Chapter XXIV
Class Patterns and Templates in Software Design

Julio Sanchez
Minnesota State University, Mankato, USA

Maria P. Canton
South Central College, USA

ABSTRACT

This chapter describes the use of design patterns as reusable components in program design. The discussion includes the two core elements: the class diagram and examples implemented in code. The authors believe that although precanned patterns have been popular in the literature, it is the patterns that we personally create or adapt that are most useful. Only after gaining intimate familiarity with a particular class structure will we be able to use it in an application. In addition to the conventional treatment of class patterns, the discussion includes the notion of a class template. A template describes functionality and object relations within a single class, while patterns refer to structures of communicating and interacting classes. The class template fosters reusability by providing a guide in solving a specific implementation problem. The chapter includes several class templates that could be useful to the software developer.

DESIGN PATTERNS

Engineers and architects have reused design elements for many years (Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King, & Angel, 1977); however, the notion of reusing elements of software design dates back only to the early 1990s. The work of Anderson (1990), Coplien (1992), and Beck and Johnson (1994) set the background for the book *Design Patterns* by Gamma, Helm, Johnson, and Vlissides (1995), which many considered the first comprehensive work on the subject.

The main justification for reusing program design components is based on the fact that the design stage is one of the most laborious and time-consuming phases of program development. Design reuse is founded in the assumption that once a programmer or programming group has found a class or object structure that solves a particular design problem, this pattern can then be reused in other projects, with

Copyright © 2009, IGI Global, distributing in print or electronic forms without written permission of IGI Global is prohibited.

considerable savings in the design effort. Anyone who has participated in the development of a substantial software project appreciates the advantages of reusing program design components.

The present-day approach to design reuse is based on a model of class associations and relationships called a class pattern or an object model. In this sense, a pattern is a solution to a design problem. Therefore, a programming problem is at the origin of every pattern. From this assumption we deduce that a pattern must offer a viable solution; it must represent a class structure that can be readily coded in the language of choice.

The fact that a programming problem is at the root of every design pattern, and the assumption that the solution offered by a particular pattern must be readily implementable in code, are the premises on which we base our approach to this topic. In the context of this chapter we see a design pattern as consisting of two core elements: a class diagram and a coded example or template, fully implemented in code. Every working programmer knows how to take a piece of existing code and reengineer it to solve the problem at hand. However, snippets of code that may or may not compile correctly are more a tease than a real aide.

Although we consider that design patterns are a reasonable and practical methodology, we must also add that it is the patterns that we ourselves create, refine, or adapt that are the most useful. It is difficult to believe that we can design and code a program based on someone else's class diagrams. Program design and coding is a task too elaborate and complicated to be done by imitation or by proxy. A programmer must gain intimate familiarity with a particular class and object structure before committing to its adoption in a project. These thoughts lead to the conclusion that it is more important to explain how we can develop our own design patterns than to offer an extensive catalog of someone's class diagrams, which can be difficult to understand, and even more difficult to apply.

CLASS TEMPLATES

Occasionally, a programmer or program designer's need is not for a structure of communicating and interacting classes but for a description of the implementation of a specific functionality within a single class. In this case we can speak of a class template rather than of a pattern. The purpose of a class template is also to foster reusability by providing a specific guide for solving a particular implementation problem. In the following sections we include several class templates that could be useful to the practicing developer.

A Pattern is Born

We begin our discussion by following through the development of a design pattern, from the original problem, through a possible solution, to its implementation in code, and concluding in a general-purpose class diagram.

One of the most obvious and frequent uses of dynamic polymorphism is in the implementation of class libraries. The simplest usable architecture is by means of an abstract class and several modules in the form of derived classes that provide the specific implementations of the library's functionality. Client code accesses a polymorphic method in the base class and the corresponding implementation is selected according to the object referenced. But in the real world a library usually consists of more than one method. Since many languages allow mixing virtual and nonvirtual functions in an abstract class, it is possible to include nonvirtual methods along with virtual and pure virtual ones. The problem in this case is that abstract classes cannot be instantiated; therefore, client code cannot create an object through which it can access the nonvirtual methods in the base class. A possible but not very effective solution is to use one of the derived classes to access the nonvirtual methods in the base class. Figure 1 depicts this situation.

The first problem of the class diagram in Figure 1 is that the client code accesses the nonvirtual

43 more pages are available in the full version of this document, which may be purchased using the "Add to Cart" button on the publisher's webpage: www.igi-global.com/chapter/class-patterns-templates-software-design/21081

Related Content

Information Technology Industry Dynamics: Impact of Disruptive Innovation Strategy
Nicholas C. Georgantzasand Evangelos Katsamakas (2010). *Emerging Systems Approaches in Information Technologies: Concepts, Theories, and Applications (pp. 274-293).*
www.irma-international.org/chapter/information-technology-industry-dynamics/38185

A Performance Improvement Model for Cloud Computing Using Simulated Annealing Algorithm
Geeta Singh, Santosh Kumarand Shiva Prakash (2022). *International Journal of Software Innovation (pp. 1-17).*
www.irma-international.org/article/a-performance-improvement-model-for-cloud-computing-using-simulated-annealing-algorithm/301222

Channel Optimization for On Field Sales Force by Integration of Business Software on Mobile Platforms
Rishi Kalraand Amit Nanchahal (2009). *Software Applications: Concepts, Methodologies, Tools, and Applications (pp. 2584-2598).*
www.irma-international.org/chapter/channel-optimization-field-sales-force/29523

A Hybrid Siamese-LSTM (Long Short-Term Memory) for Classification of Alzheimer's Disease
Aparna M.and Srinivasa B. Rao (2022). *International Journal of Software Innovation (pp. 1-14).*
www.irma-international.org/article/a-hybrid-siamese-lstm-long-short-term-memory-for-classification-of-alzheimers-disease/309720

Cooperation Between Agents to Evolve Complete Programs
Ricardo Aler, David Camachoand Alfredo Moscardini (2003). *Intelligent Agent Software Engineering (pp. 213-228).*
www.irma-international.org/chapter/cooperation-between-agents-evolve-complete/24151