International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 04 Issue: 11 | Nov -2017 www.irjet.net p-ISSN: 2395-0072

Generation of automatic code using Design Patterns

Sruthi Paramkusham

Dept. of Computer Science and Engineering, Hyderabad, Telangana, India

---

Abstract - *Design patterns raise the abstraction level at which people design and communicate design of object-oriented software. However, the mechanics of implementing design patterns is left 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

Expertise is an intangible but unquestionably valuable commodity. People acquire it slowly, through hard work and perseverance. Expertise distinguishes a novice from an expert, and it is difficult for experts to convey their expertise to novices. Capturing expertise is one challenge, communicating it is another, and assimilating it is yet another. Such are the difficulties of gaining proficiency in object-oriented software development. As a result, people have been slow to realize its touted benefits.

The emerging field of design patterns is a promising step toward meeting these challenges. Design pat-terns capture expertise in building object-oriented software. A design pattern describes a solution to a recurring design problem in a systematic and general way. Beyond a description of the problem and its solution, moreover, 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 design patterns. Pattern related discourse has flourished at object-oriented conferences, so much so that there is now a conference, devoted entirely to patterns. Books and articles have been published, and at least one. Non-profit organization (The Hillside Group) has been established to further the field.

One of the most widely cited books is Design Pattems: Elements of Re-usable Object -Oriented Software.t" which presents a catalog of 23 design patterns culled from numerous object-oriented systems. We refer to this book as De-sign Patterns throughout this paper.

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

Design patterns are not code; they must be implemented each time they are applied. Designer's sup-ply 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 that was needed to implement a pattern, it would not be a big chore. But often there are many trade-offs in a pattern to consider, and different trade-offs often work synergistically, resulting in a proliferation of variant implementations -too many to support through conventional code reuse techniques. Developers are therefore likely to duplicate their efforts and those of other developers each time they apply a pattern.

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. DESCRIBING DESIGN PATTERNS

To set the stage for the rest of the paper, we include here the pattern template used in the Design Patterns book. The template lends a uniform structure to the information, making design patterns easier to learn, compare, and use. We describe here each section of the template. The book contains a more detailed description of each section, followed by actual design patterns documented with the template.

Name: The Name of the pattern conveys its essence succinctly. A good name is vital, because it will be-come part of your design vocabulary.

Intent: The Intent is 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 pat-tern, if any, are included in this section.

Motivation: This section illustrates a design problem with an example, and shows how the class and object structures in the pattern solve the problem.

The example 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: This section shows a graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (GMT).

Participants: Participants are the classes or objects participating in the design pattern and their responsibilities.

Collaborations: This section describes how the participants collaborate to carry out their responsibilities.

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

Sample Code: Code fragments are included that il-lustrate how you might implement the pattern in C++ or Smalltalk.

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

Known Uses: This section contains examples of the pattern found in real systems. Each pattern includes at least two examples from different domains.

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

2.1 Generating code automatically-An example:

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

Our design pattern tool was developed to address these needs. 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 creates class declarations and definitions that implement the pattern. The user then adds this code to the rest of the application, often enhancing it with other application-specific functionality.

The tool also incorporates an on-line, hypertext rendition of Design Patterns. The patterns in the book 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.

Section pages: The tool displays the sections of a pat-tern (Intent, Motivation, etc.) in separate pages. These pages mirror the corresponding sections in the book. 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, al-lowing quick and easy cross-referencing.

Clicking on the right arrow button besides the Intent heading advances the user to the next section in sequence, in this case the Motivation section of the Composite design pattern. 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 inter-face. To jump to 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 jump's to the same section in that pattern.

2.2 Code Generation page:

In addition to the sections from the book, each design pattern in the tool is augmented with a page titled "Code Generation." This page comes immediately after the Related Patterns page

for the design pattern and can be accessed, just as other pages, either sequentially or randomly. The Code Generation page lets the user enter information with which to generate a custom implementation of the pattern. This page is fully integrated with the others: references to participants and other de-tails are actually hyperlinks back to the relevant discussion in the pattern. The effect is similar to a con-text-sensitive help system.

The Code Generation page for the Composite pattern is appeared. 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.

