# SOME BACKGROUND ON DESIGN PATTERNS

The term "design patterns" sounds a bit formal to the uninitiated and can be somewhat off-putting when you first encounter it. But, in fact, design patterns are just convenient ways of reusing object-oriented code between projects and between programmers. The idea behind design patterns is simple-- write down and catalog common interactions between objects that programmers have frequently found useful.

The field of design patterns goes back at least to the early 1980s. At that time, Smalltalk was the most common OO language and C++ was still in its infancy. At that time, structured programming was a commonly-used phrased and OO programming was not yet as widely supported. The idea of programming frameworks was popular however, and as frameworks developed, some of what we now called design patterns began to emerge.

One of the frequently cited frameworks was the Model-View-Controller framework for Smalltalk [Krasner and Pope, 1988], which divided the user interface problem into three parts. The parts were referred to as a *data model* which contain the computational parts of the program, the *view,* which presented the user interface, and the c*ontroller*, which interacted between the user and the view.
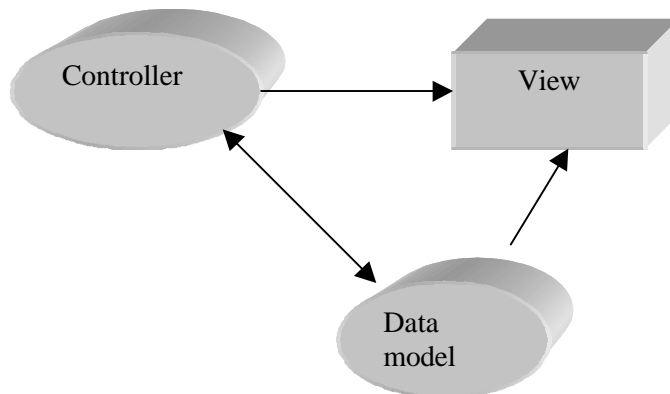
Each of these aspects of the problem is a separate object and each has its own rules for managing its data. Communication between the user, the GUI and the data should be carefully controlled and this separation of functions accomplished that very nicely. Three objects talking to each other using this restrained set of connections is an example of a powerful *design pattern*.

In other words, design patterns describe how objects communicate without become entangled in each other's data models and methods. Keeping this separation has always been an objective of good OO programming, and if you have been trying to keep objects minding their own business, you are probably using some of the common design patterns already. Interestingly enough, the MVC pattern has resurfaced now and we find it used in Java 1.2 as part of the Java Foundation Classes (JFC, or the "Swing" components).

Design patterns began to be recognized more formally in the early 1990s by Helm (1990) and Erich Gamma (1992), who described patterns incorporated in the GUI application framework, ET++. The culmination of these discussions and a number of technical meetings was the publication of the parent book in this series, *Design Patterns -- Elements of Reusable Software,* by Gamma, Helm, Johnson and Vlissides.(1995). This book, commonly referred to as the Gang of Four or "GoF" book, has had a powerful impact on those seeking to understand how to use design patterns and has become an all-time best seller. We will refer to this groundbreaking book as *Design Patterns*, throughout this book and *The Design Patterns Smalltalk Companion (*Alpert, Brown and Woolf, 1998) as the *Smalltalk Companion.*

Defining Design Patterns

We all talk about the way we do things in our everyday work, hobbies and home life and recognize repeating patterns all the time.

- Sticky buns are like dinner rolls, but I add brown sugar and nut filling to them.

- Her front garden is like mine, but, in mine I use *astilbe*.

- This end table is constructed like that one, but in this one, the doors replace drawers.

We see the same thing in programming, when we tell a colleague how we accomplished a tricky bit of programming so he doesn't have to recreate it from scratch. We simply recognize effective ways for objects to communicate while maintaining their own separate existences.

Some useful definitions of design patterns have emerged as the literature in his field has expanded:

- "Design patterns are recurring solutions to design problems you see over *et. al*., 1998).

- "Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development." (Pree, 1994)

- "Design patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detailed design… and implementation." (Coplien & Schmidt, 1995).

- "A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it" (Buschmann, *et. al.* 1996)

- "Patterns identify and specify abstractions that are above the level of single classes and instances, or of components." (Gamma, et al., 1993)

But while it is helpful to draw analogies to architecture, cabinet making and logic, design patterns are not just about the design of objects, but about the *communication* between objects. In fact, we sometimes think of them as *communication patterns*. It is the design of simple, but elegant, methods of communication that makes many design patterns so important.

Design patterns can exist at many levels from very low level specific solutions to broadly generalized system issues. There are now in fact hundreds of patterns in the literature. They have been discussed in articles and at conferences of all levels of granularity. Some are examples which have wide applicability and a few (Kurata, 1998) solve but a single problem.

It has become apparent that you don't just *write* a design pattern off the top of your head. In fact, most such patterns are *discovered* rather than written. The process of looking for these patterns is called "pattern mining," and is worthy of a book of its own.

The 23 design patterns selected for inclusion in the original *Design Patterns* book were ones which had several known applications and which were on a middle level of generality, where they could easily cross application areas and encompass several objects.

The authors divided these patterns into three types creational, structural and behavioral.

- *Creational patterns* are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.

