# Design Patterns: Overview and BackGround

## What are *design patterns*?

Here are what some authors have said about design patterns:

- Patterns identify and specify abstractions that are above the level of single classes and instances, or of components. [Gamma, Helm, Johnson, and Vlissides, 1993]
- Design patterns are recurring solutions to design problems you see over and over. [Alpert, Brown and Woolf, 1998]
- A design pattern describes how objects communicate without becoming entangled in each other's data models and methods. [Cooper 2000]
- Software design patterns are schematic descriptions of solutions to recurring problems in software design. [Jia, 2003]
- A design pattern is a recipe for solving a certain type of design problem that captures the high-level objects, their interactions, and their behaviors. [Wang, 2003]

## A Brief History of Design Patterns and the "Gang of Four"

The concept of *patterns* (in general) was originally articulated by Christopher Alexander and colleagues in the late 1970s [*The Timeless Way of Building*, 1979; *A Pattern Language—Towns, Buildings, Construction*, 1977] (They had 253 patterns.)

Somewhat later, it was recognized that there are many similarities between software design and architectural design:

- Both are creative processes that unfold within a large design space, which comprises all possible designs.
- The resulting design must satisfy the customer's needs.
- The resulting design must be feasible to engineer.
- The designers must satisfy many competing constraints and requirements.
- The designers must seek certain intrinsic yet unquantifiable qualities, such as elegance and extensibility.

Pioneering work in this area was done by the so-called "Gang of Four". [See their seminal text on the subject: *Design Patterns (Elements of Reusable Object-Oriented Software)*, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1995]

*Design Patterns* became a best-seller, is now regarded as a "ground-breaking" work, and the rest is history: Design patterns have been accepted as a very useful tool by the software development community at large.

Many additional patterns have been identified and described, and more appear all the time.

In a very small way, you are familiar with a number of "mini design patterns". A very small pattern in a programming language is sometimes called an *idiom*. Can you think of any examples?

## The Three-Step Design Pattern Learning Process

1. *Acceptance* Accept the fact design patterns are going to be important in your software development work.
2. *Recognition* Recognize the need to read about and "watch for" design patterns in order to know when you might use them.
3. *Internalization* Finally, you "internalize" the patterns in sufficient detail that you know ("instinctively"?) which one(s) might help you solve a given problem.

## The 23 Original Design Patterns (in Three Categories) (from *Java Design Patterns*, JW Cooper, 2000)

Design patterns are essentially language-independent, but are often presented in a language-sensitive way, using either Smalltalk, C++ or Java to illustrate or explain how the pattern works.

The process of "looking for patterns" is called *pattern mining*. Patterns need to be "recognized" or "discovered", and new ones are "showing up" and being described continually. The original 23 design patterns of the Gang of Four are often grouped into the following three categories:

