**Design Patterns**

**What's a design pattern?**

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

You can’t just find a pattern and copy it into your program, the way you can with off-the-shelf functions or libraries. The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your own program.

Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems. While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution. The code of the same pattern applied to two different programs may be different.

An analogy to an algorithm is a cooking recipe: both have clear steps to achieve a goal. On the other hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.

**What does the pattern consist of?**

Most patterns are described very formally so people can reproduce them in many contexts. Here are the sections that are usually present in a pattern description:

* **Intent** of the pattern briefly describes both the problem and the solution.
* **Motivation** further explains the problem and the solution the pattern makes possible.
* **Structure** of classes shows each part of the pattern and how they are related.
* **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

Some pattern catalogs list other useful details, such as applicability of the pattern, implementation steps and relations with other patterns.

**History of patterns**

Who invented patterns? That’s a good, but not a very accurate, question. Design patterns aren’t obscure, sophisticated concepts—quite the opposite. Patterns are typical solutions to common problems in object-oriented design. When a solution gets repeated over and over in various projects, someone eventually puts a name to it and describes the solution in detail. That’s basically how a pattern gets discovered.

The concept of patterns was first described by Christopher Alexander in **A Pattern Language: Towns, Buildings, Construction**. The book describes a “language” for designing the urban environment. The units of this language are patterns. They may describe how high windows should be, how many levels a building should have, how large green areas in a neighborhood are supposed to be, and so on.

The idea was picked up by four authors: Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. In 1994, they published Design Patterns: Elements of Reusable Object-Oriented Software, in which they applied the concept of design patterns to programming. The book featured 23 patterns solving various problems of object-oriented design and became a best-seller very quickly. Due to its lengthy name, people started to call it “the book by the gang of four” which was soon shortened to simply “the GoF book”.

Since then, dozens of other object-oriented patterns have been discovered. The “pattern approach” became very popular in other programming fields, so lots of other patterns now exist outside of object-oriented design as well.

Why should I learn patterns?

The truth is that you might manage to work as a programmer for many years without knowing about a single pattern. A lot of people do just that. Even in that case, though, you might be implementing some patterns without even knowing it. So why would you spend time learning them?

Design patterns are a toolkit of tried and tested solutions to common problems in software design. Even if you never encounter these problems, knowing patterns is still useful because it teaches you how to solve all sorts of problems using principles of object-oriented design.

Design patterns define a common language that you and your teammates can use to communicate more efficiently. You can say, “Oh, just use a Singleton for that,” and everyone will understand the idea behind your suggestion. No need to explain what a singleton is if you know the pattern and its name.

**Criticism of patterns**

It seems like only lazy people haven’t criticized design patterns yet. Let’s take a look at the most typical arguments against using patterns.

**Kludges for a weak programming language**

Usually the need for patterns arises when people choose a programming language or a technology that lacks the necessary level of abstraction. In this case, patterns become a kludge that gives the language much-needed super-abilities.

For example, the Strategy pattern can be implemented with a simple anonymous (lambda) function in most modern programming languages.

**Inefficient solutions**

Patterns try to systematize approaches that are already widely used. This unification is viewed by many as a dogma, and they implement patterns “to the letter”, without adapting them to the context of their project.

**Unjustified use**

If all you have is a hammer, everything looks like a nail.

This is the problem that haunts many novices who have just familiarized themselves with patterns. Having learned about patterns, they try to apply them everywhere, even in situations where simpler code would do just fine.

**Classification of patterns**

Design patterns differ by their complexity, level of detail and scale of applicability to the entire system being designed. I like the analogy to road construction: you can make an intersection safer by either installing some traffic lights or building an entire multi-level interchange with underground passages for pedestrians.

The most basic and low-level patterns are often called *idioms*. They usually apply only to a single programming language.

The most universal and high-level patterns are *architectural patterns*. Developers can implement these patterns in virtually any language. Unlike other patterns, they can be used to design the architecture of an entire application.

In addition, all patterns can be categorized by their intent, or purpose. This book covers three main groups of patterns:

**Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.

**Structural patterns** explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

**Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.