functions, lets us express nearly all code variants entirely in COGENT. Nonetheless, implementing everything in COGENT can be inconvenient, particularly when user-supplied transformations proliferate. The implementation of these transformations is not well-integrated with the COGENT language—they are independent executables that COGENT calls. The main problem here is that they are platform-specific: shell scripts on AIX, REXX scripts on OS/2, etc. Performance is also a concern, since each user-supplied transformation forks a process. We've used user-defined transformations primarily as a prototyping mechanism for built-in transformations; once we encountered a particular transformation more than a couple of times, we made it a built-in transformation (e.g., `toupper`). As a result, we limit our COGENT code to expressing language-specific syntax and semantics that require a minimum of user-supplied transformations, leaving the rest to the Mapper Perl scripts.

COGENT Example

Figure 14 shows a fragment of COGENT code that generates C++ function definitions. To generate the code, the Mapper invokes the COGENT interpreter with the following arguments:

```
generate USER_DATA=First USER_DATA=Second TRACE_CODE=1 example.t
     -- GEN_CODE > example.C
```

where

- `generate` invokes the interpreter.
- `USER_DATA` is a macro that will be expanded twice, first with `First` and then with `Second`.
- `TRACE_CODE` is a macro that's used as a flag to indicate whether extra tracing code should be emitted for debugging purposes.
- `example.t` is the file containing COGENT code.

```
<code GEN_CODE>
#include <stdlib.h>
<repeat USER_DATA>
#A_SEGMENT#
</repeat USER_DATA>
#MAIN#
</code GEN_CODE>

<code A_SEGMENT>
<clear CLASS_NAME>
<define CLASS_NAME>#USER_DATA#Class\</define>
void
#CLASS_NAME#::#CLASS_NAME#_operation () {
<test ?TRACE_CODE>
    cout << "... void "
         << "#CLASS_NAME#::#CLASS_NAME#_operation ()"
         << endl;
</test>
}
</code A_SEGMENT>

<code MAIN>
void main () {
    cout << "In main() ... " << endl;
}
</code MAIN>
```

Figure 14: COGENT code fragment

- `GEN_CODE` specifies the code segment with which to generate code.
- `example.C` is the file in which to store the generated C++ code.

Note that `USER_DATA`, `TRACE_CODE`, and `GEN_CODE` are not COGENT keywords but merely user-defined macro names used in `example.t`.

The first statement in Figure 14 marks the beginning of the `GEN_CODE` code segment. The line after it will be emitted as is, because it is not modified by COGENT directives. The next line tells the COGENT interpreter to iterate over the subsequent code up to the `</repeat>` directive. In each iteration, `USER_DATA` will be assigned one of the values supplied to the interpreter when it was invoked, in the order they were supplied (`First`, then `Second` in this case).

Whenever a macro name appears between "#" symbols (e.g., #`USER_DATA`#), it is expanded to its current value. When the name of an undefined macro is referenced in this way, it expands to the code segment with that name, if any.

In this case, `A_SEGMENT` is the code segment defined immediately after `GEN_CODE`. `A_SEGMENT` produces a function definition. It defines a `CLASS_NAME` macro that must be cleared on each iteration, because the `<define>` directive concatenates the definition and the macro's existing

```
#include <stdlib.h>
void
FirstClass::FirstClass_operation () {
    cout << "... void "
         << "FirstClass::FirstClass_operation ()"
         << endl;
}

void
SecondClass::SecondClass_operation () {
    cout << "... void "
         << "SecondClass::SecondClass_operation ()"
         << endl;
}

void main () {
    cout << "In main() ... " << endl;
}
```

Figure 15: Generated code

values. The code in `A_SEGMENT` will be evaluated and substituted where it was referenced. Similarly, `#MAIN#` will evaluate to the `MAIN` code segment.

Figure 15 shows the code generated from the COGENT code in Figure 14 given the parameters mentioned earlier.

6 Observations

Several characteristics of our design became clear only after we had implemented it. Indeed, some weren't evident until we had used it for a while. For example, though we had predicted some of the benefits of distributing the user interface, code generation, and mapping components, we didn't realize such distribution would enable interactive, near-real-time customer support. A late addition was a "Q&A" link at the bottom of every page. Clicking on that link takes the user to a page where he can type in questions. Someone on our end can respond immediately through the same page, exploiting rich text, embedded diagrams, links to supplemental materials—the full range of HTML capabilities.

The following are other characteristics we noted in retrospect.

6.1 Presentation

Although a Web-based interface works well in many ways, HTML's current limitations (such as static pages and lack of support for drag-and-drop and other direct-manipulation techniques) have lead to compromises in the user interface design. Another drawback lies in the Web's client / server split. Because the Code Generator relies on HTML forms, it must run in the server process space.