- *Structural patterns* help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.

- *Behavioral patterns* help you define the communication between objects in your system and how the flow is controlled in a complex program.

We'll be looking at Java versions of these patterns in the chapters that follow.

This Book and its Parentage

Design Patterns is a catalog of 23 generally useful patterns for writing object-oriented software. It is written as a catalog with short examples and substantial discussions of how the patterns can be constructed and applied. Most of its examples are in C++, with a few in Smalltalk. The *Smalltalk Companion* (Alpert, 1998) follows a similar approach, but with somewhat longer examples, all in Smalltalk. Further, the authors present some additional very useful advice on implementing and using these patterns.

This book takes a somewhat different approach; we provide at least one complete, visual Java program for each of the 23 patterns. This way you can not only examine the code snippets we provide, but run, edit and modify the complete working programs on the accompanying CD-ROM. You'll find a list of all the programs on the CD-ROM in Appendix A.

The Learning Process

We have found learning Design patterns is a multiple step process.

1. Acceptance
2. Recognition
3. Internalization

First, you accept the premise that design patterns are important in your work. Then, you recognize that you need to read about design patterns in order to know when you might use them. Finally, you internalize the patterns in sufficient detail that you know which ones might help you solve a given design problem.

For some lucky people, design patterns are obvious tools and they grasp their essential utility just by reading summaries of the patterns. For many of the rest of us, there is a slow induction period after we've read about a pattern followed by the proverbial "Aha!" when we see how we can apply them in our work. This book helps to take you to that final stage of internalization by providing complete, working programs that you can try out for yourself.

The examples in *Design Patterns* are brief, and are in C++ or in some cases, Smalltalk. If you are working in another language it is helpful to have the pattern examples in your language of choice. This book attempts to fill that need for Java programmers.

A set of Java examples takes on a form that is a little different than in C++, because Java is more strict in its application of OO precepts -- you can't have global variables, data structures or pointers. In addition, we'll see that the Java interfaces and abstract classes are a major contributor to how we build Java design patterns.

Studying Design Patterns

There are several alternate ways to become familiar with these patterns. In each approach, you should read this book and the parent *Design Patterns* book in one order or the other. We also strongly urge you to read the *Smalltalk Companion* for completeness, since it provides an alternate description of each of the patterns. Finally, there are a number of web sites on learning and discussing Design Patterns for you to peruse.

Notes on Object Oriented Approaches

The fundamental reason for using varies design patterns is to keep classes separated and prevent them from having to know too much about one another. There are a number of strategies that OO programmers use to achieve this separation, among them encapsulation and inheritance.

Nearly all languages that have OO capabilities support inheritance. A class that inherits from a parent class has access to all of the methods of that parent class. It also has access to all of its non-private variables. However, by starting your inheritance hierarchy with a complete, working class you may be unduly restricting yourself as well as carrying along specific method implementation baggage. Instead, *Design Patterns* suggests that you always

Program to an interface and not to an implementation.

Purring this more succinctly, you should define the top of any class hierarchy with an *abstract* class, which implements no methods, but simply defines the methods that class will support. Then, in all of your derived classes you have more freedom to implement these methods as most suits your purposes.

The other major concept you should recognize is that of *object composition.* This is simply the construction of objects that contain others: encapsulation of

several objects inside another one. While many beginning OO programmers use inheritance to solve every problem, as you begin to write more elaborate programs, the merits of object composition become apparent. Your new object can have the interface that is best for what you want to accomplish without having all the methods of the parent classes. Thus, the second major precept suggested by *Design Patterns* is

Favor object composition over inheritance.

At first this seems contrary to the customs of OO programming, but you will see any number of cases among the design patterns where we find that inclusion of one or more objects inside another is the preferred method.

The Java Foundation Classes

The Java Foundation Classes (JFC) which were introduced after Java 1.1 and incorporated into Java 1.2 are a critical part of writing good Java programs. These were also known during development as the "Swing" classes and still are informally referred to that way. They provide easy ways to write very professional-looking user interfaces and allow you to vary the look and feel of your interface to match the platform your program is running on. Further, these classes themselves utilize a number of the basic design patterns and thus make extremely good examples for study.

Nearly all of the example programs in this book use the JFC to produce the interfaces you see in the example code. Since not everyone may be familiar with these classes, and since we are going to build some basic classes from the JFC to use throughout our examples, we take a short break after introducing the creational patterns and spend a chapter introducing the JFC. While the chapter is not a complete tutorial in every aspect of the JFC, it does introduce the most useful interface controls and shows how to use them.

Many of the examples do require that the JFC libraries are installed, and we describe briefly what Jar files you need in this chapter as well.

Java Design Patterns

Each of the 23 design patterns in *Design Patterns* is discussed in the chapters that follow, along with at least one working program example for that pattern. The authors of *Design Patterns* have suggested that every pattern start with an abstract class and that you derive concrete working

classes from that abstraction. We have only followed that suggestion in cases where there may be several examples of a pattern within a program. In other cases, we start right in with a concrete class, since the abstract class only makes the explanation more involved and adds little to the elegance of the implementation.

James W. Cooper
Wilton, Connecticut
Nantucket, Massachusetts