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 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 or she 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 in-put fields with these values, the user selects An Ex-ample from the Goal: selection list and presses "OK"

2.3 Generating declarations:

The result of choosing Generate declarations as the goal and pressing "OK" with the inputs. The page that appears contains the text of a C+ + header file declaring classes for the specified participants. The page is scrolled to show declarations for the Graphic Component abstract base class and the Composite Graphic 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 participant responsibilities described for the Composite pattern. These responsibilities can vary according to the de-sign trade-offs articulated in the pattern, and the code 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 11 to let clients recover the Composite interface when all they have are references to Graphic objects.

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 and presses "OK". 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 be-hind a trade-off, he or she can click on the appropriate link to jump back to the corresponding discussion in the pattern? When finished choosing trade-offs, the user presses "OK" to commit the changes and return to the Code Generation page. Global code generation options. Users can also control certain generation parameters that apply to all the patterns. Selecting Choose generation options from the Goal: selection list and pressing "OK" yields the page. The user can choose to

- Limit file names to an eight-plus-three character format

3. SYSTEM ARCHITECTURE

The architecture of the design pattern tool 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.

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 implementers of the tool; goals for the tool itself impact how well the end-user receives and exploits the tool. We refer to the former simply as "development goals" and the latter as "end-user goals."

We have three primary development goals:

- **Fast turnaround:** Because most of this work is new and experimental, we must be able to modify the system as quickly as possible. We cannot afford delays in implementing or testing new functionality-we have too many degrees of freedom to explore.

- **Flexibility:** Fast turnaround means little if adding a new feature requires re-implementing 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 should not force an overhaul of the entire system.

- **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.

Horizon the web based dashboard that can be used to manage/ administer OpenStack services. It can be used to manage instances and images, creates key pairs, attach volumes to instances, and manipulates Swift containers etc.

Apart from this, dashboard even gives the user access to instance console and can connect to an instance through VNC. Overall, Horizon features the following:

- **Instance Management** – Create or terminate instance, view console logs and connect through VNC, Attaching volumes, etc.

- **Access and Security Management:** Create security groups, manage key pairs, assign floating IP's, etc.

- **Flavor Management:** Manage different flavors or instance virtual hardware templates.

- **Image Management:** Edit or delete images.

- **View Service catalog.**

- **Manage users, quotas and usage for projects**.

- **User Management:** Create user, etc.

- **Volume Management:** Creating volumes and snapshots.

- **Object Store Manipulation** – Create, delete containers and objects.

- **Downloading environment variables for a project**

4. RELATED WORK

Most elements of this work have had long research histories. The pattern concept arose from the work of Christopher Alexander in the 1970s. 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 Ph.D. 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. 20 In the wake of Knuth's seminal paper on context-free languages, 21 the 1970s and 1980s 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. The foundations of today's commercial software development tools- user interface builders, 4GL (fourth-generation language) 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-is, the Web gave us all we had hoped for-a viable front-end to our design pat-tern tool -and some things we had not thought to hope for, like interactive questions and answers.

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, they offer compelling new capabilities. The Web introduced no new technologies; it just 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.

5. 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 de-sign 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 lever-age that design patterns provide. For example, the tool is limited to system design and implementation; it does not support domain analysis, requirements specification, documentation, or debugging. All of these areas stand to benefit from design patterns, though at this point it might not be clear exactly how. Then again, the principles underpinning our tool were not clear until we had experience using pat-terns for design. Application is the first and necessary step; only then can we hope to automate profitably.

6. REFERENCES

[1] Portions of this paper are adapted from Design Patterns: Elements of Reusable Object-Oriented Software by E. Gamma, R. Helm, R. Johnson, and J. Vlissides, ©1995 by Addison-Wesley Publishing Co., Reading, MA. Used by permission.

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

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Ad-dison-Wesley Publishing Co., Reading, MA (1995)..

[4] Pattern Languages of Program Design, J. O. Coplien and D. C. Schmidt, Editors, Addison-Wesley Publishing Co., Reading, MA (1995).

[5] P. Coad, D. North, and M. Mayfield, Object Models: Strategies, Patterns, and Applications, Yourdon Press, Englewood Cliffs, NJ (1995).

[6] R. Gabriel, "Pattern Languages," Journal of ObjectOriented Programming 5, No.8, 72-75 (January 1994).

[7] K. Beck, "Patterns and Software Development," Dr. Dobb's Journal 19, No.2, 18-23 (1994).

[8] J. O. Coplien, "Generative Pattern Languages: An Emerging Direction of Software Design," C++ Report 6, No.6, 18 (July/August 1994).