As a result, the user must save generated code on his machine explicitly. The split also complicates saving trade-off and code generation settings across sessions.

Without abandoning HTML as a user interface description, we could overcome many of these limitations by designing our own customized browser or by adopting a browser that supports extension through an interpreted language. But by constraining ourselves to standard Web browser capabilities, we maximize the audience for our tools. Moreover, HTML is undergoing vigorous development in the service of an exploding user community. So we expect these limitations to disappear gradually.

It's easy to envision more sophisticated user interfaces. One area for improvement is the integration of pattern and code generation. Here are two examples:

1. As the user selects trade-offs, associated parts of the pattern change to reflect them. The relationships in the Structure diagram would change, for example, as would the descriptions of the participants and their collaborations. Mutually-exclusive trade-offs would elide each other automatically to reduce clutter in the trade-offs page.

2. Rather than having a separate Code Generation page, the user could specify code generation information through the pattern itself. A direct-manipulation interface could let the user edit the Structure diagram to add, remove, reorganize, and otherwise redefine the class structure, subject to pattern constraints.

Interfaces like these offer a much more dynamic and compelling metaphor than the current HTML forms-based interface.

6.2 Mapping

Perhaps the most troublesome aspect of the Web-based implementation had to do with making data persist across visits to Code Generation pages. As long as the user jumps from page to page and backtracks only via the browser's "Back" button, he will see only the most recent inputs *de facto*. But when the user shuts down and restarts the browser, or if he revisits a page through a hyperlink rather than the "Back" button, he may see obsolete data—that is, unless the system saves input information *and recreates* potentially stale pages when they are accessed through a link. This approach required

- persistent storage on the server side for each client.

- a unique identifier per client so that information can be saved on a per-user basis, thus avoiding potential conflicts. Because we didn't want to bother the user for this identifier, the Mapper synthesizes it transparently.

An alternative approach would let the browser save client-specific state locally. Unfortunately, no current Web protocol supports this capability.

6.3 Code Generation

Though we are satisfied with COGENT as a way to express code generation, we're much less satisfied with the way we integrate the generated code into an application—that is, by cut and paste. Two problems are endemic to the cut-and-paste approach:

- *Invasiveness.* The user must understand what to cut out and where to paste it, and both may be nonobvious.
- *Unreversibility.* Once a user has incorporated generated code into his application, any change that involves regenerating the code will force him to reincorporate it into the application. As a corollary, the user can't see changes to the generated code through the tool.

Clearly we need a better way to decouple generated code from a user's modifications. One approach involves doubling the number of classes the tool generates. For each class generated currently, we generate *two* classes in its stead: a **core** class that's identical to the original class except for its name, and a trivial subclass thereof—the **core subclass**, whose name matches that of the original class. Thus code that instantiates the original class ends up instantiating the corresponding core *sub*class.

Any user-specified changes to the generated code must be made to the core subclass only; the user never alters the core class. The inheritance relationship lets the user redefine or extend generated operations, add new operations, and add new instance variables. Should the code require regeneration later, the tool overwrites *only* the core class. The user's changes remain unaffected. Unless the user subsequently regenerates radically different code with the tool, the core subclass should continue to work with the new core class.

This approach has been used successfully in ibuild, a user interface builder [17], and it should work in this context as well. There is however at least one case in which it does not work well, and that's when the user must incorporate generated code into an *existing* class hierarchy. This is usually not a problem for user interface builders, which generate code that's largely self-contained and has comparatively few connections to the rest of the application. But the code that the design patterns tool generates is more broadly applicable. It is therefore likely that the generated code will require more intimate integration with existing code.

Our initial solutions to this problem focused on using multiple inheritance to mix generated code into existing code. Multiple inheritance isn't a comprehensive solution, though, because not all languages support it. Besides, it's arguably too low-level for what we're trying to do anyway. In our experience, multiple inheritance is best suited to designing type flexibility into a system *a priori*, specifically through interface mixins. In contrast, our goal is to integrate code that wasn't designed to work together.

So we need a mechanism that supports code integration explicitly and conveniently. Subject-oriented programming [13] promises just that, and we are exploring its ramifications for our tool.

7 Related Work

Most elements of this work have had long research histories. The pattern concept arose from the work of Christopher Alexander in the Seventies [1, 2]. Alexander sought to capture on paper the essence of great architecture in a structured, repeatable way. He focused on the design and construction of buildings and towns, but gradually his ideas took root in the software community, blossoming only recently. Actually, *Design Patterns* was influenced less by Alexander and more by the PhD research of Erich Gamma, which accounts for the substantial differences in these works.