- *Creational Patterns*
  This group of design patterns deals with the process of object creation: They create objects for you, and help you avoid having to instantiate objects directly. Your program gains more flexibility in deciding which objects need to be created for a given situation.
  1. *Simple Factory*
     This design pattern returns an instance of one of several possible classes, depending on the data provided to it.
  2. *Factory Method*
     This design pattern provides a simple decision-making class that returns one of several possible subclasses of an abstract base class, depending on the data that are provided to it.
  3. *Abstract Factory*
     This design pattern provides an interface to create and return one of several families of related objects.
  4. *Singleton*
     This design pattern represents a class of which there may be no more than one instance. It thus provides a single point of access to that instance.
  5. *Builder*
     This design pattern separates the construction of a complex object from its representation, so that several different representations can be created, depending on the needs of the program.
  6. *Prototype*
     This design pattern starts with an instantiated class, which it copies or clones to make new instances. These instances can then be further tailored using their public methods.
- *Structural Patterns*
  This group of design patterns deals with the static composition and structure of objects and classes: They help you compose groups of objects into larger structures, such as complex user interfaces and accounting data.
  1. *Adapter*
     This design pattern can be used to make one class interface match another, for easier programming.
  2. *Bridge*
     This design pattern separates an object's interface from its implementation, so you can vary them separately.
  3. *Composite*
     This design pattern creates an object (a composition of objects), each of which may be a simple or composite object.
  4. *Decorator*
     This design pattern lets you add responsibilities to objects dynamically.
  5. *Façade*
     This design pattern is used to make a single class represent an entire subsystem.
  6. *Flyweight*
     This design pattern is used for sharing objects, where each instance does not contain its own state, but stores it externally. This approach allows efficient sharing of objects to save space when there are many instances, but only a few different types.
  7. *Proxy*
     This design pattern creates a simple object that takes the place of a more complex object which may be invoked later, such as when the program runs in a networked environment.
- *Behavioral Patterns*
  This group of design patterns deals primarily with dynamic interaction among classes and objects: They help you define the communication between objects in your system and how the flow is controlled in a complex program.
  1. *Chain of Responsibility*
     This design pattern allows decoupling between objects by passing a request from one object to the next in a chain until the request is recognized.
  2. *Command*
     This design pattern utilizes simple objects to represent the execution of software commands, and allows you to support logging and undoable commands.
  3. *Interpreter*
     This design pattern provides a definition of how to include language elements within a program.
  4. *Iterator*
     This design pattern formalizes the way we move through a list of data within a class.
  5. *Mediator*
     This design pattern defines how communication between objects can be simplified by using a separate object to keep all objects from having to know about each other.
  6. *Observer*
     This design pattern defines how multiple objects can be notified of a change.
  7. *State*
     This design pattern allows an object to modify its behavior when its internal state changes.
  8. *Strategy*
     This design pattern encapsulates an algorithm inside a class.
  9. *Template Method*
     This design pattern provides an abstract definition of an algorithm.
  10. *Visitor*
      This design pattern adds polymorphic functions to a class non-invasively.

## Other Design Patterns

As we hinted at above, there are now at least hundreds, and perhaps thousands, of design patterns being used by software developers. Many of them apply to very specific situations, and are of little interest outside a particular field of endeavor. But there are also quite a number of additional design patterns of general applicability. We may add to the list given below, from time to time, as we encounter new patterns that are of some use or interest.

- *Architectural Patterns*
  This group of design patterns deals with the coupling of subsystems of a system, and tends to promote loose coupling of those subsystems. Thus these patterns specify how these subsystems interact with one another.
  1. *Model-View-Controller*
     This design pattern separates application data (contained in the *Model*) from graphical or other presentation components (the *View*) and the input-processing logic (the controller).
  2. *Layers*
     This design pattern divides the functionality of a system into different *layers*. Each layer contains a set of system responsibilities and depends only on the "services" provided by the next lower layer. The idea is that a designer should be able to modify one layer without having to modify any of the other layers. A typical (and common) example of this is the so-called *three-tier application model*, in which
     - The "top tier" or "client tier" is the application's user interface, and might be something like a web browser.
     - The "bottom tier" or "information tier" maintains information for the application, usually storing it in a database of some kind.
     - The "middle tier" acts as an intermediary between the client tier and the information tier. This middle tier processes client requests and, based on these requests, reads data from and writes data to the information tier. Then the middle tier processes data from the information tier and presents the results to the client tier.

## The Parts of Any Design Pattern

At a minimum, any design pattern must provide the following:

- The pattern name
- The pattern "intent" (a description of the problem the design pattern is intended to solve)
- A description of the solution (possibly with accompanying code and/or usage examples)
- A description of the "consequences" (design decisions and compromises made, which may be of help to a user when deciding whether this particular design pattern meets the user's design goals)

There are more elaborate schemes. For example, *Design Patterns* uses the following sections in its descriptive scheme for each pattern:

- Name
- Intent
- Also know as (alternate terminology)
- Applicability
- Structure (class or object diagram)
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

List of All Topics