Template-based code generation is a well-researched area as well, going back to Floyd's work on symbol manipulation specification [7]. In the wake of Knuth's seminal paper on context-free

languages [14], the Seventies and Eighties saw prodigious research into attribute grammars. They were applied broadly, first as a vehicle for expressing programming language semantics, then as an aid in compiler construction, and ultimately as a basis for generating entire programming environments [15, 11, 12]. The foundations of today's commercial software development tools—user interface builders, 4GL application generators, "wizards," and case tools of every persuasion—rest on these research strata.

Finally, there is the Web. Few could have missed its rise to ubiquity. We are convinced it represents a whole new application platform, although there is much controversy over its ultimate destiny: whether it will assimilate today's applications or vice versa, for example. But even as it is, the Web gave all we had hoped for—a viable front-end to our design pattern tool—and some things we hadn't thought to hope for, like interactive Q&A.

In fact, the Web, design patterns, and our tool share a salient attribute: each is deliberately unnovel in its constituent parts, but as an amalgam, each offers compelling new capabilities. The Web introduced no new technologies; it merely composed existing ones synergistically. Likewise, design patterns recast proven techniques in a new expository form. We merely combined equal parts patterns, the Web, and code generation to help automate some mundane aspects of pattern application.

8 Conclusion

Automatic code generation adds a dimension of utility to design patterns. Users can see how domain concepts map into code that implements the pattern, and they can see how different trade-offs change the code. Once generated, the user can put the code to work immediately, if not quite noninvasively.

Much remains to be explored. The concept of design patterns is in its infancy—the tools that support the concept, even more so. Our tool is just a start. It exploits only a fraction of the intellectual leverage that design patterns provide. For example, the tool is limited to system design and implementation; it does not support domain analysis or requirements specification or documentation or debugging. All of these areas stand to benefit from design patterns, though it might not be clear at this point exactly how. Then again, the principles underpinning our tool were not clear until we had experience using patterns for design. Application is the first and necessary step; only then can we hope to automate profitably.

References

[1] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[2] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

[3] Kent Beck. Patterns and software development. *Dr. Dobb's Journal*, 19(2):18–23, 1994.

[4] Peter Coad, David North, and Mark Mayfield. *Object Models: Strategies, Patterns, and Applications*. Yourdon Press, Englewood Cliffs, NJ, 1995.

[5] James O. Coplien. Generative pattern languages: An emerging direction of software design. *C++ Report*, 6(6):18, 1994.

[6] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1995.

[7] Robert. W. Floyd. A descriptive language for symbol manipulation. *Journal of the ACM*, 8:579–584, October 1961.

[8] Richard Gabriel. Pattern languages. *Journal of Object-Oriented Programming*, 6(2):14, 1994.

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

[10] Ian S. Graham. *The HTML Sourcebook*. J. Wiley & Sons, New York, 1995.

[11] Susan L. Graham. Table-driven code generation. *Computer*, 17(8), August 1980.

[12] A.N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, December 1986.

[13] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In *ACM OOPSLA '93 Conference Proceedings*, pages 411–428, Washington, D.C., October 1993.

[14] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[15] K. Raiha. Bibliography on attribute grammars. *SIGPLAN Notices*, 15(3):35–44, March 1980.

[16] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[17] John M. Vlissides and Steven Tang. A Unidraw-based user interface builder. In *Proceedings of the ACM SIGGRAPH Fourth Annual Symposium on User Interface Software and Technology*, Hilton Head, SC, November 1991.

[18] Larry Wall and Randal L. Schwartz. *Programming perl*. O'Reilly & Associates, Inc., Sebastopol, CA, 1991.

Biographical Sketches

Frank Budinsky is an Advisory Engineer at the IBM Toronto Software Laboratory. He has developed numerous object-oriented applications and is currently lead designer for the Compound Document Framework support to be included in IBM Open Class. Frank's interests include subject-oriented programming and design patterns, particularly their applicability in framework programming environments. Frank holds BS and MS degrees in Electrical Engineering from the University of Toronto.

Marilyn Finnie is a member of the Advanced Tools Development area of the Toronto Laboratory. She joined IBM in 1983 with a BSc from the University of Western Ontario. She worked in various projects dealing with distributed systems (electronic mail protocols, X.500, OSF DCE) before becoming involved with design patterns and code generation technology.

Patsy Yu is a developer at the Toronto Laboratory. She worked for many years in compiler development. Over the past few years her focus has been on object-oriented design and tool integration technology. Currently she is a member of the Design Patterns Tool team. Patsy holds a BSc in Computer Science for Data Management from the University of Toronto.

John Vlissides is a member of the research staff at the IBM T.J. Watson Research Center in Hawthorne, New York. He has practiced object-oriented technology for over a decade as a designer, implementer, researcher, lecturer, and consultant. He has published widely and is a columnist for *C++ Report*. John holds a BS degree from the University of Virginia and MS and Ph.D. degrees from Stanford University, all in Electrical Engineering.